

Virtex-II Pro™ Platform FPGA Developer's Kit

March 2002 Release





The Xilinx logo shown above is a registered trademark of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

"Xilinx" and the Xilinx logo are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, XC5210 are registered Trademarks of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Bencher, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Bencher, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, Nano-Blaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, Rocket I/O, SelectI/O, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM IBM Logo PowerPC PowerPC Logo Blue Logic CoreConnect CodePack

All other trademarks are the property of their respective owners.

Xilinx does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 2002 Xilinx, Inc. All Rights Reserved.

Virtex-II Pro™ Platform FPGA Developer's Kit

March 2002 Release

Revision History

The following table summarizes changes made to each version of this document.

Date	Version	Revision
03/04/02	March 2002	Initial Xilinx release.

Table of Contents

Schedule of Figures	33
Schedule of Tables	47

Introduction to Documentation

Developer's Kit Documentation Organization	67
Getting Started	68
Using the Reference Systems	68
Learning About the Virtex-II Pro Device	68
Learning About the PowerPC 405 CPU	68
Learning About the Rocket I/O Multi-Gigabit Transceiver	68
Learning About the Virtex-II Pro Developer's Kit Design Flow	69
Learning About the Software Development Tools	69
Learning About the Hardware IP	69
Learning About the Software IP and Applications	69
Conclusion	69
Further Reading	70
Documentation Provided by Xilinx	70
IBM® CoreConnect™ Documentation	70
Software Development Documentation	71

Volume 1: Advance Product Specification

Data Sheet Module 1:

Virtex-II Pro™ Platform FPGAs: Introduction and Overview

Summary of Virtex-II Pro Features	75
Rocket I/O Features	75
PowerPC RISC Core Features	75
Virtex-II Pro Platform FPGA Technology	76
General Description	76
Architecture	77
Virtex-II Pro Array Overview	77
Virtex-II Pro Features	77
IP Core and Reference Support	80
Hardware Cores	80
Software Cores	80
Virtex-II Pro Device/Package Combinations and Maximum I/Os	80
Virtex-II Pro Ordering Information	81
Revision History	81
Virtex-II Pro Data Sheet Modules	81

Data Sheet Module 2:

Virtex-II Pro™ Platform FPGAs: Functional Description

Virtex-II Pro Array Functional Description	83
Virtex-II Pro Compared to Virtex-II Devices.....	83
Functional Description: Rocket I/O Multi-Gigabit Transceiver (MGT)	83
Overview	83
Clock Synthesizer	86
Clock and Data Recovery	86
Transmitter	86
Receiver.....	87
Loopback	87
Elastic and Transmitter Buffers	87
CRC.....	89
Configuration.....	89
Reset / Power Down.....	89
Power Sequencing	89
Functional Description: Processor Block	90
Processor Block Overview	90
Embedded PowerPC 405 RISC Core	90
On-Chip Memory (OCM) Controllers.....	90
Clock/Control Interface Logic	91
CPU-FPGA Interfaces	91
CoreConnect™ Bus Architecture	92
Functional Description: PowerPC 405 Core	93
PPC405 Core.....	93
Instruction and Data Cache	93
Fetch and Decode Logic	94
Execution Unit	94
Memory Management Unit (MMU)	94
Timers.....	95
Interrupts.....	95
Debug Logic	95
Big Endian and Little Endian Support	96
Functional Description: FPGA	96
Input/Output Blocks (IOBs)	96
Digitally Controlled Impedance (DCI).....	101
Configurable Logic Blocks (CLBs)	104
3-State Buffers	112
CLB/Slice Configurations.....	112
18 Kb Block SelectRAM Resources	113
18-Bit x 18-Bit Multipliers.....	116
Global Clock Multiplexer Buffers	117
Digital Clock Manager (DCM)	119
Routing.....	122
Configuration.....	123
Revision History	126
Virtex-II Pro Data Sheet Modules	126

Data Sheet Module 3:

Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics

Virtex-II Pro Electrical Characteristics	127
Virtex-II Pro DC Characteristics.....	127
Power-On Power Supply Requirements.....	129

Select I/O DC Input and Output Levels	130
LDT DC Specifications (LDT_25)	131
LVDS DC Specifications (LVDS_25)	131
Extended LVDS DC Specifications (LVDSEXT_25)	131
Rocket I/O DC Input and Output Levels	132
Virtex-II Pro Performance Characteristics	133
Virtex-II Pro Switching Characteristics	135
Testing of Switching Characteristics	135
PowerPC Switching Characteristics	136
Rocket I/O Switching Characteristics	139
IOB Input Switching Characteristics	144
IOB Input Switching Characteristics Standard Adjustments	145
IOB Output Switching Characteristics	146
IOB Output Switching Characteristics Standard Adjustments	147
Clock Distribution Switching Characteristics	153
CLB Switching Characteristics	153
CLB Distributed RAM Switching Characteristics	154
CLB Shift Register Switching Characteristics	154
Multiplier Switching Characteristics	155
Block SelectRAM Switching Characteristics	156
TBUF Switching Characteristics	156
JTAG Test Access Port Switching Characteristics	156
Virtex-II Pro Pin-to-Pin Output Parameter Guidelines	157
Global Clock Input to Output Delay for LVCMOS25, 12 mA, Fast Slew Rate, With DCM	157
Global Clock Input to Output Delay for LVCMOS25, 12 mA, Fast Slew Rate, Without DCM	158
Virtex-II Pro Pin-to-Pin Input Parameter Guidelines	159
Global Clock Set-Up and Hold for LVCMOS25 Standard, With DCM	159
Global Clock Set-Up and Hold for LVCMOS25 Standard, Without DCM	160
DCM Timing Parameters	160
Operating Frequency Ranges	160
Input Clock Tolerances	162
Output Clock Jitter	163
Output Clock Phase Alignment	163
Miscellaneous Timing Parameters	164
Frequency Synthesis	164
Parameter Cross-Reference	165
Revision History	165
Virtex-II Pro Data Sheet Modules	165

Data Sheet Module 4:

Virtex-II Pro™ Platform FPGAs: Pinout Information

Virtex-II Pro Device/Package Combinations and Maximum I/Os	167
Virtex-II Pro Pin Definitions	168
Pin Definitions	169
FG256 Fine-Pitch BGA Package	171
FG256 Fine-Pitch BGA Package Specifications (1.00mm pitch)	179
FG456 Fine-Pitch BGA Package	180
FG456 Fine-Pitch BGA Package Specifications (1.00mm pitch)	194
FF672 Flip-Chip Fine-Pitch BGA Package	195
FF672 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)	215

FF896 Flip-Chip Fine-Pitch BGA Package	216
FF896 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)	241
FF1152 Flip-Chip Fine-Pitch BGA Package	242
FF1152 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)	275
FF1517 Flip-Chip Fine-Pitch BGA Package	276
FF1517 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)	314
BF957 Flip-Chip BGA Package	315
BF957 Flip-Chip BGA Package Specifications (1.27mm pitch)	341
Revision History	342
Virtex-II Pro Data Sheet Modules	342

Volume 2: PowerPC® 405 Processor

Volume 2(a): PPC405 User Manual

About This Book

Document Organization	347
Document Conventions	348
General Conventions	348
Instruction Fields	349
Pseudocode Conventions	351
Operator Precedence.....	353
Registers	353
Terms	354
Additional Reading	357

Chapter 1: Introduction to the PPC405

PowerPC Architecture Overview	359
PowerPC Architecture Levels.....	360
PowerPC Embedded-Environment Architecture	362
PowerPC Book-E Architecture	365
PPC405 Features	365
Privilege Modes	366
Address Translation Modes.....	367
Addressing Modes	367
Data Types.....	367
Register Set Summary.....	368
PPC405 Organization.....	370

Chapter 2: Operational Concepts

Execution Model	377
Synchronization Operations	378
Context Synchronization	378
Execution Synchronization	378
Storage Synchronization.....	379
Processor Operating Modes	379

Privileged Mode	379
User Mode	380
Memory Organization	380
Effective-Address Calculation	380
Physical Memory	381
Virtual Memory	381
Memory Management	381
Addressing Modes	382
Operand Conventions	383
Byte Ordering	385
Operand Alignment	389
Instruction Conventions	390
Instruction Forms	390
Instruction Classes	391
PowerPC Book-E Instruction Classes	393

Chapter 3: User Programming Model

User Registers	395
Special-Purpose Registers (SPRs)	396
General-Purpose Registers (GPRs)	396
Condition Register (CR)	397
Fixed-Point Exception Register (XER)	399
Link Register (LR)	399
Count Register (CTR)	400
User-SPR General-Purpose Register	400
SPR General-Purpose Registers	401
Time-Base Registers	401
Exception Summary	402
Branch and Flow-Control Instructions	403
Conditional Branch Control	403
Branch Instructions	404
Branch Prediction	406
Branch-Target Address Calculation	408
Condition-Register Logical Instructions	412
System Call	412
System Trap	413
Integer Load and Store Instructions	414
Operand-Address Calculation	414
Load Instructions	417
Store Instructions	420
Load and Store with Byte-Reverse Instructions	421
Load and Store Multiple Instructions	422
Load and Store String Instructions	423
Integer Instructions	425
Arithmetic Instructions	426
Logical Instructions	431
Compare Instructions	434
Rotate Instructions	435
Shift Instructions	439
Multiply-Accumulate Instruction-Set Extensions	441
Modulo and Saturating Arithmetic	441
Multiply-Accumulate Instructions	442
Negative Multiply-Accumulate Instructions	449
Multiply Halfword to Word Instructions	455

Floating-Point Emulation	458
Processor-Control Instructions.....	458
Condition-Register Move Instructions.....	459
Special-Purpose Register Instructions.....	460
Synchronizing Instructions	460
Implementation of ieio and sync Instructions.....	461
Synchronization Effects of PowerPC Instructions.....	461
Semaphore Synchronization.....	462
Memory-Control Instructions.....	463

Chapter 4: PPC405 Privileged-Mode Programming Model

Privileged Registers	465
Special-Purpose Registers	467
Machine-State Register	467
SPR General-Purpose Registers.....	468
Processor-Version Register	469
Device Control Registers.....	470
Privileged Instructions.....	470
System Linkage.....	470
Processor-Control Instructions.....	471
Processor Wait State	472

Chapter 5: Memory-System Management

Memory-System Organization	473
Memory-System Features	474
Cache Organization.....	474
Instruction-Cache Operation	477
Data-Cache Operation	479
Data-Cache Performance.....	481
Accessing Memory.....	483
Memory Coherency.....	484
Atomic Memory Access.....	484
Ordering Memory Accesses.....	484
Preventing Inappropriate Speculative Accesses.....	485
Memory-System Control.....	487
Storage Attributes.....	487
Storage-Attribute Control Registers	488
Cache Control.....	492
Cache Instructions	492
Core-Configuration Register.....	495
Software Management of Cache Coherency	499
How Coherency is Lost	499
Enforcing Coherency With Software.....	501
Self-Modifying Code.....	503
Cache Debugging.....	504
icread Instruction	504
dcread Instruction	505

Chapter 6: Virtual-Memory Management

Real Mode.....	507
Virtual Mode	508

Process-ID Register	510
Page-Translation Table	510
Translation Look-Aside Buffer	511
TLB Entries	512
TLB Access.....	515
TLB-Access Failures	516
Virtual-Mode Access Protection	518
TLB Access-Protection Controls.....	518
Zone Protection.....	518
Effect of Access Protection on Cache-Control Instructions.....	519
UTLB Management	521
Recording Page Access and Page Modification	522
Maintaining Shadow-TLB Consistency	523

Chapter 7: Exceptions and Interrupts

Overview	525
Synchronous and Asynchronous Exceptions.....	526
Precise and Imprecise Interrupts	526
Partially-Executed Instructions	526
PPC405D5 Exceptions and Interrupts	527
Critical and Noncritical Exceptions	528
Transferring Control to Interrupt Handlers	528
Returning from Interrupt Handlers.....	530
Simultaneous Exceptions and Interrupt Priority	531
Persistent Exceptions and Interrupt Masking	532
Interrupt-Handling Registers	533
Machine-State Register Following an Interrupt.....	533
Save/Restore Registers 0 and 1	534
Save/Restore Registers 2 and 3	535
Exception-Vector Prefix Register	536
Exception-Syndrome Register	536
Data Exception-Address Register	538
Interrupt Reference	538
Critical-Input Interrupt (0x0100)	539
Machine-Check Interrupt (0x0200)	540
Data-Storage Interrupt (0x0300)	542
Instruction-Storage Interrupt (0x0400)	544
External Interrupt (0x0500)	545
Alignment Interrupt (0x0600)	546
Program Interrupt (0x0700)	547
FPU-Unavailable Interrupt (0x0800).....	549
System-Call Interrupt (0x0C00)	550
APU-Unavailable Interrupt (0x0F20)	551
Programmable-Interval Timer Interrupt (0x1000).....	552
Fixed-Interval Timer Interrupt (0x1010)	553
Watchdog-Timer Interrupt (0x1020).....	554
Data TLB-Miss Interrupt (0x1100).....	555
Instruction TLB-Miss Interrupt (0x1200).....	556
Debug Interrupt (0x2000)	557

Chapter 8: Timer Resources

Time Base	560
Reading and Writing the Time Base.....	561

Computing Time of Day.....	562
Timer-Event Registers	563
Programmable-Interval Timer Register	563
Timer-Control Register.....	564
Timer-Status Register.....	565
Timer-Event Interrupts	565
Watchdog-Timer Events.....	566
Programmable-Interval Timer Events.....	568
Fixed-Interval Timer Events	569

Chapter 9: Debugging

Debug Modes	572
Internal-Debug Mode	572
External-Debug Mode.....	572
Debug-Wait Mode.....	573
Real-Time Trace-Debug Mode	573
Debug Registers	573
Debug-Control Registers.....	574
Debug-Status Register	577
Instruction Address-Compare Registers.....	578
Data Address-Compare Registers.....	579
Data Value-Compare Registers	579
Debug Events	579
Instruction-Complete Debug Event.....	581
Branch-Taken Debug Event.....	582
Exception-Taken Debug Event.....	582
Trap-Instruction Debug Event.....	582
Unconditional Debug Event	583
Instruction Address-Compare Debug Event.....	583
Data Address-Compare Debug Event.....	585
Data Value-Compare Debug Event	589
Imprecise Debug Event	592
Freezing the Timers.....	592
Debug Interface	593
JTAG Debug Port.....	593
JTAG Connector	593
BSDL.....	595

Chapter 10: Reset and Initialization

Reset	597
Processor State After Reset	597
First Instruction	599
Initialization	599
Sample Initialization Code.....	601

Chapter 11: Instruction Set

Instruction Encoding	606
Split-Field Notation.....	607
Alphabetical Instruction Listing	607

Appendix A: Register Summary

Register Cross-Reference	803
General-Purpose Registers.....	804
Machine-State Register and Condition Register.....	806
Special-Purpose Registers	806
Time-Base Registers	811
Device Control Registers	811

Appendix B: Instruction Summary

Instructions Sorted by Mnemonic	813
Instructions Sorted by Opcode	817
Instructions Grouped by Function	822
Instructions Grouped by Form.....	828
Instruction Set Information	833
List of Mnemonics and Simplified Mnemonics	838

Appendix C: Simplified Mnemonics

Branch Instructions	857
True/False Conditional Branches.....	857
Comparison Conditional Branches.....	860
Branch Prediction	863
Compare Instructions	864
CR-Logical Instructions	864
Rotate and Shift Instructions	865
Special-Purpose Registers	866
Subtract Instructions	867
TLB-Management Instructions	868
Trap Instructions	868
Other Simplified Mnemonics	870
No Operation	870
Load Immediate.....	870
Load Address.....	870
Move Register	870
Complement Register	870
Move to Condition Register.....	871

Appendix D: Programming Considerations

Synchronization Examples	873
Fetch and No-Op	874
Fetch and Store	874
Fetch and Add.....	874
Fetch and AND	874
Test and Set	874
Compare and Swap.....	875
Lock Acquisition and Release.....	875
List Insertion	876
Multiple-Precision Shifts	876
Code Optimization Guidelines	878

Conditional Branches.....	878
Floating-Point Emulation	879
Cache Usage	879
Alignment.....	880
Instruction Performance.....	880
General Rules	880
Branches.....	880
Multiplies.....	881
Scalar Load Instructions	882
Scalar Store Instructions	883
String and Multiple Instructions	883
Instruction Cache Misses.....	884

Appendix E: PowerPC® 6xx/7xx Compatibility

Registers	885
Machine-State Register	887
Processor-Version Register	888
Memory Management.....	888
Memory Translation	888
Memory Protection	889
Memory Attributes.....	889
Cache Management.....	890
Exceptions.....	890
Timer Resources.....	891
Other Differences.....	892
Instructions.....	892
Endian Support.....	892
Debug Resources	892
Power Management	892

Appendix F: PowerPC® Book-E Compatibility

Registers	893
Machine-State Register	895
Processor-Version Register	896
Memory Management.....	896
Memory Translation	896
Memory Protection	897
Memory Attributes.....	897
Caches.....	898
Memory Synchronization.....	898
Exceptions.....	898
Timer Resources.....	900
Other Differences.....	900
Instructions.....	900
Debug Resources	900

Index.....	901

Volume 2(b): PPC405 Processor Block Manual

About This Book

Document Organization	909
Document Conventions	909
General Conventions	909
Registers	910
Terms	910
Additional Reading	913

Chapter 1: Introduction to the PowerPC® 405 Processor

PowerPC Architecture	915
PowerPC Embedded-Environment Architecture	916
PPC405 Software Features	918
Privilege Modes	920
Address Translation Modes	920
Addressing Modes	920
Data Types	920
Register Set Summary	921
PPC405 Hardware Organization	923
Central-Processing Unit	924
Exception Handling Logic	925
Memory Management Unit	925
Instruction and Data Caches	926
Timer Resources	926
Debug	927
PPC405 Interfaces	927
PPC405 Performance	928

Chapter 2: Input/Output Interfaces

Signal Naming Conventions	932
Clock and Power Management Interface	933
CPM Interface I/O Signal Summary	933
CPM Interface I/O Signal Descriptions	934
CPU Control Interface	937
CPU Control Interface I/O Signal Summary	937
CPU Control Interface I/O Signal Descriptions	937
Reset Interface	939
Reset Requirements	939
Reset Interface I/O Signal Summary	940
Reset Interface I/O Signal Descriptions	940
Instruction-Side Processor Local Bus Interface	943
Instruction-Side PLB Operation	943
Instruction-Side PLB I/O Signal Table	945
Instruction-Side PLB Interface I/O Signal Descriptions	946
Instruction-Side PLB Interface Timing Diagrams	954
Data-Side Processor Local Bus Interface	965
Data-Side PLB Operation	965
Data-Side PLB Interface I/O Signal Table	967

Data-Side PLB Interface I/O Signal Descriptions.....	969
Data-Side PLB Interface Timing Diagrams.....	980
Device-Control Register Interface	994
DCR Interface I/O Signal Summary	996
DCR Interface I/O Signal Descriptions.....	997
DCR Interface Timing Diagrams.....	999
External Interrupt Controller Interface	1004
EIC Interface I/O Signal Summary	1004
EIC Interface I/O Signal Descriptions.....	1004
JTAG Interface	1006
JTAG Interface I/O Signal Table.....	1007
JTAG Interface I/O Signal Descriptions	1008
Debug Interface	1011
Debug Interface I/O Signal Summary	1011
Debug Interface I/O Signal Descriptions	1011
Trace Interface	1014
Trace Interface Signal Summary	1014
Trace Interface I/O Signal Descriptions	1015
Additional FPGA Specific Signals	1017
Additional FPGA I/O Signal Descriptions.....	1017

Chapter 3: PowerPC® 405 OCM Controller

Introduction	1019
Functional Features	1019
Common Features	1019
Data-Side OCM (DSOCM)	1020
Instruction-Side OCM (ISOCM)	1020
OCM Controller Operation	1020
Operational Summary	1020
DSOCM Ports.....	1021
DSOCM Attributes	1022
ISOCM Ports	1023
ISOCM Attributes.....	1025
Timing Specification	1026
Single-Cycle Mode	1026
Multi-Cycle Mode	1027
Programmer's Model	1028
DCR Registers	1028
References	1030
Application Notes	1030
Interfacing to Block RAM.....	1030
Size vs. Performance	1030
Application Example	1031

Appendix A: RISCWatch and RISCTrace Interfaces

RISCWatch Interface	1039
RISCTrace Interface	1041

Signal Summary	1043

Index	1049

Volume 3: Rocket I/O™ Transceiver User Guide

Chapter 1: Introduction

Rocket I/O Features	1055
In This User Guide	1055
Naming Conventions.....	1056
For More Information.....	1056
Further Reading	1056
Documentation Provided by Xilinx	1056
IBM® CoreConnect™ Documentation	1057
Software Development Documentation.....	1057

Chapter 2: Rocket I/O™ Transceiver Overview

Basic Architecture and Capabilities.....	1059
Clock Synthesizer	1061
Clock and Data Recovery.....	1062
Transmitter.....	1062
FPGA Transmit Interface.....	1062
8B/10B Encoder	1062
Disparity Control.....	1062
Transmit FIFO	1063
Serializer	1063
Receiver.....	1063
Deserializer.....	1063
Receiver Termination.....	1064
8B/10B Decoder.....	1064
Loopback.....	1064
Elastic and Transmitter Buffers	1065
Receiver Buffer.....	1065
Clock Correction.....	1065
Channel Bonding	1066
Transmitter Buffer	1067
CRC.....	1067
Reset/Power Down	1067

Chapter 3: Digital Design Considerations

List of Available Ports.....	1069
Primitive Attributes	1073
Modifiable Primitives	1078
Byte Mapping.....	1082
Clocking.....	1082
Clock Signals	1082
Clock Ratio	1083
Digital Clock Manager (DCM) Examples	1083
Multiplexed Clocking Scheme.....	1093
Clock Dependency	1094
Resets	1097

Rocket I/O Transceiver Instantiations	1098
PLL Operation and Clock Recovery	1098
Clock Correction Count	1099
RX_LOSS_OF_SYNC_FSM	1099
8B/10B Operation	1100
Vitesse Disparity Example	1102
Status Signals	1103
8B/10B Encoding	1103
8B/10B Serial Output Format	1112
CRC Operation	1113
CRC Generation	1113
CRC Latency	1114
CRC Limitations	1114
CRC Modes	1114
Channel Bonding (Channel-to-Channel Alignment)	1116

Chapter 4: Analog Design Considerations

Serial I/O Description	1119
Pre-emphasis Techniques	1120
Differential Receiver	1123
Jitter	1123
Total Jitter (DJ + RJ)	1123
Clock and Data Recovery	1123
PCB Design Requirements	1125
Power Filtering	1125
High-Speed Serial Trace Design	1127
Power Consumption	1130
Reference Clock	1130

Chapter 5: Simulation and Implementation

Simulation Models	1131
Smart Model	1131
HSPICE	1131
Behavioral	1131
Implementation Tools	1131
Synthesis	1131
Par	1131
UCF Example	1132
Implementing Clock Schemes	1132
Diagnostic Signals	1133
LOOPBACK	1133

Appendix A: Rocket I/O™ Cell Models

Summary	1135
Verilog Module Declarations	1135
GT_AURORA_1	1135
GT_AURORA_2	1136
GT_AURORA_4	1137
GT_CUSTOM	1138
GT_ETHERNET_1	1139

GT_ETHERNET_2.....	1139
GT_ETHERNET_4.....	1140
GT_FIBRE_CHAN_1.....	1141
GT_FIBRE_CHAN_2.....	1142
GT_FIBRE_CHAN_4.....	1143
GT_INFINIBAND_1	1144
GT_INFINIBAND_2	1144
GT_INFINIBAND_4	1145
GT_XAUI_1	1146
GT_XAUI_2	1147
GT_XAUI_4	1148

Volume 4: Design Flow

Chapter 1: Introduction to Virtex-II Pro™ Design Flow

Introduction	1153
Design Flow Input Files	1155
Software Source Files	1155
Configuration Files.....	1155
Hardware Design Source Files	1155
Constraint Files	1155
Simulation Models	1155
Design Flow Processes	1155
Build Software	1155
Run Data2BRAM	1156
Simulate	1156
Synthesize	1156
Implement FPGA.....	1156
Generate Timing Data.....	1156
Generate Bitstream	1156
Further Reading	1157
Documentation Provided by Xilinx	1157
IBM® CoreConnect™ Documentation	1157
Software Development Documentation.....	1158

Chapter 2: Design Flow Overview

Introduction	1159
Related Documentation	1159
Software	1159
make	1160
GCC	1161
Binutils	1162
GDB	1163
Cross-Compilation and the powerpc-eabi Target.....	1163
Hardware	1164
Targets	1165
sw	1166
func_sim.....	1167
synth	1170
fpga.....	1172

ba_sim	1174
bit	1175
File Syntax	1176
flow.cfg.....	1176
LST File (src.lst).....	1179
Tool-Specific Files.....	1181
Special Notes	1181
HDL = verilog	1181
Example Files	1182
flow.cfg.....	1182
src.lst.....	1184
Requirements	1185
Directory Structure	1186

Chapter 3: PPC405 Bus Functional Model Usage

Overview	1189
Description.....	1189
Bus Restrictions.....	1190
BFL	1190

Chapter 4: SmartModel® Usage

Overview	1193
Description.....	1193
Available SmartModels.....	1193
Using Models in a Design	1194
Verilog LMTV PLI Tasks	1194
PPC405 Model.....	1194
Rocket I/O Transceiver Model.....	1197
Invoking the Simulator	1197
Include Wrapper Files	1197
Include PLI for Verilog-XL Only	1197
Further Reading	1197

Volume 5: Software Development Tools

Chapter 1: Introduction to Software Development Tools

Chapter 2: Using Data2BRAM

Abstract	1203
Introduction	1203
Feature Summary	1204
Simulation Features	1204
Software Debug Features	1204
FPGA Implementation Features.....	1204
A Primer on How Data2BRAM Works.....	1205
General Concepts	1205

Simulating Designs With Data2BRAM	1211
Software Debugging Using Data2BRAM	1213
FPGA Implementation Tools Interaction.....	1213
Tutorial On Running the Stand-Alone Data2BRAM	1213
Technical Reference for Data2BRAM.....	1213
Data2BRAM Command Line Reference	1213
How Data2BRAM Handles Bitwise Twisting of Data	1215
Tool Behavior in Xilinx ISE 4.1i	1218
Known Limitations.....	1220
Future Enhancements	1220
Modified Backus-Naur Form for .bmm File Syntax	1221
Appendix A - Example Data2BRAM .bmm File.....	1222
Appendix B - Example Data2BRAM .v File Output	1225
Appendix C - Example Data2BRAM .mem File Output	1228
Appendix D - Example Snippet of Data2BRAM .ucf File.....	1229

Chapter 3: GNU Software Tool Chain

Overview	1233
Related Documentation.....	1233
Built-in Help Pages	1233
Man Pages.....	1233
Info Pages	1234
Internet Pages.....	1234
Books About Programming in C	1234
Online Documents About Programming in C	1235
Terminology	1235
Tool Flow	1236
Make	1236
GCC	1237
Binutils	1238
AS.....	1238
AR	1238
LD	1238
OBJDUMP.....	1239
GDB	1239
Cross-Compilation and the powerpc-eabi Target	1239
Building GNU Software.....	1240

Volume 6: Reference Systems

Chapter 1: Introduction to Reference Systems

Summary	1245
Introduction	1245
Reference System Information.....	1245
Simple PPC405 Reference System.....	1246
PPC405 PPE Reference System Using Rocket I/O Transceivers	1246
Embedded PPC405 Reference System.....	1246
Further Reading	1247

Documentation Provided by Xilinx	1247
IBM® CoreConnect™ Documentation	1247
Software Development Documentation.....	1248

Chapter 2: Simple PPC405 Reference System

Introduction	1251
Hardware	1251
Overview	1251
Processor Local Bus (PLB).....	1252
Device Control Registers (DCR).....	1252
Other Devices.....	1252
Clock/Reset Distribution	1252
HDL Organization	1254
Simulation and Verification	1255
SWIFT and BFM CPU Models	1255
Behavioral Models.....	1256
Synthesis and Implementation	1256
Design Flow Environment	1256
File Listings	1257
Instructions for Running Functional Simulations	1259
SWIFT Simulations.....	1259
BFM Simulations	1260
Instructions for Synthesizing the Design	1261
Instructions for FPGA Implementation	1261
Instructions for Back-Annotated Timing Simulation	1262
SWIFT Simulations.....	1262
BFM Simulations	1262
Memory Map	1263
Extending or Modifying the Design	1263
Adding or Removing PLB Devices.....	1263
Adding or Removing DCR Devices.....	1264
Behavioral Models/Testbenches.....	1264
Modifying Software	1264
File Structure for Simple PPC405 Reference System	1266

Chapter 3: PPC405 PPE Reference System Using Rocket I/O™ Transceivers

Introduction	1267
Hardware	1268
Overview	1268
Processor Block Interfaces.....	1268
Device Control Registers (DCR).....	1269
Rocket I/O Transceiver Interface.....	1270
Rocket I/O Attributes	1270
Clock/Reset Distribution	1271
Theory of Operation	1272
Frame Format.....	1272
Example Applications	1274
HDL Organization	1279
Simulation and Verification	1281

SWIFT and BFM CPU Models	1281
Behavioral Models	1281
Synthesis and Implementation	1281
Design Flow Environment	1282
File Listings	1282
Instructions for Running Functional Simulations	1284
SWIFT Simulations	1285
BFM Simulations	1285
Instructions for Synthesizing the Design	1286
Instructions for FPGA Implementation	1287
Instructions for Back-Annotated Timing Simulation	1287
SWIFT Simulations	1287
BFM Simulations	1288
Memory Map	1288
Extending or Modifying the Design	1288
File Structure for PPE Reference System	1289

Chapter 4: Embedded PPC405 Reference System

Introduction	1291
Hardware	1291
Overview	1291
Processor Local Bus (PLB)	1292
On-Chip Peripheral Bus (OPB)	1293
Device Control Registers (DCR)	1294
Other Devices	1295
Interrupts	1295
Clock/Reset Distribution	1295
HDL Organization	1298
Simulation and Verification	1300
SWIFT and BFM CPU Models	1300
Behavioral Models	1300
Synthesis and Implementation	1301
Design Flow Environment	1301
File Listings	1301
Instructions for Running Functional Simulations	1305
SWIFT Simulations	1305
BFM Simulations	1306
Instructions for Synthesizing the Design	1307
Instructions for FPGA Implementation	1307
Instructions for Back-Annotated Timing Simulation	1308
SWIFT Simulations	1308
BFM Simulations	1309
Memory Map	1309
Extending or Modifying the Design	1311
Adding or Removing PLB Devices	1311
Adding or Removing OPB Devices	1311
Adding or Removing DCR Devices	1312
Other Modifications	1312
Behavioral Models/Testbenches	1312
Modifying Software	1313

Volume 7: Hardware IP Specifications

Chapter 1: Introduction to Hardware IP

Summary	1319
Introduction	1319
Hardware IP Sizes.....	1319
Further Reading	1321
Documentation Provided by Xilinx	1321
IBM® CoreConnect™ Documentation	1321
Software Development Documentation.....	1322

Chapter 2: Using IPIF to Build IP

Abstract	1323
Introduction	1323
SRAM Protocol Overview of IPIF	1324
Basic Write Transactions	1325
Basic Read Transactions	1326
IPIF Status and Control Signals.....	1326
Using IPIF to Create a GPIO Peripheral from Scratch	1327
Using IPIF to Connect a Pre-existent Peripheral to the Bus.....	1328
Conclusion.....	1329

Chapter 3: IPIF Specification

Overview	1331
IPIF Master Module Overview.....	1332
IPIF Slave Modules Overview	1332
Signal Conventions	1333
Bus Numbering and Bit Ordering.....	1334
Parameter Indexing Versus Parameter Numbering.....	1334
IPIF Modules in an Example OPB System.....	1334
Design Considerations.....	1336
DMA Engine	1336
Interrupts.....	1337
Bus Arbiter and Bridges	1337
Data Bus Width.....	1338
Retry, Error, and Timeout Suppress.....	1338
IPIF Module Specifications	1338
Slave DMA Handshake Module	1338
Slave Control Register Module	1343
Slave SRAM Module.....	1351
Slave FIFO Module.....	1360
Master Module.....	1368
IPIF Parameterization	1378
IPIF Signals	1381

Chapter 4: OPB Arbiter

Summary	1385
Introduction	1385
OPB Arbiter Overview	1385
Features	1385
OPB Arbitration Protocol	1385
OPB Arbiter I/O Signals	1387
OPB Arbiter Register Descriptions	1388
OPB Arbiter Priority Register	1389
OPB Arbiter Control Register	1390
OPB Arbiter Block Diagram	1391
OPB Slave Interface (IPIF)	1392
Priority Register Logic	1392
Control Register Logic	1394
IP2OPB Data Mux	1394
Arbitration Logic	1395
Park/Lock Logic	1396
Watchdog Timer	1398

Chapter 5: OPB Configuration ROM Module

Overview	1401
Related Documents	1401
Features	1401
Module Port Interface	1401
Implementation	1402
Design Extensions and Improvements	1403

Chapter 6: OPB GPIO Module

Overview	1405
Related Documents	1405
Features	1405
Module Port Interface	1405
Implementation	1406
Memory Map	1409

Chapter 7: OPB LCD Controller

Overview	1411
Related Documents	1411
Features	1411
Module Port Interfaces	1411
Implementation	1412
Design Overview	1412
Read/Write Timing	1413
Memory Map	1415

Chapter 8: OPB 16450 UART Controller

Overview	1417

Related Documents	1417
Features	1417
UART Background	1418
UART I/O Signals	1418
UART Register Definition	1420
UART Interface (IPIF)	1420
UART Register Logic	1420
UART Block Diagram	1426
Specification Exceptions	1426
System Clock	1426
Register Addresses	1426

Chapter 9: OPB 16550 UART Controller

Overview	1427
Related Documents	1427
Features	1427
UART Background	1428
UART I/O Signals	1428
UART Register Definition	1430
UART Interface (IPIF)	1430
UART Register Logic	1430
UART Block Diagram	1437
Specification Exceptions	1437
FIFO Control Register	1437
System Clock	1437
Register Addresses	1437

Chapter 10: OPB IIC Controller

Important Note Regarding Philips I²C-Bus Technology	1439
Overview	1439
Related Documents	1439
Features	1439
IIC Background	1440
IIC I/O Signals	1441
IIC Register Descriptions	1442
IIC Control Register	1442
IIC Interrupt Descriptions	1446
Interrupts	1446
Flow Descriptions	1447
Flows	1447
Throttle Description	1448
IIC Block Diagram	1449
Specification Exceptions	1449
Exceptions to the Philips I2C-Bus Specification, Version 2.1, January 2000	1449

Chapter 11: OPB to DCR Bridge

Overview	1451
Related Documents	1451

Features	1451
Module Port Interfaces	1451
Implementation	1453

Chapter 12: OPB to PCI Bridge

Overview	1455
Related Documents	1455
Features	1455
Module Port Interface	1455
Implementation	1457
Memory Map	1459
Xilinx LogiCORE PCI 64/66	1459

Chapter 13: PLB Arbiter

Overview	1461
Bus Control Unit	1462
Address Path Unit	1463
Read Data Path Unit	1463
Write Data Path Unit	1463
Watchdog Timer	1463
Clock and Power Management	1463
Reset Considerations	1463
Supported Configurability	1463
Programmable Arbitration	1463
PLB Arbiter Signals	1464
PLB Arbiter Interfaces	1466
Master[n] Interface	1467
Slave Interface	1476
DCR Interface	1490
Clock Interface	1492
PLB Arbiter Registers	1493
PLB Arbiter Control Register (PACR)	1494
PLB Error Address Register (PEAR)	1494
PLB Error Status Register (PESR)	1494
PLB Arbiter Operations	1498
Single Read Transfer Bus Time-Out	1499
Single Write Transfer Bus Time-Out	1500
Line Read Transfer Bus Time-Out	1501
Line Write Transfer Bus Time-Out	1502
Burst Read Transfer Bus Time-Out	1503
Burst Write Transfer Bus Time-Out	1504
Read Transfer	1505
Pipelined Read Transfer	1506
Pipelined Write Transfer	1507

Chapter 14: PLB BRAM Controller

Overview	1509
Related Documents	1509
Features	1509
Module Port Interface	1510

PLB Slave Interface Signals	1510
BRAM Interface Signals.....	1511
Parameters	1512
High-Level Description	1512
Implementation	1512
Design Extensions and Improvements	1513
User Port Configuration	1513
Number of BRAMs.....	1513
Code / Data Space Initialization	1513

Chapter 15: PLB UART Controller

Overview	1515
Related Documents	1515
Features	1515
Module Port Interface	1515
PLB Slave Interface Signals	1515
UART Interface Signals	1517
Implementation	1518
PLB IPIF	1519
Differences From 16550 UART Discrete Chips	1520

Chapter 16: PLB SRAM/Flash Controller

Overview	1521
Related Documents	1522
Features	1522
Module Port Interface	1522
Address Mapping.....	1524
Controller State Machine.....	1525
SRAM Timing Diagrams	1526
Flash Timing Diagrams	1528
Conclusion	1529
Further Reading	1529
Documentation Provided by Xilinx	1529
IBM® CoreConnect™ Documentation	1530
Software Development Documentation.....	1530

Chapter 17: PLB ZBT SRAM Controller

Overview	1533
Related Documents	1533
Features	1533
Module Port Interface	1534
PLB Interface Signals	1534
ZBT SRAM Interface Signals	1535
Read and Write Transactions.....	1536
Read Transactions	1537
Write Transactions	1538
Address Mapping.....	1538
Controller State Machine	1540
Conclusion	1541

Chapter 18: PLB DDR SDRAM Controller

Overview	1543
Related Documents	1543
Features	1543
Module Port Interface	1544
DDR Controller	1546
Initialization and Refresh	1547
Address Mux	1547
Data Path	1548
Controller State Machine	1549
Timing Diagrams	1551
Conclusion	1552

Chapter 19: PLB to OPB Bridge-Out Module

Overview	1553
Related Documents	1553
Features	1553
Module Port Interface	1554
PLB Slave Interface Signals	1554
OPB Master Interface Signals	1555
Bus Error Address/Status Registers	1556
Parameters	1556
Implementation	1557
High-Level Description	1557
PLB Interface	1558
PLB-to-OPB Transfer Interface	1559
OPB Interface	1559
OPB-to-PLB Receive Data Interface	1560
BEAR/BESR Logic	1560
PLB and OPB Transaction Behavior	1560
PLB Transaction Qualifiers and Arbitration Signals	1560
PLB Data Transfers	1561
OPB Data Transfers	1561
Bus Error Address and Status Registers	1562

Chapter 20: Interrupt Controller

Overview	1565
Features	1565
Introduction to Interrupt Controllers	1565
Intel 8051	1565
Zilog Z80	1566
Motorola 68332	1566
MIPS	1566
ARM	1566
IBM PowerPC 405GP Universal Interrupt Controller (UIC)	1567
Interrupt Controller Organization	1567
INTC Detailed Register Description	1570
Parameters	1578
C_INTC_DCR_DECODE_ADDR (Default Value: 10'h3f8)	1578

C_INTC_EDGE_SENSE (Default Value: 32'b1111_1111_1111_1111_1111_1111_1111_1111)	1579
C_INTC_NUM_OF_INTS (Default Value: 32)	1579
C_DCR_ABUS_NUM_BITS (Default Value: 10)	1579
C_DCR_DBUS_NUM_BITS (Default Value: 32)	1580
Expanding the Number of Available Interrupts (Daisy-Chaining Controllers)	1580
General Interrupt Information and Helpful Hints	1581
Interrupt Servicing vs. Interrupt Processing	1581
Interrupt Acknowledgment	1581
Verilog Module Port Declarations	1582

Chapter 21: Packet Processing Engine Controller

Overview	1583
Related Documents	1583
Features	1583
Module Port Interface	1584
Control/Status Signals	1584
Rocket I/O Transceiver Interface Signals	1585
BRAM Signals	1586
Implementation	1586
Transmit Function	1587
Receive Functions	1588
Frame Format	1589
Data Acquisition/Storage	1590
Device Control Register (DCR) Bus Interface	1590
Processor DCR Interface	1592

Volume 8: Software IP and Applications

Chapter 1: Introduction to Software IP and Applications

Chapter 2: Stand-Alone Board Support Package

Overview	1597
Building the BSP	1597
Boot Code	1598
boot.S	1598
crt0.S	1598
eabi.S	1598
Bus Error Registers	1599
unsigned int bexr_BaseAdr;	1599
unsigned int bexr_Read(unsigned int adr);	1599
void bexr_Write(unsigned int adr, unsigned int val);	1599
Cache	1600
void writeCCR0(unsigned int val);	1600
void enable_dcache(unsigned int regions);	1600
void disable_dcache(void);	1600
void flush_dcache_line(unsigned int adr);	1600

void store_dcache_line(unsigned int adr);	1600
void enable_icache(unsigned int regions);	1600
void disable_icache(void);.....	1600
void invalidate_icache(void);	1600
void invalidate_icache_line(unsigned int adr);.....	1601
Configuration	1602
unsigned int Processor_Frequency;.....	1602
unsigned int PLB_Frequency;.....	1602
unsigned int OPB_Frequency;.....	1602
Exception Handling	1603
void exception_Init(void);.....	1603
void exception_RegisterHandler(exception_intVector Exception_number, void * exception_handler, void * user_value);.....	1603
void exception_DeregisterHandler(exception_intVectors exception_number)	1604
void exception_EnableInterrupts (exception_intTypes val);	1604
void exception_DisableInterrupts (exception_intTypes val);	1604
Files	1605
int read(int fd, char *buf, int nbytes);	1605
int write(int fd, char *buf, int nbytes);.....	1605
int select(int fd, int *readmask, int *writemask, int *excpmask, int timeout);.....	1605
int isatty(int fd);	1605
General Purpose I/O	1606
unsigned int gpio_BaseAdr;	1606
unsigned int gpio_Read(unsigned int adr);.....	1606
void gpio_Write(unsigned int adr, unsigned int val);	1606
IIC Support	1607
unsigned int IICBaseAdr;.....	1607
void iic_Init(unsigned int baseAdr);.....	1607
unsigned int iic_Read(unsigned int baseAdr, unsigned char deviceAdr, unsigned char *buf, unsigned int len);.....	1607
unsigned int iic_Write(unsigned int baseAdr, unsigned int deviceAdr, unsigned char *buf, unsigned int len);.....	1607
IIC AT24CXX	1608
unsigned int iicat24cxx_Read(unsigned int baseAdr, unsigned char deviceAdr, unsigned int memAdr, unsigned char *buf, unsigned int len);.....	1608
unsigned int iicat24cxx_Write(unsigned int baseAdr, unsigned char deviceAdr, unsigned int memAdr, unsigned char *buf, unsigned int len);.....	1608
LCD	1609
unsigned int lcd_BaseAdr;.....	1609
void lcd_Open(void);.....	1609
void lcd_Write(unsigned char val);	1609
void lcd_ClearScreen(void);.....	1609
void lcd_SetPos(unsigned char val);.....	1609
Memory and I/O Access	1610
unsigned char io_rb(unsigned int adr);.....	1610
unsigned char io_read_byte(unsigned int adr);.....	1610
unsigned char m_rb(unsigned int adr);	1610
unsigned char mem_read_byte(unsigned int adr);.....	1610
unsigned short io_rh(unsigned int adr);.....	1610
unsigned short io_read_halfword(unsigned int adr);.....	1610
unsigned short m_rh(unsigned int adr);.....	1610
unsigned short mem_read_halfword(unsigned int adr);	1610
unsigned int io_rw(unsigned int adr);	1610
unsigned int io_read_word(unsigned int adr);.....	1610
unsigned int m_rw(unsigned int adr);	1610

unsigned int mem_read_word(unsigned int adr);	1610
void io_wb(unsigned int adr, unsigned char val);	1610
void io_write_byte(unsigned int adr, unsigned char val);	1610
void m_wb(unsigned int adr, unsigned char val);	1610
void mem_write_byte(unsigned int adr, unsigned char val);	1610
void io_wh(unsigned int adr, unsigned short val);	1610
void io_write_halfword(unsigned int adr, unsigned short val);	1610
void m_wh(unsigned int adr, unsigned short val);	1610
void mem_write_halfword(unsigned int adr, unsigned short val);	1610
void io_ww(unsigned int adr, unsigned int val);	1610
void io_write_word(unsigned int adr, unsigned int val);	1610
void m_ww(unsigned int adr, unsigned int val);	1610
void mem_write_word(unsigned int adr, unsigned int val);	1610
unsigned short io_rhr(unsigned int adr);	1611
unsigned short io_read_halfword_reversed(unsigned int adr);	1611
unsigned short m_rhr(unsigned int adr);	1611
unsigned short mem_read_halfword_reversed(unsigned int adr);	1611
unsigned int io_rwr(unsigned int adr);	1611
unsigned int io_read_word_reversed(unsigned int adr);	1611
unsigned int m_rwr(unsigned int adr);	1611
unsigned int mem_read_word_reversed(unsigned int adr);	1611
void io_whr(unsigned int adr, unsigned short val);	1611
void io_write_halfword_reversed(unsigned int adr, unsigned short val);	1611
void m_whr(unsigned int adr, unsigned short val);	1611
void mem_write_halfword_reversed(unsigned int adr, unsigned short val);	1611
void io_wwr(unsigned int adr, unsigned int val);	1611
void io_write_word_reversed(unsigned int adr, unsigned int val);	1611
void m_wwr(unsigned int adr, unsigned int val);	1611
void mem_write_word_reversed(unsigned int adr, unsigned int val);	1611
Memory Management	1612
char *sbrk(int nbytes);	1612
PCI	1613
void pci_init(void);	1613
int pci_find_device(int busDevFunc, unsigned short vendorid, unsigned short deviceid);	1613
int pci_find_device_type(int busDevFunc, int classCode);	1613
int pci_scan(int busDevFunc);	1613
int pci_map(int busDevFunc, int bar, unsigned int *physAdr, unsigned int *size);	1614
unsigned int pci_read_config_reg(int busDevFunc, int reg, int width);	1614
void pci_write_config_reg(int busDevFunc, int reg, unsigned int val, int width);	1614
void pci_out32(unsigned int adr, unsigned int val);	1614
void pci_out16(unsigned int adr, unsigned short val);	1614
void pci_out8(unsigned int adr, unsigned char val);	1614
unsigned int pci_in32(unsigned int adr);	1615
unsigned short pci_in16(unsigned int adr);	1615
unsigned char pci_in8(unsigned int adr);	1615
Process	1616
Processor-Specific Include Files	1617
StrataFlash	1618
unsigned int strataflash_BaseAdr;	1618
unsigned int strataflash_Size;	1618
int strataflash_AddressToBlock(unsigned int adr, unsigned int *block);	1618
int strataflash_BlockToAddress(unsigned int block, unsigned int *adr);	1618
int strataflash_Write(unsigned int adr, unsigned char *buf, unsigned int len);	1618

int strataflash_EraseBlock(unsigned int block);.....	1618
int strataflash_LockBlock(unsigned int block);.....	1618
int strataflash_UnlockBlock(unsigned int block);.....	1618
Time	1619
typedef struct MTime;	1619
int mtime_compare(MTime c0, MTime c1);.....	1619
void mtime_settime(MTime mtime);.....	1619
void mtime_gettime(MTime *mtime);.....	1619
void mtime_add(MTime a0, MTime a1, MTime *r);	1619
void mtime_subtract(MTime s0, MTime s1, MTime *r);.....	1619
void mtime_TSRClearStatusBits(unsigned long Bitmask);.....	1619
void mtime_PITSetInterval(unsigned long interval);.....	1620
void mtime_PITEnableInterrupt(void);.....	1620
void mtime_PITDisableInterrupt(void);.....	1620
void mtime_PITEnableAutoReload(void);.....	1620
void mtime_PITDisableAutoReload(void);.....	1621
void mtime_PITClearInterrupt(void);.....	1621
unsigned int usleep(unsigned int __useconds);.....	1621
unsigned int sleep(unsigned int __seconds);	1621
int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);	1621
UART	1622
unsigned int Uart_Frequency;.....	1622
unsigned int Uart1BaseAdr;	1622
unsigned int Uart2BaseAdr;	1622
void UART_Init(unsigned int baseAdr, int baud, char data, char parity, char stop);	1622
unsigned char UART_Read(unsigned int baseAdr);	1622
void UART_Write(unsigned int baseAdr, unsigned char c);.....	1622
void UART_Select(unsigned int baseAdr, int *readmask, int *writemask, int *excpmask);.....	1623

Chapter 3: PPC405 Software: Demonstration and Test Applications

Overview	1625
Related Documentation	1627
flasher	1628
Description	1628
S-Record Format.....	1628
Usage.....	1628
Files.....	1628
Building for V2PDK	1629
hello	1630
Description	1630
Usage.....	1630
Files.....	1630
Building for V2PDK	1630
hello_gpio	1632
Description	1632
Usage.....	1632
Files.....	1632
Building for V2PDK	1632
hello_iic_at24cxx	1634
Description	1634
Usage.....	1634

Files.....	1634
Building for V2PDK	1634
hello_lcd	1636
Description	1636
Usage.....	1636
Files.....	1636
Building for V2PDK	1636
hello_pit	1638
Description	1638
Usage.....	1638
Files.....	1638
Building for V2PDK	1638
hello_uart	1640
Description	1640
Usage.....	1640
Files.....	1640
Building for V2PDK	1640
scan_pci	1642
Description	1642
Usage.....	1642
Files.....	1642
Building for V2PDK	1643
xrom	1644
Description	1644
Usage.....	1644
Menu Structure	1644
Test Characterization	1645
Files.....	1652
Building for V2PDK	1653

Chapter 4: Simulation Terminal

Overview	1655
Syntax	1655
File Format	1655
Terminology	1655
Input File.....	1655
Output File	1656
Remarks	1656

Schedule of Figures

Introduction to Documentation

Volume 1:

Advance Product Specification

Data Sheet Module 1:

Virtex-II Pro™ Platform FPGAs: Introduction and Overview

Figure 1: Virtex-II Pro Ordering Information	81

Data Sheet Module 2:

Virtex-II Pro™ Platform FPGAs: Functional Description

Figure 1: Virtex-II Pro Generic Architecture Overview	83
Figure 2: Rocket I/O Block Diagram	85
Figure 3: Clock Correction in Receiver	88
Figure 4: Channel Bonding (Alignment)	88
Figure 5: Processor Block Architecture	90
Figure 6: CoreConnect Block Diagram	92
Figure 7: PPC405 Core Block Diagram	93
Figure 8: Relationship of Timer Facilities to Base Clock	95
Figure 9: Virtex-II Pro Input/Output Tile	96
Figure 10: Virtex-II Pro IOB Block	97
Figure 11: Double Data Rate Registers	98
Figure 12: Register / Latch Configuration in an IOB Block	99
Figure 13: LVCMOS SelectI/O Standard	99
Figure 14: SSTL or HSTL SelectI/O Standards	100
Figure 15: Virtex-II Pro I/O Banks: Top View for Wire-Bond Packages (CS, FG, and BG)	101
Figure 16: Virtex-II Pro I/O Banks: Top View for Flip-Chip Packages (FF and BF)	101
Figure 17: DCI in a Virtex-II Pro Bank	102
Figure 18: Internal Series Termination	102
Figure 19: DCI Usage Examples	103
Figure 20: Virtex-II Pro CLB Element	104
Figure 21: Virtex-II Pro Slice Configuration	104
Figure 22: Virtex-II Pro Slice (Top Half)	105
Figure 23: Register / Latch Configuration in a Slice	106
Figure 24: Distributed SelectRAM (RAM16x1S)	107
Figure 25: Single-Port Distributed SelectRAM (RAM32x1S)	107
Figure 26: Dual-Port Distributed SelectRAM (RAM16x1D)	107

Figure 27: Shift Register Configurations	108
Figure 28: Cascadable Shift Register	108
Figure 29: MUXF5 and MUXFX multiplexers	109
Figure 30: Fast Carry Logic Path	110
Figure 31: Horizontal Cascade Chain	111
Figure 32: Wide-Input AND Gate (16 Inputs).....	111
Figure 33: Virtex-II Pro 3-State Buffers	112
Figure 34: 3-State Buffer Connection to Horizontal Lines	112
Figure 35: 18 Kb Block SelectRAM Memory in Single-Port Mode	113
Figure 36: 18 Kb Block SelectRAM in Dual-Port Mode.....	114
Figure 37: WRITE_FIRST Mode	115
Figure 38: READ_FIRST Mode.....	115
Figure 39: NO_CHANGE Mode	115
Figure 40: XC2VP4 Block RAM Column Layout.....	116
Figure 41: SelectRAM and Multiplier Blocks	116
Figure 42: Multiplier Block	116
Figure 43: Virtex-II Pro Clock Pads.....	117
Figure 44: Virtex-II Pro Clock Multiplexer Buffer Configuration.....	117
Figure 45: Virtex-II Pro Clock Distribution	118
Figure 46: Virtex-II Pro BUFG Function	118
Figure 47: Virtex-II Pro BUFGCE Function	118
Figure 48: Virtex-II Pro BUFGMUX Function.....	119
Figure 49: Clock Multiplexer Waveform Diagram	119
Figure 50: Digital Clock Manager	119
Figure 51: Fine-Phase Shifting Effects	121
Figure 52: Hierarchical Routing Resources	123

Data Sheet Module 3:

Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics

Figure 1: Reference Clock (REFCLK) Timing Parameters.....	139
Figure 2: Receive Latency (Maximum)	140
Figure 3: Transmit Latency (Maximum, Including CRC)	141

Data Sheet Module 4:

Virtex-II Pro™ Platform FPGAs: Pinout Information

Figure 1: FG256 Fine-Pitch BGA Package Specifications	179
Figure 2: FG456 Fine-Pitch BGA Package Specifications	194
Figure 3: FF672 Flip-Chip Fine-Pitch BGA Package Specifications	215
Figure 4: FF896 Flip-Chip Fine-Pitch BGA Package Specifications	241
Figure 5: FF1152 Flip-Chip Fine-Pitch BGA Package Specifications	275
Figure 6: FF1517 Flip-Chip Fine-Pitch BGA Package Specifications	314
Figure 7: BF957 Flip-Chip BGA Package Specifications.....	341

Volume 2: PowerPC® 405 Processor

Volume 2(a): PPC405 User Manual

About This Book

Chapter 1: Introduction to the PPC405

<i>Figure 1-1: Relationship of PowerPC Architectures</i>	362
<i>Figure 1-2: PPC405 Registers</i>	369
<i>Figure 1-3: PPC405 Organization</i>	371

Chapter 2: Operational Concepts

<i>Figure 2-1: PowerPC 32-Bit Memory Management</i>	382
<i>Figure 2-2: Operand Data Types</i>	384

Chapter 3: User Programming Model

<i>Figure 3-1: PPC405 User Registers</i>	396
<i>Figure 3-2: General Purpose Registers (R0-R31)</i>	396
<i>Figure 3-3: Condition Register (CR)</i>	397
<i>Figure 3-4: CR_n Field</i>	397
<i>Figure 3-5: Fixed Point Exception Register (XER)</i>	399
<i>Figure 3-6: Link Register (LR)</i>	400
<i>Figure 3-7: Count Register (CTR)</i>	400
<i>Figure 3-8: User SPR General-Purpose Register (USPRG0)</i>	401
<i>Figure 3-9: SPR General-Purpose Registers (SPRG4-SPRG7)</i>	401
<i>Figure 3-10: Time-Base Register</i>	401
<i>Figure 3-11: Branch-to-Relative Addressing</i>	409
<i>Figure 3-12: Branch-Conditional to Relative Addressing</i>	409
<i>Figure 3-13: Branch-to-Absolute Addressing</i>	410
<i>Figure 3-14: Branch-Conditional to Absolute Addressing</i>	410
<i>Figure 3-15: Branch-Conditional to Link-Register Addressing</i>	411
<i>Figure 3-16: Branch-Conditional to Count-Register Addressing</i>	411
<i>Figure 3-17: Register-Indirect with Immediate-Index Addressing</i>	415
<i>Figure 3-18: Register-Indirect with Index Addressing</i>	416
<i>Figure 3-19: Register-Indirect Addressing</i>	416
<i>Figure 3-20: Load and Store with Byte-Reverse Instructions</i>	422
<i>Figure 3-21: Load and Store Multiple Instructions</i>	423
<i>Figure 3-22: Load and Store String Instructions</i>	425
<i>Figure 3-23: Rotate Mask Generation</i>	436
<i>Figure 3-24: Rotate Left then AND-with-Mask Immediate Example</i>	437
<i>Figure 3-25: Rotate Left then Mask-Insert Immediate Example</i>	438
<i>Figure 3-26: Logical-Shift Examples</i>	440

<i>Figure 3-27: Algebraic-Shift Example</i>	441
<i>Figure 3-28: Multiply-Accumulate Cross-Halfword to Word Operation</i>	444
<i>Figure 3-29: Multiply-Accumulate High-Halfword to Word Operation</i>	446
<i>Figure 3-30: Multiply-Accumulate Low-Halfword to Word Operation</i>	449
<i>Figure 3-31: Negative Multiply-Accumulate Cross-Halfword to Word Operation</i>	451
<i>Figure 3-32: Negative Multiply-Accumulate High-Halfword to Word Operation</i>	453
<i>Figure 3-33: Negative Multiply-Accumulate Low-Halfword to Word Operation</i>	455
<i>Figure 3-34: Multiply Cross-Halfword to Word Operation</i>	456
<i>Figure 3-35: Multiply High-Halfword to Word Operation</i>	457
<i>Figure 3-36: Multiply Low-Halfword to Word Operation</i>	458
<i>Figure 3-37: mtrcf Field Mask (CRM) Format</i>	459
<i>Figure 3-38: mtrcf Example</i>	460

Chapter 4: PPC405 Privileged-Mode Programming Model

<i>Figure 4-1: PPC405 Privileged Registers</i>	466
<i>Figure 4-2: Machine-State Register (MSR)</i>	467
<i>Figure 4-3: SPR General-Purpose Registers (SPRG0–SPRG7)</i>	469
<i>Figure 4-4: Processor-Version Register (PVR)</i>	469

Chapter 5: Memory-System Management

<i>Figure 5-1: PPC405 Memory-System Organization</i>	473
<i>Figure 5-2: Logical Structure of the PPC405 Cache Arrays</i>	475
<i>Figure 5-3: Address Fields Used to Access Caches</i>	476
<i>Figure 5-4: Data-Cache Access Example</i>	476
<i>Figure 5-5: Instruction Flow from the Instruction-Cache Unit</i>	477
<i>Figure 5-6: Data Flow to/from the Data-Cache Unit</i>	479
<i>Figure 5-7: Data-Cache Write-Through Register (DCWR)</i>	489
<i>Figure 5-8: Data-Cache Cacheability Register (DCCR)</i>	490
<i>Figure 5-9: Instruction-Cache Cacheability Register (ICCR)</i>	490
<i>Figure 5-10: Storage Guarded Register (SGR)</i>	491
<i>Figure 5-11: Storage User-Defined 0 Register (SU0R)</i>	491
<i>Figure 5-12: Storage Little-Endian Register (SLER)</i>	492
<i>Figure 5-13: Core-Configuration Register (CCR0)</i>	496
<i>Figure 5-14: Example of Shared-Memory Allocation</i>	502
<i>Figure 5-15: Instruction-Cache Debug-Data Register (ICDBDR)</i>	504
<i>Figure 5-16: Information Fields Loaded by dcread into rD</i>	505

Chapter 6: Virtual-Memory Management

<i>Figure 6-1: Virtual-Mode Address Translation</i>	508
<i>Figure 6-2: Process-Mapping Example</i>	509
<i>Figure 6-3: Process-ID Register (PID)</i>	510
<i>Figure 6-4: Page-Translation Table and TLB Organization</i>	511
<i>Figure 6-5: ITLB/DTLB/UTLB Address Translation Flow</i>	512
<i>Figure 6-6: TLB-Entry Format</i>	513
<i>Figure 6-7: General Process for Examining a TLB Entry</i>	516

<i>Figure 6-8: Zone-Protection Register (ZPR)</i>	519

Chapter 7: Exceptions and Interrupts

<i>Figure 7-1: PPC405 Exception Mechanism</i>	529
<i>Figure 7-2: Save/Restore Registers 0 and 1</i>	535
<i>Figure 7-3: Save/Restore Registers 2 and 3</i>	535
<i>Figure 7-4: Exception-Vector Prefix Register (EVPR)</i>	536
<i>Figure 7-5: Exception-Syndrom Register (ESR)</i>	536
<i>Figure 7-6: Data Exception-Address Register (DEAR)</i>	538

Chapter 8: Timer Resources

<i>Figure 8-1: PPC405 Timer Resources</i>	560
<i>Figure 8-2: Time-Base Register</i>	560
<i>Figure 8-3: Programmable-Interval Timer Register (PIT)</i>	563
<i>Figure 8-4: Timer-Control Register (TCR)</i>	564
<i>Figure 8-5: Timer-Status Register (TSR)</i>	565
<i>Figure 8-6: Watchdog-Event State Machine</i>	567

Chapter 9: Debugging

<i>Figure 9-1: Debug-Control Register 0 (DBCR0)</i>	574
<i>Figure 9-2: Debug-Control Register 1 (DBCR1)</i>	575
<i>Figure 9-3: Debug-Status Register (DBSR)</i>	577
<i>Figure 9-4: Instruction Address-Compare Registers (IAC1–IAC4)</i>	579
<i>Figure 9-5: Data Address-Compare Registers (DAC1, DAC2)</i>	579
<i>Figure 9-6: Data Value-Compare Registers (DVC1, DVC2)</i>	579
<i>Figure 9-7: IAC Address-Range Specification</i>	585
<i>Figure 9-8: DAC Address-Range Specification</i>	588
<i>Figure 9-9: JTAG-Connector Physical Layout</i>	594

Chapter 10: Reset and Initialization

Chapter 11: Instruction Set

<i>Figure 11-1: Instruction Description Format</i>	605

Appendix A: Register Summary

Appendix B: Instruction Summary

Appendix C: Simplified Mnemonics

Appendix D: Programming Considerations

<i>Figure D-1: String Access Example</i>	883
<i>Figure D-2: Multiple-Word Access Example</i>	884

Appendix E: PowerPC® 6xx/7xx Compatibility

Appendix F: PowerPC® Book-E Compatibility

Volume 2(b):

PPC405 Processor Block Manual

About This Book

Chapter 1: Introduction to the PowerPC® 405 Processor

Figure 1-1: PPC405 Registers	922
Figure 1-2: PPC405 Organization.....	924

Chapter 2: Input/Output Interfaces

Figure 2-1: CPM Interface Block Symbol	933
Figure 2-2: CPU Control Interface Block Symbol	937
Figure 2-3: Reset Interface Block Symbol.....	940
Figure 2-4: Instruction-Side PLB Interface Block Symbol	946
Figure 2-5: Attachment of ISPLB Between 32-Bit Slave and 64-Bit Master	951
Figure 2-6: ISPLB Non-Pipelined Cacheable Sequential Fetch (Case 1)	956
Figure 2-7: ISPLB Non-Pipelined Cacheable Sequential Fetch (Case 2)	957
Figure 2-8: ISPLB Pipelined Cacheable Sequential Fetch (Case 1).....	958
Figure 2-9: ISPLB Pipelined Cacheable Sequential Fetch (Case 2).....	959
Figure 2-10: ISPLB Non-Pipelined Non-Cacheable Sequential Fetch	960
Figure 2-11: ISPLB Pipelined Non-Cacheable Sequential Fetch.....	961
Figure 2-12: ISPLB 2:1 Core-to-PLB Line Fetch.....	962
Figure 2-13: ISPLB 3:1 Core-to-PLB Line Fetch.....	963
Figure 2-14: ISPLB Aborted Fetch Request	964
Figure 2-15: Data-Side PLB Interface Block Symbol	968
Figure 2-16: Attachment of DSPLB Between 32-Bit Slave and 64-Bit Master.....	972
Figure 2-17: DSPLB Three Consecutive Line Reads.....	982
Figure 2-18: DSPLB Line Read/Word Read/Line Read	983
Figure 2-19: DSPLB Three Consecutive Word Reads.....	984
Figure 2-20: DSPLB Three Consecutive Line Writes	985
Figure 2-21: DSPLB Line Write/Word Write/Line Write.....	986
Figure 2-22: DSPLB Three Consecutive Word Writes	987
Figure 2-23: DSPLB Line Write/Line Read/Word Write.....	988
Figure 2-24: DSPLB Word Write/Word Read/Word Write/Line Read.....	989
Figure 2-25: DSPLB Word Write/Line Read/Line Write.....	990
Figure 2-26: DSPLB 2:1 Core-to-PLB Line Read	991
Figure 2-27: DSPLB 3:1 Core-to-PLB Line Write	992
Figure 2-28: DSPLB Aborted Data-Access Request	993
Figure 2-29: DCR Chain Block Diagram	995
Figure 2-30: DCR Bus Implementation	996

Figure 2-31: DCR Interface Block Symbol.....	997
Figure 2-32: DCR Interface Mode 0, 1:1 Clocking, Latched Acknowledge	999
Figure 2-33: DCR Interface Mode 0, 1:1 Clocking, Combinatorial Acknowledge	1000
Figure 2-34: DCR Interface Mode 0, 2:1 Clocking, Latched Acknowledge	1001
Figure 2-35: DCR Interface Mode 0, 1:2 Clocking, Latched Acknowledge	1002
Figure 2-36: DCR Interface Mode 1, 1:1 Clocking, Combinatorial Acknowledge	1003
Figure 2-37: EIC Interface Block Symbol.....	1004
Figure 2-38: JTAG Interface Block Symbol	1007
Figure 2-39: Debug Interface Block Symbol	1011
Figure 2-40: Trace Interface Block Symbol.....	1014
Figure 2-41: FPGA-Specific Interface Block Symbol.....	1017

Chapter 3: PowerPC® 405 OCM Controller

Figure 3-1: DSOCM Block	1021
Figure 3-2: DSOCM to BRAM Interface: 8K Example	1022
Figure 3-3: ISOCM Block	1023
Figure 3-4: ISOCM to BRAM Interface: 8K Example	1024
Figure 3-5: Single-Cycle Static Performance Analysis of DSOCM (Load Operation)	1026
Figure 3-6: Multi-Cycle Static Performance Analysis of DSOCM (Load Operation)	1027
Figure 3-7: DSOCM DCR Registers.....	1028
Figure 3-8: ISOCM DCR Registers	1029

Appendix A: RISCWatch and RISCTrace Interfaces

Figure A-1: JTAG-Connector Physical Layout.....	1039
Figure A-2: Trace-Connector Physical Layout.....	1041

Volume 3: Rocket I/O™ Transceiver User Guide

Chapter 1: Introduction

Chapter 2: Rocket I/O™ Transceiver Overview

Figure 2-1: Rocket I/O Transceiver Block Diagram	1060
Figure 2-2: Clock Correction in Receiver	1065
Figure 2-3: Channel Bonding (Alignment)	1066

Chapter 3: Digital Design Considerations

Figure 3-1: Two-Byte Clock	1084
Figure 3-2: Four-Byte Clock	1087
Figure 3-3: One-Byte Clock.....	1090
Figure 3-4: Multiplexed REFCLK	1094
Figure 3-5: Rocket I/O Timing Relative to Clock Edge	1096
Figure 3-6: 8B/10B Data Flow	1100
Figure 3-7: 10-Bit TX Data Map with 8B/10B Bypassed	1102

<i>Figure 3-8: 10-Bit RX Data Map with 8B/10B Bypassed</i>	1102
<i>Figure 3-9: 8B/10B Parallel to Serial Conversion</i>	1112
<i>Figure 3-10: 4-Byte Serial Structure.....</i>	1113
<i>Figure 3-11: CRC Packet Format</i>	1114
<i>Figure 3-12: USER_MODE / FIBRE_CHANNEL Mode</i>	1115
<i>Figure 3-13: Ethernet Mode</i>	1115
<i>Figure 3-14: Infiniband Mode</i>	1115
<i>Figure 3-15: Local Route Header.....</i>	1116

Chapter 4: Analog Design Considerations

<i>Figure 4-1: Differential Amplifier.....</i>	1119
<i>Figure 4-2: Alternating K28.5+ with No Pre-Emphasis</i>	1121
<i>Figure 4-3: K28.5+ with Pre-Emphasis.....</i>	1121
<i>Figure 4-5: Eye Diagram: with 30% Pre-Emphasis</i>	1122
<i>Figure 4-4: Eye Diagram: without Pre-Emphasis</i>	1122
<i>Figure 4-6: Power Supply Circuit Using LT1963 Regulator</i>	1125
<i>Figure 4-7: Power Filtering Network for One Transceiver.....</i>	1126
<i>Figure 4-8: Example Power Filtering PCB Layout for Four MGTs, Top Layer.....</i>	1126
<i>Figure 4-9: Example Power Filtering PCB Layout for Four MGTs, Bottom Layer.....</i>	1127
<i>Figure 4-10: Single-Ended Trace Geometry.....</i>	1128
<i>Figure 4-11: Microstrip Edge-Coupled Differential Pair</i>	1129
<i>Figure 4-12: Stripline Edge-Coupled Differential Pair</i>	1129
<i>Figure 4-13: AC-Coupled Serial Link</i>	1129
<i>Figure 4-14: DC-Coupled Serial Link</i>	1129
<i>Figure 4-15: Reference CLock Oscillator Interface</i>	1130

Chapter 5: Simulation and Implementation

<i>Figure 5-1: 2VP2 Implementation.....</i>	1132
<i>Figure 5-2: 2VP50 Implementation.....</i>	1132

Appendix A: Rocket I/O™ Cell Models

Volume 4: Design Flow

Chapter 1: Introduction to Virtex-II Pro™ Design Flow

<i>Figure 1-1: Virtex-II Pro Design Flow</i>	1154

Chapter 2: Design Flow Overview

<i>Figure 2-1: Steps for Building Hello.elf</i>	1160
<i>Figure 2-2: Hardware Overview</i>	1164
<i>Figure 2-3: Generating Software Executable and Memory Initialization Files.....</i>	1166
<i>Figure 2-4: Performing a Functional Simulation.....</i>	1167
<i>Figure 2-5: Synthesizing the Design.....</i>	1170

<i>Figure 2-6: Placing and Routing the Design</i>	1172
<i>Figure 2-7: Performing Back-Annotated Timing Simulation</i>	1174
<i>Figure 2-8: Generating Bitstream and PROM Files</i>	1175

Chapter 3: PPC405 Bus Functional Model Usage

<i>Figure 3-1: Structure of PPC405 BFM</i>	1189
<i>Figure 3-2: Compilation Procedure</i>	1191

Chapter 4: SmartModel® Usage

Volume 5: Software Development Tools

Chapter 1: Introduction to Software Development Tools

Chapter 2: Using Data2BRAM

<i>Figure 2-1: Basic Input & Output Files of Data2BRAM</i>	1205
<i>Figure 2-2: Software and Hardware Flows Using Data2BRAM</i>	1210
<i>Figure 2-3: Example Layout of a 64KB Memory for PPC405 using BRAMs</i>	1215
<i>Figure 2-4: Example of Bitwise Swapping on BRAMs.....</i>	1217

Chapter 3: GNU Software Tool Chain

<i>Figure 3-1: Steps for Building Hello.elf</i>	1236

Volume 6: Reference Systems

Chapter 1: Introduction to Reference Systems

Chapter 2: Simple PPC405 Reference System

<i>Figure 2-1: High-Level Hardware View of Simple PPC405 Reference System.....</i>	1251
<i>Figure 2-2: Clock Generation</i>	1253
<i>Figure 2-3: Simple Reference System Reset Logic</i>	1254
<i>Figure 2-4: Organization of Higher-Level HDL Files.....</i>	1255
<i>Figure 2-5: Map of Directories Below \$V2PRO/platforms/simple</i>	1257
<i>Figure 2-6: Design Flow for Functional Simulation</i>	1265

Chapter 3: PPC405 PPE Reference System Using Rocket I/O™ Transceivers

<i>Figure 3-1: Serial Packet Flow</i>	1267
<i>Figure 3-2: High-Level Hardware View of the PPE Reference System.....</i>	1268
<i>Figure 3-3: Clock Generation</i>	1272

<i>Figure 3-4: Rocket I/O Transceiver Serial Frame Format.....</i>	1272
<i>Figure 3-5: Design Flow</i>	1279
<i>Figure 3-6: Organization of Higher-Level HDL Files.....</i>	1280
<i>Figure 3-7: Map of Directories Below \$V2PRO/platforms/pkt_proc.....</i>	1282

Chapter 4: Embedded PPC405 Reference System

<i>Figure 4-1: High-Level Hardware View of Embedded PPC405 Reference System</i>	1292
<i>Figure 4-2: Clock Generation</i>	1296
<i>Figure 4-3: Embedded Reference System Reset Logic</i>	1297
<i>Figure 4-4: Organization of Higher-Level HDL Files.....</i>	1299
<i>Figure 4-5: Map of Directories Below \$V2PRO/platforms/embedded</i>	1302
<i>Figure 4-6: Design Flow for Functional Simulation</i>	1314

Volume 7: Hardware IP Specifications

Chapter 1: Introduction to Hardware IP

Chapter 2: Using IPIF to Build IP

<i>Figure 2-1: IPIF SRAM Module Interface</i>	1324
<i>Figure 2-2: IPIF Simple SRAM Write Cycle</i>	1325
<i>Figure 2-3: IPIF Simple SRAM Read Cycle</i>	1326
<i>Figure 2-4: IPIF SRAM Module to GPIO logic Interface.....</i>	1327

Chapter 3: IPIF Specification

<i>Figure 3-1: Scope of Interfaces in This Specification</i>	1331
<i>Figure 3-2: General Model of the IPIF Backplane.....</i>	1333
<i>Figure 3-3: Signal Name Format Example</i>	1333
<i>Figure 3-4: Example System Using IPIF to Connect to OPB</i>	1335
<i>Figure 3-5: Example of Three Interrupt Scenarios</i>	1337
<i>Figure 3-6: TBD</i>	1338
<i>Figure 3-7: Example Slave DMA Handshake Application.....</i>	1339
<i>Figure 3-8: Generic Instance of the IPIF Slave DMA Handshake Module</i>	1340
<i>Figure 3-9: IPIF Slave DMA Handshake Signal Protocol.....</i>	1341
<i>Figure 3-10: Example Application of an IPIF Slave Control Register Module.....</i>	1345
<i>Figure 3-11: Generic Instance of the IPIF Slave Control Register</i>	1347
<i>Figure 3-12: Timing Diagrams for Slave IP Module Register Protocol Read & Write</i>	1348
<i>Figure 3-13: Example Application of IPIF Slave SRAM Module.....</i>	1352
<i>Figure 3-14: Slave IP Module SRAM Protocol Block Diagram</i>	1354
<i>Figure 3-15: IPIF Slave SRAM Module Single Write Transaction</i>	1355
<i>Figure 3-16: IPIF Slave SRAM Module Single Read Transaction</i>	1356
<i>Figure 3-17: IPIF Slave SRAM Module Write Burst Transaction</i>	1356
<i>Figure 3-18: IPIF Slave SRAM Module Read Burst Transaction</i>	1357
<i>Figure 3-19: Example Application of IPIF Slave FIFO Module.....</i>	1361

<i>Figure 3-20: Generic Slave FIFO Module Block Diagram</i>	1363
<i>Figure 3-21: Timing Diagram for Slave IP FIFO Module Read and Write Transactions</i>	1364
<i>Figure 3-22: Example Application of IPIF Master Module</i>	1369
<i>Figure 3-23: Master IP Protocol Block Diagram</i>	1370
<i>Figure 3-24: Timing Diagram for the Master IP Module Protocol Single Write Transaction</i>	1372
<i>Figure 3-25: Timing Diagram for the Master IP Module Protocol Single Read Transaction</i>	1373
<i>Figure 3-26: Timing Diagram for the Master IP Module Protocol Burst Write Transaction</i>	1374
<i>Figure 3-27: Timing Diagram for the Master IP Module Protocol Burst Read Transaction</i>	1375

Chapter 4: OPB Arbiter

<i>Figure 4-1: OPB Basic Bus Arbitration</i>	1386
<i>Figure 4-2: Continuous Master Bus Request</i>	1386
<i>Figure 4-3: Bus Locking</i>	1387
<i>Figure 4-4: Multiple Bus Request Arbitration</i>	1387
<i>Figure 4-5: OPB Arbiter Priority Register</i>	1389
<i>Figure 4-6: OPB Arbiter Control Register</i>	1390
<i>Figure 4-7: OPB Arbiter Top-level Block Diagram</i>	1391
<i>Figure 4-8: IPIF Data Transfer Protocol</i>	1392
<i>Figure 4-9: Fixed Priority Arbitration</i>	1393
<i>Figure 4-10: Dynamic Priority Arbitration</i>	1393
<i>Figure 4-11: Priority Register Logic</i>	1394
<i>Figure 4-12: Arbitration Logic</i>	1395
<i>Figure 4-13: Park/Lock Logic</i>	1396
<i>Figure 4-14: Bus Parking on Master Not Last</i>	1397
<i>Figure 4-15: Bus Parking on Last Master</i>	1398
<i>Figure 4-16: OPB Timeout Error</i>	1399
<i>Figure 4-17: OPB Timeout Error Suppression</i>	1399

Chapter 5: OPB Configuration ROM Module

<i>Figure 5-1: Schematic of CROM Device Implemented with IPIF Module</i>	1403

Chapter 6: OPB GPIO Module

<i>Figure 6-1: Schematic of GPIO Module Using the IPIF Slave SRAM Module</i>	1407
<i>Figure 6-2: Schematic of GPIO Module Using the IPIF Slave Control Register Module</i>	1408

Chapter 7: OPB LCD Controller

<i>Figure 7-1: Schematic of LCD Controller Using the IPIF Slave SRAM Module</i>	1413
<i>Figure 7-2: Read Cycle Timing Diagram</i>	1414
<i>Figure 7-3: Write Cycle Timing Diagram</i>	1414

Chapter 8: OPB 16450 UART Controller

Figure 8-1: UART Top-Level Block Diagram..... 1426

Chapter 9: OPB 16550 UART Controller

Figure 9-1: UART Top-Level Block Diagram..... 1437

Chapter 10: OPB IIC Controller

Figure 10-1: Data Transfer on the IIC Bus 1440

Figure 10-2: IIC Top-level Block Diagram..... 1449

Chapter 11: OPB to DCR Bridge

Figure 11-1: OPB to DCR Bridge 1453

Chapter 12: OPB to PCI Bridge

Figure 12-1: OPB to PCI Bridge..... 1458

Chapter 13: PLB Arbiter

Figure 13-1: PLB Arbiter Interconnection..... 1461

Figure 13-2: PLB Arbiter Block Diagram 1462

Figure 13-3: Master[n] Interface..... 1467

Figure 13-4: Slave Interface 1476

Figure 13-5: DCR Interface 1491

Figure 13-6: Clock Interface..... 1492

Figure 13-7: PLB Arbiter Control Register (PACR)..... 1494

Figure 13-8: PLB Error Status Register (PESR) 1495

Figure 13-9: Single Read Transfer Bus Time-Out 1499

Figure 13-10: Single Write Transfer Bus Time-Out 1500

Figure 13-11: Line Read Transfer Bus Time-Out..... 1501

Figure 13-12: Line Write Transfer Bus Time-Out..... 1502

Figure 13-13: Burst Read Transfer Bus Time-Out 1503

Figure 13-14: Burst Write Transfer Bus Time-Out 1504

Figure 13-15: Read Transfer..... 1505

Figure 13-16: Back-to-Back Reads..... 1506

Figure 13-17: Pipelined Write Transfer 1507

Chapter 14: PLB BRAM Controller

Figure 14-1: High-Level View of PLB BRAM Controller 1512

Chapter 15: PLB UART Controller

Figure 15-1: Connection Between PLB IPIF and Synthesizable UART Core 1519

Chapter 16: PLB SRAM/Flash Controller

Figure 16-1: SRAM/Flash Controller Module Block Diagram 1521

Figure 16-2: Mapping between the PLB and External SRAM Address 1524

<i>Figure 16-3: Controller State Machine</i>	1525
<i>Figure 16-4: Timing Diagram for Cacheline Write of Two Doublewords</i>	1526
<i>Figure 16-5: Timing Diagram for Cacheline Read of Two Doublewords</i>	1527
<i>Figure 16-6: Flash Write Cycle</i>	1528
<i>Figure 16-7: Flash Read Cycle</i>	1529

Chapter 17: PLB ZBT SRAM Controller

<i>Figure 17-1: ZBT SRAM Controller Module Block Diagram</i>	1533
<i>Figure 17-2: Read Transaction Timing</i>	1537
<i>Figure 17-3: Write Transaction Timing</i>	1538
<i>Figure 17-4: Mapping Between the ZBT SRAM Controller & External ZBT SRAMs</i>	1539
<i>Figure 17-5: Controller State Machine Overview</i>	1540

Chapter 18: PLB DDR SDRAM Controller

<i>Figure 18-1: PLB DDR SDRAM Controller Module Block Diagram</i>	1543
<i>Figure 18-2: DDR Controller Block Diagram</i>	1546
<i>Figure 18-3: Mapping Between the PLB and the External DDR SDRAM Address</i>	1547
<i>Figure 18-4: Read and Write Data Flow</i>	1548
<i>Figure 18-5: Controller State Machine Overview</i>	1549
<i>Figure 18-6: Timing Diagram for Cacheline Write of Two Double Words</i>	1551
<i>Figure 18-7: Timing Diagram for Cacheline Read of Two Double Words</i>	1552

Chapter 19: PLB to OPB Bridge-Out Module

<i>Figure 19-1: High-Level Overview of the PLB to OPB Bridge</i>	1557

Chapter 20: Interrupt Controller

<i>Figure 20-1: INTC Block Diagram</i>	1568
<i>Figure 20-2: INTC Programmer's Model</i>	1569
<i>Figure 20-3: INTC Physical Hardware Model</i>	1569
<i>Figure 20-4: Interrupt Status Register (ISR)</i>	1570
<i>Figure 20-5: Interrupt Pending Register (IPR)</i>	1571
<i>Figure 20-6: Interrupt Enable Register (IER)</i>	1572
<i>Figure 20-7: Interrupt Acknowledge Register (IAR)</i>	1573
<i>Figure 20-8: Set Interrupt Enable (SIE) Register</i>	1574
<i>Figure 20-9: Clear Interrupt Enable (CIE) Register</i>	1575
<i>Figure 20-10: Interrupt Vector Register (IVR)</i>	1576
<i>Figure 20-11: Master Enable Register (MER)</i>	1578
<i>Figure 20-12: Expanding INTC interrupt sources</i>	1580

Chapter 21: Packet Processing Engine Controller

<i>Figure 21-1: Overview of the Packet Processing Engine Interface Logic</i>	1587
<i>Figure 21-2: Rocket I/O Transceiver Serial Frame Format</i>	1589

Volume 8: Software IP and Applications

Chapter 1: Introduction to Software IP and Applications

Chapter 2: Stand-Alone Board Support Package

**Chapter 3: PPC405 Software:
Demonstration and Test Applications**

Chapter 4: Simulation Terminal

Schedule of Tables

Volume 1: Advance Product Specification

Data Sheet Module 1:

Virtex-II Pro™ Platform FPGAs: Introduction and Overview

Table 1: Virtex-II Pro FPGA Family Members	75
Table 2: Dual-Port and Single-Port Configurations.....	78
Table 3: Virtex-II Pro Device/Package Combinations and Maximum Number of Available I/Os (Advance Information)	80

Data Sheet Module 2:

Virtex-II Pro™ Platform FPGAs: Functional Description

Table 1: Standards Supported by the Rocket I/O MGT	83
Table 2: Supported Rocket I/O Transceiver Primitives.....	89
Table 3: Supported Single-Ended I/O Standards	96
Table 4: Supported Differential Signal I/O Standards.....	97
Table 5: Supported DCI I/O Standards	97
Table 6: LVTTL and LVCMOS Programmable Currents (Sink and Source)	100
Table 7: Compatible Output Standards.....	101
Table 8: SelectI/O Controlled Impedance Buffers	102
Table 9: SelectI/O Buffers With On-Chip Parallel Termination	102
Table 10: Distributed SelectRAM Configurations	106
Table 11: ROM Configuration.....	107
Table 12: Virtex-II Pro 3-State Buffers	112
Table 13: Logic Resources in One CLB	112
Table 14: Virtex-II Pro Logic Resources Available in All CLBs	113
Table 15: Dual- and Single-Port Configurations	113
Table 16: Dual-Port Mode Configurations	114
Table 17: 18 Kb Block SelectRAM Port Aspect Ratio	114
Table 18: Control Functions.....	115
Table 19: Virtex-II Pro SelectRAM Memory Available.....	116
Table 20: Multiplier Resources	117
Table 21: DCM Status Pins	120
Table 22: CLKDV Duty Cycle for Non-integer Divides	120
Table 23: Fine Phase Shifting Control Pins	120
Table 24: DCM Frequency Ranges	121
Table 25: DCM Organization	122
Table 26: Virtex-II Pro Configuration Mode Pin Settings	124
Table 27: Virtex-II Pro Bitstream Lengths.....	125

Data Sheet Module 3:

Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics

Table 1: Absolute Maximum Ratings	127
Table 2: Recommended Operating Conditions	128
Table 3: DC Characteristics Over Recommended Operating Conditions	128
Table 4: Quiescent Supply Current	129
Table 5: Power-On Current for Virtex-II Pro Devices	130
Table 6: DC Input and Output Levels.....	130
Table 7: LDT DC Specifications.....	131
Table 8: LVDS DC Specifications.....	131
Table 9: Extended LVDS DC Specifications.....	131
Table 10: Rocket I/O DC Specifications	132
Table 11: Pin-to-Pin Performance	133
Table 12: Register-to-Register Performance	133
Table 13: Virtex-II Pro Device Speed Grade Designations	135
Table 14: Processor Clocks Absolute AC Characteristics.....	136
Table 15: Processor Block Switching Characteristics.....	136
Table 16: Processor Block PLB Switching Characteristics	137
Table 17: Processor Block JTAG Switching Characteristics.....	137
Table 18: PowerPC 405 Data-Side On-Chip Memory Switching Characteristics.....	138
Table 19: PowerPC 405 Instruction-Side On-Chip Memory Switching Characteristics	138
Table 20: Rocket I/O Reference Clock Switching Characteristics.....	139
Table 21: Rocket I/O Receiver Switching Characteristics.....	140
Table 22: Rocket I/O Transmitter Switching Characteristics	141
Table 23: Rocket I/O RXUSRCLK Switching Characteristics	142
Table 24: Rocket I/O RXUSRCLK2 Switching Characteristics	142
Table 25: Rocket I/O TXUSRCLK Switching Characteristics	143
Table 26: IOB Input Switching Characteristics.....	144
Table 27: IOB Input Switching Characteristics Standard Adjustments	145
Table 28: IOB Output Switching Characteristics.....	146
Table 29: IOB Output Switching Characteristics Standard Adjustments	147
Table 30: Delay Measurement Methodology	151
Table 31: Standard Capacitive Loads	152
Table 32: Clock Distribution Switching Characteristics.....	153
Table 33: CLB Switching Characteristics	153
Table 34: CLB Distributed RAM Switching Characteristics	154
Table 35: CLB Shift Register Switching Characteristics	154
Table 36: Multiplier Switching Characteristics	155
Table 37: Block SelectRAM Switching Characteristics	156
Table 38: TBUF Switching Characteristics.....	156
Table 39: JTAG Test Access Port Switching Characteristics	156
Table 40: Global Clock Input to Output Delay for LVCMOS25, 12 mA, Fast Slew Rate, With DCM.....	157
Table 41: Global Clock Input to Output Delay for LVCMOS25, 12 mA, Fast Slew Rate, Without DCM.....	158

Table 42: Global Clock Set-Up and Hold for LVCMOS25 Standard, With DCM	159
Table 43: Global Clock Set-Up and Hold for LVCMOS25 Standard, Without DCM ..	160
Table 44: Operating Frequency Ranges	160
Table 45: Input Clock Tolerances	162
Table 46: Output Clock Jitter	163
Table 47: Output Clock Phase Alignment	163
Table 48: Miscellaneous Timing Parameters	164
Table 49: Frequency Synthesis	164
Table 50: Parameter Cross-Reference	165

Data Sheet Module 4:

Virtex-II Pro™ Platform FPGAs: Pinout Information

Table 1: Wire-Bond Packages Information	167
Table 2: Flip-Chip Packages Information	167
Table 3: Virtex-II Pro Available I/Os and Rocket I/O MGT Pins per Device/Package Combination	167
Table 4: 3.3V SelectI/O Banks	168
Table 5: Virtex-II Pro Pin Definitions	169
Table 6: FG256 — XC2VP2 and XC2VP4	171
Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7	180
Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7	195
Table 9: FF896 — XC2VP7 and XC2VP20	216
Table 10: FF1152 — XC2VP20 and XC2VP50	242
Table 11: FF1517 — XC2VP50	276
Table 12: BF957 — XC2VP20 and XC2VP50	315

Volume 2:

PowerPC® 405 Processor

Volume 2(a):

PPC405 User Manual

About This Book

Table P-1: General Notational Conventions	348
Table P-2: Instruction Field Definitions	349
Table P-3: Pseudocode Conventions	351
Table P-4: Operator Precedence	353
Table P-5: PPC405 Registers	353

Chapter 1: Introduction to the PPC405

Table 1-1: Three Levels of PowerPC Architecture	360
Table 1-2: Operating-Environment Features of the PowerPC Embedded-Environment Architecture	364

Chapter 2: Operational Concepts

<i>Table 2-1: Memory Operand Alignment Requirements</i>	389
<i>Table 2-2: Performance Effects of Operand Alignment</i>	390
<i>Table 2-3: Instructions Causing Alignment Exceptions</i>	390
<i>Table 2-4: PowerPC Embedded-Environment Instructions.....</i>	393

Chapter 3: User Programming Model

<i>Table 3-1: CR0-Field Bit Settings</i>	398
<i>Table 3-2: CRn-Field Bit Settings</i>	398
<i>Table 3-3: Fixed Point Exception Register (XER) Bit Definitions.....</i>	399
<i>Table 3-4: BO Field Bit Definitions</i>	403
<i>Table 3-5: Valid BO Opcode-Field Encoding</i>	404
<i>Table 3-6: Branch-Unconditional Instructions</i>	405
<i>Table 3-7: Branch-Conditional Instructions</i>	405
<i>Table 3-8: Branch-Conditional to Link-Register Instructions.....</i>	405
<i>Table 3-9: Branch-Conditional to Count-Register Instructions</i>	406
<i>Table 3-10: Condition-Register Logical Instructions</i>	412
<i>Table 3-11: System-Call Instruction</i>	412
<i>Table 3-12: System-Trip Instructions</i>	413
<i>Table 3-13: TO Field Bit Definitions.....</i>	413
<i>Table 3-14: Load Byte and Zero Instructions</i>	417
<i>Table 3-15: Load Halfword and Zero Instructions</i>	418
<i>Table 3-16: Load-Word and Zero Instructions.....</i>	418
<i>Table 3-17: Load Halfword Algebraic Instructions</i>	419
<i>Table 3-18: Store Byte Instructions</i>	420
<i>Table 3-19: Store Halfword Instructions</i>	421
<i>Table 3-20: Store Word Instructions</i>	421
<i>Table 3-21: Load and Store with Byte-Reverse Instructions.....</i>	422
<i>Table 3-22: Load and Store Multiple Instructions</i>	423
<i>Table 3-23: Load and Store String Instructions.....</i>	424
<i>Table 3-24: Integer-Addition Instructions.....</i>	426
<i>Table 3-25: Integer-Subtraction Instructions.....</i>	428
<i>Table 3-26: Negation Instructions.....</i>	430
<i>Table 3-27: Multiply Instructions</i>	430
<i>Table 3-28: Divide Instructions.....</i>	431
<i>Table 3-29: AND and NAND Instructions.....</i>	432
<i>Table 3-30: OR and NOR Instructions.....</i>	432
<i>Table 3-31: XOR and Equivalence Instructions</i>	433
<i>Table 3-32: Sign-Extension Instructions</i>	434
<i>Table 3-33: Count Leading-Zeros Instructions.....</i>	434
<i>Table 3-34: Algebraic-Comparison Instructions.....</i>	435
<i>Table 3-35: Logical-Comparison Instructions.....</i>	435
<i>Table 3-36: Rotate Left then AND-with-Mask Instructions</i>	436
<i>Table 3-37: Rotate Left then Mask-Insert Instructions</i>	438
<i>Table 3-38: Logical-Shift Instructions.....</i>	439

<i>Table 3-39: Algebraic-Shift Instructions</i>	440
<i>Table 3-40: Multiply-Accumulate Cross-Halfword to Word Instructions</i>	442
<i>Table 3-41: Multiply-Accumulate High-Halfword to Word Instructions.....</i>	444
<i>Table 3-42: Multiply-Accumulate Low-Halfword to Word Instructions</i>	447
<i>Table 3-43: Negative Multiply-Accumulate Cross-Halfword to Word Instructions.....</i>	450
<i>Table 3-44: Negative Multiply-Accumulate High-Halfword to Word Instructions</i>	452
<i>Table 3-45: Negative Multiply-Accumulate Low-Halfword to Word Instructions.....</i>	454
<i>Table 3-46: Multiply Cross-Halfword to Word Instructions</i>	455
<i>Table 3-47: Multiply High-Halfword to Word Instructions</i>	456
<i>Table 3-48: Multiply Low-Halfword to Word Instructions</i>	457
<i>Table 3-49: Condition-Register Move Instructions</i>	459
<i>Table 3-50: Special-Purpose Register Instructions</i>	460
<i>Table 3-51: Synchronizing Instructions.....</i>	461
<i>Table 3-52: Synchronization Effects of PowerPC Instructions</i>	462
<i>Table 3-53: Semaphore Synchronization Instructions.....</i>	462
<i>Table 3-54: Memory-Control Instructions, User Mode.....</i>	463

Chapter 4: PPC405 Privileged-Mode Programming Model

<i>Table 4-1: Machine-State Register (MSR) Bit Definitions.....</i>	467
<i>Table 4-2: Processor-Version Register (PVR) Bit Definitions.....</i>	469
<i>Table 4-3: PPC405 Privileged Instructions.....</i>	470
<i>Table 4-4: System-Linkage Instruction.....</i>	471
<i>Table 4-5: Machine-State Register Instructions</i>	471
<i>Table 4-6: Special-Purpose Register Instructions</i>	472
<i>Table 4-7: Device Control Register Instructions.....</i>	472

Chapter 5: Memory-System Management

<i>Table 5-1: Data-Cache to PLB Priority Examples.....</i>	483
<i>Table 5-2: Storage-Attribute Control-Register Address Ranges.....</i>	489
<i>Table 5-3: Privileged and User Cache-Control Instructions</i>	492
<i>Table 5-4: Instruction-Cache Control Instructions</i>	493
<i>Table 5-5: Data-Cache Control Instructions.....</i>	494
<i>Table 5-6: Core-Configuration Register (CCR0) Field Definitions.....</i>	496
<i>Table 5-7: PLB-Request Priority Encoding</i>	497
<i>Table 5-8: Instruction-Cache Debug-Data Register (ICDBDR) Field Definitions.....</i>	504
<i>Table 5-9: dcread Information-Field Definitions</i>	505

Chapter 6: Virtual-Memory Management

<i>Table 6-1: Process-ID Register (PID) Field Definitions</i>	510
<i>Table 6-2: Page-Translation Bit Ranges by Page Size.....</i>	514
<i>Table 6-3: Zone-Protection Register (ZPR) Bit Definitions</i>	519
<i>Table 6-4: Effect of Cache-Control Instruction Access Violations</i>	520
<i>Table 6-5: TLB-Management Instructions</i>	521

Chapter 7: Exceptions and Interrupts

<i>Table 7-1: Exceptions Supported by the PPC405D5</i>	527
<i>Table 7-2: Interrupt Priority for Simultaneous Exceptions</i>	531
<i>Table 7-3: Effect of Interrupts on Machine-State Register Contents</i>	534
<i>Table 7-4: Exception-Vector Prefix Register (EVPR) Field Definitions</i>	536
<i>Table 7-5: Exception-Syndrome Register (ESR) Field Definitions</i>	537

Chapter 8: Timer Resources

<i>Table 8-1: Time-Base Register Instructions</i>	561
<i>Table 8-2: Time-Base Register Numbers</i>	561
<i>Table 8-3: Timer-Control Register (TCR) Field Definitions</i>	564
<i>Table 8-4: Timer-Status Register (TSR) Field Definitions</i>	565
<i>Table 8-5: Watchdog Time-Out Periods</i>	566
<i>Table 8-6: Fixed-Interval Timer-Event Periods</i>	569

Chapter 9: Debugging

<i>Table 9-1: Debug-Control Register 0 (DBCR0) Field Definitions</i>	574
<i>Table 9-2: Debug-Control Register 1 (DBCR1) Field Definitions</i>	576
<i>Table 9-3: Debug-Status Register (DBSR) Field Definitions</i>	577
<i>Table 9-4: Debug Resources Used by Debug Events</i>	580
<i>Table 9-5: IAC Exact-Address Match Resources</i>	584
<i>Table 9-6: DAC Exact-Address Match Resources</i>	586
<i>Table 9-7: Effect of D1S/D2S Size-Field Encoding</i>	587
<i>Table 9-8: Examples of Using the D1S Size Field</i>	587
<i>Table 9-9: DAC Address-Range Match Resources</i>	588
<i>Table 9-10: DAC Events Caused by Cache-Control Instructions</i>	589
<i>Table 9-11: Examples of Using DVC1 Controls</i>	591
<i>Table 9-12: DVC Event Status</i>	591
<i>Table 9-13: JTAG Connector Signals</i>	594

Chapter 10: Reset and Initialization

<i>Table 10-1: MSR State Following Reset</i>	598
<i>Table 10-2: SPR Contents Following Reset</i>	599

Chapter 11: Instruction Set

Appendix A: Register Summary

<i>Table A-1: PPC405 Register Cross-Reference</i>	803
<i>Table A-2: General-Purpose Registers</i>	805
<i>Table A-3: Machine-State and Condition Registers</i>	806
<i>Table A-4: Special-Purpose Registers Sorted by Name</i>	806
<i>Table A-5: Special-Purpose Registers Sorted by SPRN</i>	808
<i>Table A-6: Special-Purpose Registers Sorted by SPRF</i>	809
<i>Table A-7: Time-Base Registers</i>	811

Appendix B: Instruction Summary

<i>Table B-1: Instructions Sorted by Mnemonic</i>	813
<i>Table B-2: Instructions Sorted by Opcode</i>	818
<i>Table B-3: Integer Add and Subtract Instructions</i>	822
<i>Table B-4: Integer Divide and Multiply Instructions</i>	823
<i>Table B-5: Integer Multiply-Accumulate Instructions</i>	823
<i>Table B-6: Integer Compare Instructions</i>	824
<i>Table B-7: Integer Logical Instructions</i>	824
<i>Table B-8: Integer Rotate Instructions</i>	824
<i>Table B-9: Integer Shift Instructions</i>	824
<i>Table B-10: Integer Load Instructions</i>	825
<i>Table B-11: Integer Store Instructions</i>	825
<i>Table B-12: Integer Load and Store with Byte Reverse Instructions</i>	826
<i>Table B-13: Integer Load and Store Multiple Instructions</i>	826
<i>Table B-14: Integer Load and Store String Instructions</i>	826
<i>Table B-15: Branch Instructions</i>	826
<i>Table B-16: Condition Register Logical Instructions</i>	826
<i>Table B-17: System Linkage Instructions</i>	827
<i>Table B-18: Trap Instructions</i>	827
<i>Table B-19: Synchronization Instructions</i>	827
<i>Table B-20: Processor Control Instructions</i>	827
<i>Table B-21: Cache Management Instructions</i>	828
<i>Table B-22: TLB Management Instructions</i>	828
<i>Table B-23: B Form</i>	828
<i>Table B-24: D Form</i>	828
<i>Table B-25: I Form</i>	829
<i>Table B-26: M Form</i>	829
<i>Table B-27: SC Form</i>	830
<i>Table B-28: X Form</i>	830
<i>Table B-29: XFX Form</i>	832
<i>Table B-30: XL Form</i>	832
<i>Table B-31: XO Form</i>	832
<i>Table B-32: Instruction Set Information</i>	834
<i>Table B-33: Complete List of Instruction Mnemonics</i>	839

Appendix C: Simplified Mnemonics

<i>Table C-1: Abbreviations for True/False Conditional Branches</i>	857
<i>Table C-2: Simplified Branch-Conditional Mnemonics, True/False Conditions</i>	858
<i>Table C-3: Branch (True/False) to Relative/Absolute (LK=0)</i>	858
<i>Table C-4: Branch (True/False) to LR/CTR (LK=0)</i>	859
<i>Table C-5: Branch (True/False) to Relative/Absolute (LK=1)</i>	859
<i>Table C-6: Branch (True/False) to LR/CTR (LK=1)</i>	860
<i>Table C-7: Abbreviations for Comparison Conditional Branches</i>	860
<i>Table C-8: Simplified Branch-Conditional Mnemonics, Comparison Conditions</i>	861
<i>Table C-9: Branch (Comparison) to Relative/Absolute (LK=0)</i>	861

Table C-10: Branch (Comparison) to LR/CTR (LK=0).....	862
Table C-11: Branch (Comparison) to Relative/Absolute (LK=1).....	862
Table C-12: Branch (Comparison) to LR/CTR (LK=1).....	863
Table C-13: Simplified Mnemonics for Compare Instructions.....	864
Table C-14: Simplified Mnemonics for CR-Logical Instructions.....	864
Table C-15: Simplified Mnemonics for Rotate and Shift Instructions.....	865
Table C-16: Simplified Mnemonics for Special-Purpose Register Instructions.....	866
Table C-17: Simplified Mnemonics for Subtract Instructions.....	868
Table C-18: Simplified Mnemonics for TLB-Management Instructions.....	868
Table C-19: Abbreviations for Trap Comparison Conditions.....	869
Table C-20: Simplified Mnemonics for Trap Instructions.....	869
Table C-21: Simplified Mnemonic for No-op.....	870
Table C-22: Simplified Mnemonics for Load Immediate.....	870
Table C-23: Simplified Mnemonic for Load Address.....	870
Table C-24: Simplified Mnemonics for Move Register.....	870
Table C-25: Simplified Mnemonics for Complement Register.....	871
Table C-26: Simplified Mnemonic for Move to Condition Register.....	871

Appendix D: Programming Considerations

Table D-1: Multiply and MAC Instruction Timing.....	882

Appendix E: PowerPC® 6xx/7xx Compatibility

Table E-1: 40x Registers Not Supported by 6xx/7xx Processors.....	885
Table E-2: 6xx/7xx Registers Not Supported by 40x Processors.....	886
Table E-3: Comparison of MSR Bit Definitions.....	887
Table E-4: Summary of Memory Translation Differences.....	889
Table E-5: 40x Cache-Management Instructions.....	890
Table E-6: Summary of Exception and Interrupt Vector Differences.....	891

Appendix F: PowerPC® Book-E Compatibility

Table F-1: Registers Not Defined in PowerPC Book-E Architecture.....	894
Table F-2: Renumbered/Renamed Registers in the PowerPC Book-E Architecture.....	894
Table F-3: New Registers in the PowerPC Book-E Architecture.....	895
Table F-4: Comparison of MSR Bit Definitions.....	895
Table F-5: Summary of Memory Translation Extensions.....	897
Table F-6: PowerPC 40x Cache-Management Instructions.....	898
Table F-7: Exceptions and Associated IVOR _n Registers.....	899
Table F-8: Comparison of ESR Bit Definitions.....	899

Volume 2(b): PPC405 Processor Block Manual

About This Book

Table 3-1: General Notational Conventions.....	909
Table 3-2: PPC405x3 Registers.....	910

Chapter 1: Introduction to the PowerPC® 405 Processor

Table 1-1: Three Levels of PowerPC Architecture	916
Table 1-2: OEA Features of the PowerPC Embedded-Environment Architecture	917
Table 1-3: PPC405 Cycles per Instruction.....	928

Chapter 2: Input/Output Interfaces

Table 2-1: Signal Name Prefix Definitions	932
Table 2-2: CPM Interface I/O Signals	934
Table 2-3: CPU Control Interface I/O Signals	937
Table 2-4: Multiply and MAC Instruction Timing.....	938
Table 2-5: Valid Reset Signal Combinations and Effect on DBSR(MRR).....	940
Table 2-6: Reset Interface I/O Signals.....	940
Table 2-7: Instruction-Side PLB Interface Signal Summary.....	946
Table 2-8: PLB-Request Priority Encoding	949
Table 2-9: Number of Transfers Required for Instruction-Fetch Requests.....	951
Table 2-10: Contents of ICU Read-Data Bus During Line Transfer.....	952
Table 2-11: Key to ISPLB Timing Diagram Abbreviations	955
Table 2-12: Data-Side PLB Interface I/O Signal Summary	968
Table 2-13: Interpretation of DCU Byte Enables During Word Transfers	972
Table 2-14: PLB-Request Priority Encoding	973
Table 2-15: Contents of DCU Write-Data Bus During Eight-Word Line Transfer.....	975
Table 2-16: Contents of DCU Read-Data Bus During Eight-Word Line Transfer	978
Table 2-17: Key to DSPLB Timing Diagram Abbreviations.....	981
Table 2-18: DCR Interface I/O Signals.....	997
Table 2-19: EIC Interface I/O Signals.....	1004
Table 2-20: PPC405 Standard Product JTAG Instruction Codes.....	1007
Table 2-21: JTAG Interface I/O Signals	1008
Table 2-22: Debug Interface I/O Signals	1011
Table 2-23: Trace Interface Signals.....	1014
Table 2-24: Purpose of C405TRCTRIGGEREVENTTYPE[0:10] Signals	1015
Table 2-25: Additional FPGA I/O Signals.....	1017

Chapter 3: PowerPC® 405 OCM Controller

Table 3-1: Block RAM Configurations in a Virtex-II FPGA	1030
Table 3-2: ISBRAM and DSBRAM Configuration Range Using the 14 Least-Significant Processor Block Address Outputs	1031

Appendix A: RISCWatch and RISCTrace Interfaces

Table A-1: JTAG Connector Signals for RISCWatch	1040
Table A-2: PPC405x3 to RISCWatch Signal Mapping	1040
Table A-3: Trace Connector Signals for RISCTrace	1041
Table A-4: PPC405x3 to RISCTrace Signal Mapping	1042
Table B-1: PPC405x3 Interface Signals in Alphabetical Order	1043

Volume 3: Rocket I/O™ Transceiver User Guide

Chapter 1: Introduction

Chapter 2: Rocket I/O™ Transceiver Overview

<i>Table 2-1: Rocket I/O Cores</i>	1059
<i>Table 2-2: Communications Standards Supported by Rocket I/O Transceiver.....</i>	1059
<i>Table 2-3: Serial Baud Rates and the SERDES_10B Attribute</i>	1060
<i>Table 2-4: Supported Rocket I/O Transceiver Primitives</i>	1061
<i>Table 2-5: Running Disparity Control.....</i>	1062
<i>Table 2-6: Loopback Options.....</i>	1064
<i>Table 2-7: Reset and Power Control Descriptions.....</i>	1068
<i>Table 2-8: Power Control Descriptions.....</i>	1068

Chapter 3: Digital Design Considerations

<i>Table 3-1: GT_CUSTOM⁽¹⁾, GT_AURORA, GT_FIBRE_CHAN⁽²⁾, GT_ETHERNET⁽²⁾, GT_INFINIBAND, and GT_XAUI Primitive Ports</i>	1069
<i>Table 3-2: Rocket I/O Transceiver Attributes.....</i>	1073
<i>Table 3-3: Default Attribute Values: GT_AURORA, GT_CUSTOM, GT_ETHERNET</i>	1078
<i>Table 3-4: Default Attribute Values: GT_FIBRE_CHAN, GT_INFINIBAND, and GT_XAUI</i>	1080
<i>Table 3-5: Control/Status Bus Association to Data Bus Byte Paths.....</i>	1082
<i>Table 3-6: Clock Ports.....</i>	1082
<i>Table 3-7: Data Width Clock Ratios</i>	1083
<i>Table 3-8: DCM Outputs for Different DATA_WIDTHs</i>	1083
<i>Table 3-9: Parameters Relative to the RX User Clock (RXUSRCLK)</i>	1094
<i>Table 3-10: Parameters Relative to the RX User Clock2 (RXUSRCLK2)</i>	1094
<i>Table 3-11: Parameters Relative to the TX User Clock2 (TXUSRCLK2).....</i>	1095
<i>Table 3-12: Miscellaneous Clock Parameters</i>	1096
<i>Table 3-13: Clock Correction Sequence / Data Correlation for 16-Bit Data Port.....</i>	1099
<i>Table 3-14: RXCLKCORCNT Definition</i>	1099
<i>Table 3-15: 8B/10B Bypassed Signal Significance</i>	1101
<i>Table 3-16: Running Disparity Modes with 8B/10B Enabled.....</i>	1101
<i>Table 3-17: Valid Data Characters</i>	1103
<i>Table 3-18: Valid Control “K” Characters</i>	1112
<i>Table 3-19: Effects of CRC on Transceiver Latency</i>	1114
<i>Table 3-20: Global and Local Headers</i>	1116
<i>Table 3-21: Bonded Channel Connections.....</i>	1116
<i>Table 3-22: Master/Slave Channel Bonding Attribute Settings.....</i>	1117

Chapter 4: Analog Design Considerations

<i>Table 4-1: Differential Transmitter Parameters</i>	1120
<i>Table 4-2: Pre-emphasis Values</i>	1120

<i>Table 4-3: Differential Receiver Parameters</i>	1123
<i>Table 4-4: CDR Parameters</i>	1124
<i>Table 4-5: Transceiver Power Supplies</i>	1125

Chapter 5: Simulation and Implementation

<i>Table 5-1: LOOPBACK Modes</i>	1133

Appendix A: Rocket I/O™ Cell Models

Volume 4: Design Flow

Chapter 1: Introduction to Virtex-II Pro™ Design Flow

Chapter 2: Design Flow Overview

<i>Table 2-1: Compiler Flags</i>	1161
<i>Table 2-1: Primary Make Targets</i>	1165
<i>Table 2-2: Configuration Variables</i>	1176
<i>Table 2-3: File Types</i>	1179
<i>Table 2-4: Boolean Operators</i>	1180
<i>Table 2-5: File Type versus Make Target for Shorthand Notation</i>	1180
<i>Table 2-6: Design Flow Tool Support</i>	1185

Chapter 3: PPC405 Bus Functional Model Usage

Chapter 4: SmartModel® Usage

<i>Table 4-1: Models for Simulating Xilinx Library Elements</i>	1193
<i>Table 4-2: PPC405 SmartModel Register Windows</i>	1195
<i>Table 4-3: Files in the \$V2PRO/platforms/embedded/sim/testbench/verilog/ Directory</i>	1196

Volume 5: Software Development Tools

Chapter 1: Introduction to Software Development Tools

Chapter 2: Using Data2BRAM

Chapter 3: GNU Software Tool Chain

<i>Table 3-1: Online GNU Software Documentation</i>	1234
<i>Table 3-2: Software Terminology</i>	1235
<i>Table 3-3: Compiler Flags</i>	1237

Volume 6: Reference Systems

Chapter 1: Introduction to Reference Systems

Chapter 2: Simple PPC405 Reference System

<i>Table 2-1: Files in the \$V2PRO/platforms/simple Directory.....</i>	1257
<i>Table 2-2: Files in the \$V2PRO/platforms/simple/sys Directory.....</i>	1257
<i>Table 2-3: Files in the \$V2PRO/platforms/simple/sys/verilog Directory.....</i>	1258
<i>Table 2-4: Files in the \$V2PRO/platforms/simple/sim/func_sim Directory.....</i>	1258
<i>Table 2-5: Files in the \$V2PRO/platforms/simple/sim/bfl Directory</i>	1258
<i>Table 2-6: Files in the \$V2PRO/platforms/simple/sim/ba_sim Directory.....</i>	1258
<i>Table 2-7: Files in the \$V2PRO/platforms/simple/sim/testbench/verilog Directory ...</i>	1259
<i>Table 2-8: Files in the \$V2PRO/platforms/simple/par Directory.....</i>	1259
<i>Table 2-9: Files in the \$V2PRO/platforms/simple/syn Directory</i>	1259
<i>Table 2-10: PLB Device Memory Map</i>	1263

Chapter 3: PPC405 PPE Reference System Using Rocket I/O™ Transceivers

<i>Table 3-1: DSB RAM Buffer Allocation</i>	1269
<i>Table 3-2: Disabling the External Interrupt Interface.....</i>	1269
<i>Table 3-3: OCM DCR Register Summary</i>	1270
<i>Table 3-4: Adjustment of OCM Multicycle Mode Clock Ratio</i>	1271
<i>Table 3-5: Ordered Set Primitives.....</i>	1273
<i>Table 3-6: Values of Certain Registers Out of Reset.....</i>	1276
<i>Table 3-7: Application Firmware Modules</i>	1278
<i>Table 3-8: Files in the \$V2PRO/platforms/pkt_proc Directory.....</i>	1282
<i>Table 3-9: Files in the \$V2PRO/platforms/pkt_proc/sys Directory</i>	1283
<i>Table 3-10: Files in the \$V2PRO/platforms/pkt_proc/sys/verilog Directory</i>	1283
<i>Table 3-11: Files in the \$V2PRO/platforms/pkt_proc/sim/func_sim Directory</i>	1283
<i>Table 3-12: Files in the \$V2PRO/platforms/pkt_proc/sim/bfl Directory.....</i>	1283
<i>Table 3-13: Files in the \$V2PRO/platforms/pkt_proc/sim/ba_sim Directory.....</i>	1283
<i>Table 3-14: Files in the \$V2PRO/platforms/pkt_proc/sim/testbench/verilog Directory</i>	1284
<i>Table 3-15: Files in the \$V2PRO/platforms/pkt_proc/par Directory.....</i>	1284
<i>Table 3-16: Files in the \$V2PRO/platforms/pkt_proc/syn Directory</i>	1284
<i>Table 3-17: Files in the \$V2PRO/source/hw/verilog/pkt_proc Directory</i>	1284
<i>Table 3-18: Processor Memory Map</i>	1288

Chapter 4: Embedded PPC405 Reference System

<i>Table 4-1: Connections from the IP to the Interrupt Controllers</i>	1295
<i>Table 4-2: Files in the \$V2PRO/platforms/embedded Directory.....</i>	1302
<i>Table 4-3: Files in the \$V2PRO/platforms/embedded/sys Directory.....</i>	1302
<i>Table 4-4: Files in the \$V2PRO/platforms/embedded/sys/verilog Directory.....</i>	1302

<i>Table 4-5: Files in the \$V2PRO/platforms/embedded/sys/mem Directory.....</i>	1303
<i>Table 4-6: Files in the \$V2PRO/platforms/embedded/sim/func_sim Directory</i>	1303
<i>Table 4-7: Files in the \$V2PRO/platforms/embedded/sim/bfl Directory</i>	1303
<i>Table 4-8: Files in the \$V2PRO/platforms/embedded/sim/ba_sim Directory.....</i>	1304
<i>Table 4-9: Files in the \$V2PRO/platforms/embedded/sim/testbench/verilog Directory.....</i>	1304
<i>Table 4-10: Files in the \$V2PRO/platforms/embedded/par Directory.....</i>	1304
<i>Table 4-11: Files in the \$V2PRO/platforms/embedded/syn Directory</i>	1304
<i>Table 4-12: SWIFT Simulation Programs.....</i>	1306
<i>Table 4-13: Memory Maps.....</i>	1310

Volume 7: Hardware IP Specifications

Chapter 1: Introduction to Hardware IP

<i>Table 1-1: Actual Logic Utilization from Embedded Reference System.....</i>	1320

Chapter 2: Using IPIF to Build IP

Chapter 3: IPIF Specification

<i>Table 3-1: IPIF Slave DMA Handshake Signals Connecting to IP</i>	1342
<i>Table 3-2: IPIF Slave DMA Handshake Module Parameters.....</i>	1343
<i>Table 3-3: Signals for the IPIF Slave Control Register Module.....</i>	1349
<i>Table 3-4: IPIF Slave Control Register Parameters</i>	1350
<i>Table 3-5: Signals for the IPIF Slave SRAM Module</i>	1358
<i>Table 3-6: Parameters for the IPIF Slave SRAM Module.....</i>	1359
<i>Table 3-7: Signals for the IPIF Slave FIFO Module</i>	1365
<i>Table 3-8: Parameters for the IPIF Slave FIFO Module.....</i>	1367
<i>Table 3-9: Signals for the IPIF Master Module</i>	1376
<i>Table 3-10: Parameters for the IPIF Master Module</i>	1377
<i>Table 3-11: Complete Parameters for All IPIF Modules</i>	1378
<i>Table 3-12: IPIF Signals Connecting to IP for All Modules</i>	1381

Chapter 4: OPB Arbiter

<i>Table 4-1: OPB Arbiter I/O Signals</i>	1387
<i>Table 4-2: OPB Arbiter Registers</i>	1388
<i>Table 4-3: OPB Arbiter Priority Register Bit Definitions</i>	1389
<i>Table 4-4: OPB Arbiter Control Register Bit Definitions</i>	1390

Chapter 5: OPB Configuration ROM Module

<i>Table 5-1: Global Signals</i>	1401
<i>Table 5-2: OPB Slave Signals.....</i>	1402
<i>Table 5-3: Parameters</i>	1402

Chapter 6: OPB GPIO Module

Table 6-1: Global Signals	1405
Table 6-2: OPB Slave Signals.....	1406
Table 6-3: External I/O Pins	1406
Table 6-4: Parameters	1406
Table 6-5: Register Definitions	1409

Chapter 7: OPB LCD Controller

Table 7-1: Global Signals	1411
Table 7-2: OPB Slave Signals.....	1411
Table 7-3: External I/O Pins	1412
Table 7-4: Parameters	1412
Table 7-5: Register Definitions	1415

Chapter 8: OPB 16450 UART Controller

Table 8-1: UART I/O Signals	1418
Table 8-2: UART Registers	1420
Table 8-3: Receiver Buffer Register Bit Definitions.....	1420
Table 8-4: Transmitter Holding Register Bit Definitions	1421
Table 8-5: Interrupt Enable Register Bit Definitions	1421
Table 8-6: Interrupt Identification Register Bit Definitions	1422
Table 8-7: Line Control Register Bit Definitions	1422
Table 8-8: Modem Control Register Bit Definitions.....	1423
Table 8-9: Line Status Register Bit Definitions.....	1424
Table 8-10: Modem Status Register Bit Definitions.....	1424
Table 8-11: Scratch Register Bit Definitions.....	1425
Table 8-12: Divisor (Least Significant Byte) Register Bit Definitions.....	1425
Table 8-13: Divisor (Most Significant Byte) Register Bit Definitions.....	1425

Chapter 9: OPB 16550 UART Controller

Table 9-1: UART I/O Signals	1428
Table 9-2: UART Registers	1430
Table 9-3: Receiver Buffer Register Bit Definitions.....	1430
Table 9-4: Transmitter Holding Register Bit Definitions	1431
Table 9-5: Interrupt Enable Register Bit Definitions	1431
Table 9-6: Interrupt Identification Register Bit Definitions	1432
Table 9-7: FIFO Control Register Bit Definitions.....	1432
Table 9-8: Line Control Register Bit Definitions	1433
Table 9-9: Modem Control Register Bit Definitions.....	1434
Table 9-10: Line Status Register Bit Definitions.....	1434
Table 9-11: Modem Status Register Bit Definitions.....	1435
Table 9-12: Scratch Register Bit Definitions.....	1436
Table 9-13: Divisor (Least Significant Byte) Register Bit Definitions.....	1436
Table 9-14: Divisor (Most Significant Byte) Register Bit Definitions.....	1436

Chapter 10: OPB IIC Controller

Table 10-1: IIC I/O Signals	1441
Table 10-2: IIC Registers.....	1442
Table 10-3: IIC Control Register Bit Definitions	1442
Table 10-4: Status Register Bits	1444
Table 10-5: IIC Transmit FIFO.....	1445
Table 10-6: IIC Receive FIFO	1445
Table 10-7: Address Register Bits	1445
Table 10-8: Transmit FIFO Occupancy Register Bits	1445
Table 10-9: Receive FIFO Occupancy Register Bits.....	1446
Table 10-10: Receive FIFO Programmable Depth Interrupt Register Bits.....	1446

Chapter 11: OPB to DCR Bridge

Table 11-1: Global Signals	1451
Table 11-2: OPB Slave Signals.....	1452
Table 11-3: DCR Master Signals	1452
Table 11-4: Parameters	1452

Chapter 12: OPB to PCI Bridge

Table 12-1: Global Signals	1455
Table 12-2: OPB Slave Signals.....	1456
Table 12-3: PCI Master Signals	1456
Table 12-4: Parameters	1457
Table 12-5: OPB to PCI Bridge Memory Map	1459

Chapter 13: PLB Arbiter

Table 13-1: Priority Order for Bus Masters	1464
Table 13-2: PLB Arbiter Pin Description	1464
Table 13-3: M_priority[n*2:n*2+1] Request Priority Level	1471
Table 13-4: PLB Address Bus Signal Bits	1473
Table 13-5: Read Burst Size.....	1474
Table 13-6: M_msize[n:n+1] Master Size.....	1475
Table 13-7: Byte Enable Signals ⁽¹⁾	1479
Table 13-8: Byte Enable Signals During Burst Transfers.....	1480
Table 13-9: PLB Transfer Size Signals	1482
Table 13-10: PLB Transfer Type Signals.....	1483
Table 13-11: PLB Read Word Address Signals.....	1489
Table 13-12: PLB Arbiter Registers.....	1493
Table 13-13: PLB Arbiter Control Register (PACR).....	1494
Table 13-14: PLB Error Status Register (PESR)	1495

Chapter 14: PLB BRAM Controller

Table 14-1: Global Signals	1510
Table 14-2: Transfer Qualifier Signals.....	1510
Table 14-3: Address Pipelining Signals.....	1511

<i>Table 14-4: Write Data Bus Signals</i>	1511
<i>Table 14-5: Read Data Bus Signals</i>	1511
<i>Table 14-6: BRAM Interface Signals</i>	1511
<i>Table 14-7: Parameters</i>	1512

Chapter 15: PLB UART Controller

<i>Table 15-1: Global Signals</i>	1515
<i>Table 15-2: Transfer Qualifier Signals.....</i>	1516
<i>Table 15-3: Address Pipelining Signals.....</i>	1516
<i>Table 15-4: Write Data Bus Signals</i>	1517
<i>Table 15-5: Read Data Bus Signals</i>	1517
<i>Table 15-6: UART Interface Pin Description.....</i>	1517
<i>Table 15-7: Parameters</i>	1518

Chapter 16: PLB SRAM/Flash Controller

<i>Table 16-1: SRAM Controller Signals.....</i>	1522
<i>Table 16-2: SRAM/Flash Controller Parameters</i>	1524

Chapter 17: PLB ZBT SRAM Controller

<i>Table 17-1: PLB Interface Pin Description</i>	1534
<i>Table 17-2: ZBT SRAM Interface Pin Description</i>	1535

Chapter 18: PLB DDR SDRAM Controller

<i>Table 18-1: PLB Interface Pin Description</i>	1544
<i>Table 18-2: DDR SDRAM Interface Pin Description</i>	1545
<i>Table 18-3: Other System Signals Pin Description</i>	1545

Chapter 19: PLB to OPB Bridge-Out Module

<i>Table 19-1: Global Signals</i>	1554
<i>Table 19-2: Transfer Qualifier Signals.....</i>	1554
<i>Table 19-3: Address Pipelining Signals.....</i>	1555
<i>Table 19-4: Write Data Bus Signals</i>	1555
<i>Table 19-5: Read Data Bus Signals</i>	1555
<i>Table 19-6: OPB Master Interface Signals.....</i>	1555
<i>Table 19-7: DCR Slave Interface Signals.....</i>	1556
<i>Table 19-8: Error Status Signal</i>	1556
<i>Table 19-9: Parameters</i>	1556
<i>Table 19-10: Bus Error Address Registers (BEAR) and Bus Error Status Registers (BESR)</i>	1562

Chapter 20: Interrupt Controller

<i>Table 20-1: Interrupt Vector Register Values</i>	1577

Chapter 21: Packet Processing Engine Controller

<i>Table 21-1: PPE Interface Signals</i>	1584

<i>Table 21-2: Rocket I/O Transceiver Interface Signals.....</i>	1585
<i>Table 21-3: GT_CUSTOM Primitive Attributes Used in Packet Processing Engine...</i>	1586
<i>Table 21-4: Ordered Set Primitives.....</i>	1589
<i>Table 21-5: Packet Processing Engine DCR Control Registers</i>	1590
<i>Table 21-6: PPC405 DCR Interface Signals</i>	1592

Volume 8: Software IP and Applications

Chapter 1: Introduction to Software IP and Applications

Chapter 2: Stand-Alone Board Support Package

<i>Table 2-1: BSP Directory Structure.....</i>	1597
<i>Table 2-2: Exception Handler Parameters</i>	1603
<i>Table 2-3: Registered Exception Types and Values</i>	1603
<i>Table 2-4: Possible Values of val for Enabling Interrupts</i>	1604
<i>Table 2-5: iic_Read() Parameters.....</i>	1607
<i>Table 2-6: iic_Write() Parameters</i>	1607
<i>Table 2-7: iicat24cxx_Read() Parameters.....</i>	1608
<i>Table 2-8: iicat24cxx_Write() Parameters.....</i>	1608
<i>Table 2-9: pci_map() Function</i>	1614
<i>Table 2-10: pci_read_config_reg() Function.....</i>	1614
<i>Table 2-11: mltime_compare() Function</i>	1619
<i>Table 2-12: Bitmask Parameter Values</i>	1620
<i>Table 2-13: Parameters for Initializing the UART at REGBASE</i>	1622

Chapter 3: PPC405 Software: Demonstration and Test Applications

<i>Table 3-1: Hardware Used with Various Demo and Test Applications.....</i>	1626
<i>Table 3-2: Software Used with Various Demo and Test Applications.....</i>	1627
<i>Table 3-3: Reference Designs Usable with the flasher Application</i>	1628
<i>Table 3-4: Files Used in the flasher Application.....</i>	1628
<i>Table 3-5: V2PDK Possible Targets.....</i>	1629
<i>Table 3-6: Optionally Added Definitions to CFLAGS</i>	1629
<i>Table 3-7: Required Libraries</i>	1629
<i>Table 3-8: Reference Designs Usable with the hello Application.....</i>	1630
<i>Table 3-9: Files Used in the hello Application</i>	1630
<i>Table 3-10: V2PDK Possible Targets.....</i>	1631
<i>Table 3-11: Required Libraries</i>	1631
<i>Table 3-12: Reference Designs Usable with the hello_gpio Application</i>	1632
<i>Table 3-13: Files Used in the hello_gpio Application.....</i>	1632
<i>Table 3-14: V2PDK Possible Targets.....</i>	1632
<i>Table 3-15: Optionally Added Definitions to CFLAGS</i>	1633
<i>Table 3-16: Required Libraries</i>	1633

<i>Table 3-17: Reference Designs Usable with the hello_iic_at24cxx Application</i>	<i>1634</i>
<i>Table 3-18: Files Used in the hello_iic_at24cxx Application</i>	<i>1634</i>
<i>Table 3-19: V2PDK Possible Targets.....</i>	<i>1635</i>
<i>Table 3-20: Required Libraries</i>	<i>1635</i>
<i>Table 3-21: Reference Designs Usable with the hello_lcd Application</i>	<i>1636</i>
<i>Table 3-22: Files Used in the hello_lcd Application</i>	<i>1636</i>
<i>Table 3-23: V2PDK Possible Targets.....</i>	<i>1636</i>
<i>Table 3-24: Optionally Added Definitions to CFLAGS</i>	<i>1637</i>
<i>Table 3-25: Required Libraries</i>	<i>1637</i>
<i>Table 3-26: Reference Designs Usable with the hello_pit Application</i>	<i>1638</i>
<i>Table 3-27: Files Used in the hello_pit Application.....</i>	<i>1638</i>
<i>Table 3-28: V2PDK Possible Targets.....</i>	<i>1638</i>
<i>Table 3-29: Optionally Added Definitions to CFLAGS</i>	<i>1639</i>
<i>Table 3-30: Required Libraries</i>	<i>1639</i>
<i>Table 3-31: Reference Designs Usable with the hello_uart Application</i>	<i>1640</i>
<i>Table 3-32: Files Used in the hello_uart Application.....</i>	<i>1640</i>
<i>Table 3-33: V2PDK Possible Targets.....</i>	<i>1641</i>
<i>Table 3-34: Required Libraries</i>	<i>1641</i>
<i>Table 3-35: Reference Designs Usable with the scan_pci Application</i>	<i>1642</i>
<i>Table 3-36: Files Used in the scan_pci Application.....</i>	<i>1642</i>
<i>Table 3-37: V2PDK Possible Targets.....</i>	<i>1643</i>
<i>Table 3-38: Required Libraries</i>	<i>1643</i>
<i>Table 3-39: Reference Designs Usable with the xrom Application.....</i>	<i>1644</i>
<i>Table 3-40: Main Menu</i>	<i>1644</i>
<i>Table 3-41: Main.Tests Menu</i>	<i>1644</i>
<i>Table 3-42: Main.Cache Settings Menu</i>	<i>1645</i>
<i>Table 3-43: Single Byte, Halfword, and Word Accesses.....</i>	<i>1645</i>
<i>Table 3-44: Four-Word Transactions</i>	<i>1647</i>
<i>Table 3-45: Eight-Word Transactions</i>	<i>1648</i>
<i>Table 3-46: Combinations.....</i>	<i>1649</i>
<i>Table 3-47: Access Pattern for Four-Word Transactions with Byte Read or Write Operations.....</i>	<i>1650</i>
<i>Table 3-48: LCD Test Steps</i>	<i>1651</i>
<i>Table 3-49: LED Test Steps.....</i>	<i>1651</i>
<i>Table 3-50: IIC Test Steps.....</i>	<i>1652</i>
<i>Table 3-51: Files Used by the xrom Application</i>	<i>1652</i>
<i>Table 3-52: V2PDK Possible Targets.....</i>	<i>1653</i>
<i>Table 3-53: Optionally Added Definitions to CFLAGS</i>	<i>1653</i>
<i>Table 3-54: Required Libraries</i>	<i>1653</i>

Chapter 4: Simulation Terminal

<i>Table 4-1: Simulation File Names.....</i>	<i>1656</i>

Introduction to Documentation

Welcome to the *Virtex-II Pro™ Developer's Kit*! This kit is designed to provide you with the information required to build systems using the Virtex-II Pro family. We have strived to provide adequate content to enable you to quickly learn about the tremendous features this product offers. Virtex-II Pro has new and unique features, such as the IBM® PowerPC® CPU and Rocket I/O™ Multi-Gigabit Transceivers which will enable greater flexibility and life to your products.

This document attempts to provide some context to the wealth of information provided with the Virtex-II Pro Developer's Kit. For example, the PowerPC is a complex RISC processor and requires a bit of documentation to fully understand. Similarly, the Rocket I/O Multi-Gigabit Transceivers require a fair amount of information in order to be used effectively. Thus the goal of the remainder of this introduction is to permit easy access to the information contained in the Developer's Kit.

Developer's Kit Documentation Organization

The Virtex-II Pro Developer's Kit contains a large volume of material. This material has been organized into eight volumes. These eight volumes are:

Volume 1, Advance Product Specification

Volume 2, PowerPC® 405 Processor

Volume 3, Rocket I/O™ Transceiver User Guide

Volume 4, Design Flow

Volume 5, Software Development Tools

Volume 6, Reference Systems

Volume 7, Hardware IP Specifications

Volume 8, Software IP and Applications

Volumes 1-3 relate to aspects of the Virtex-II Pro device. Volume 1 contains the basic product specification of the device, while Volume 2 focuses on the PowerPC CPU contained within the device. Volume 3 contains additional design information on the Rocket I/O Multi-Gigabit Transceiver.

Volume 4 is intended to introduce the reader to the software and hardware flows used throughout each of the reference systems. It is also an introduction to bus functional model simulation.

Volume 5 is designed to provide important information on how the software development tools operate. The current release of the Developer's Kit provides the GNU toolset as a standard part of the install. With these tools, customers can expect to write C, C++ and PowerPC assembly code. This volume will help act as an introduction to this process. It will not, however, teach you how to write programs.

Volumes 6, 7, and 8 go hand in hand with one another to show how to use the Virtex-II Pro family of devices. Volume 6 covers the three reference systems. It details the inner working of each of these systems, and provides tutorials on how to run these designs through each step. The three reference systems are built from the hardware and software IP that is provided in the kit. The two remaining volumes document each of these individual elements. Volume 7 documents each of the hardware IPs which come with the kit, as well as the IP InterFace (IPIF). The IPIF is a means for customers to quickly build new IP, or to hook up existent IP. Volume 8 documents the software IP and applications which are used as part of the reference systems. It contains complete documentation on the APIs for the stand-alone Board Support Package, libraries, and applications shipped with the kit.

Getting Started

Using the Reference Systems

The major theme of the Virtex-II Pro Developer's Kit is to provide customers with several known working examples, and then to allow them to modify those examples. The three Reference Systems provided in this kit fill this need. The Simple Reference System contains a very simple bus interface, memory and two UARTs. It also provides an introduction to execution of PowerPC software (in simulation) of a 'hello world' program. The Packet Processing Reference System contains a simple system of three PPC405 CPUs and three Rocket I/O MGTs. The software provided with this Reference System illustrates the basic principals of processing incoming MGT data packets. Finally, the Developer's Kit provides the Embedded Reference System. This system is designed to illustrate a typical embedded system and the associated hardware and software IP.

Volume 6 provides the documentation for each of these Reference Systems. We would recommend considering the systems as tutorials. First begin with the Simple Reference System, then move to the Packet Processing system, and complete with the Embedded Reference System. Once each system has been completely run through the tool flow, the Embedded Reference System can provide the start for a customer specific Virtex-II Pro design. Directions for adding and subtracting from the design are contained in the Embedded Reference System chapter within Volume 6.

Learning About the Virtex-II Pro Device

Volume one presents the Advance Product Specification for the Virtex-II Pro devices. This document summarizes the various features in the devices, including their I/O capability, device capacity, package, etc. This information serves as a broad introduction to the many features which have been packed into the Virtex-II Pro devices.

Learning About the PowerPC 405 CPU

If one wishes to learn about the PowerPC, and more specifically about the PPC405, we recommend reading Volume 2. This Volume provides the PPC405 User's Manual as well as the PPC405 Processor Block Manual. The User's Manual teaches about how the CPU operates, its registers, instruction set, CPU architecture, caches, memory management, etc. In contrast the Processor Block Manual teaches about the interface of the CPU to the FPGA, and how they operate. It may also be necessary to refer to the CoreConnect documentation that it listed below for additional information.

Learning About the Rocket I/O Multi-Gigabit Transceiver

The Rocket I/O Multi-Gigabit Transceiver, or MGT, is a complex piece of logic built into the Virtex-II Pro devices to provide high bandwidth communications. Every Virtex-II Pro device has at least four MGTs, with more provided as the devices increase in size. The MGTs are capable of 3.125 Gbit/Sec communications rates, and include many features.

These features are documented in Volume 1 as part of the Virtex-II Pro Advance Product Specification. In addition, the Developer's Kit provides a Rocket I/O User Guide in Volume 3 to assist in understanding more of the usage details of the MGT.

Learning About the Virtex-II Pro Developer's Kit Design Flow

The Virtex-II Pro Developer's Kits introduces a simple automation means to ease the efforts of managing complex system designs. This "Design Flow" allows a customer to complete functional and back-annotated timing simulation, synthesis, implementation, bit generation, and software compilation using the standard Makefile facility. Volume 4 documents this flow, and provides details that can be used to modify one of the Reference Systems. This volume also documents how to use the Bus Functional Model for the PPC405 Processor Block.

Learning About the Software Development Tools

Volume 5 introduces the customer to the GNU Software Development environment. This volume is intended as a very short course in the GNU environment, and to document additional tools which are used as part of the design flow. It is beyond the scope of the Developer's Kit to teach programming skills, and therefore it is assumed that the reader is familiar with the typical software development process.

Learning About the Hardware IP

The Virtex-II Pro Developer's Kit provides many examples of hardware IP. Xilinx will provide a large number of common functions that customers require in order to build systems. Volume 6 documents those pieces of hardware IP which are included in the Virtex-II Pro Developer's Kit. One very important hardware IP that is also documented is the IP InterFace, or IPIF. The IPIF is designed to greatly accelerate time to market by turning the complex bus interactions into a simple interface. This interface can be used to hook up other hardware IP that customers have, or even to develop new IP. Each of the Reference Systems uses examples of the IPIF to illustrate how it can be used in customer systems.

Learning About the Software IP and Applications

Volume 8 is packed with useful software information. This volume documents the Stand-alone Board Support Package (BSP), libraries, and software applications which are included with the Developer's Kit. The stand-alone BSP documentation includes the requisite calling API information to enable customers to use these device drivers in their own code. Additionally, the volume illustrates the C libraries which are supported in this release of the Developer's Kit. Volume 8 also documents applications which use these other elements to perform various functions, for example 'hello world'. The volume also covers brief aspects of the targets available for build using 'make'.

Conclusion

We hope that you will find the Virtex-II Pro Developer's Kit to be of great value to your product efforts. We have tried to cover as much as possible within the limited scope of the kit. If you have suggestions for improvement, or have questions, please contact the relevant support contact listed in the letter that came with your Developer's Kit.

Further Reading

The Virtex-II Pro™ Developer's Kit contains a wealth of valuable information that will assist you in your design efforts. The documentation contained within the eight volumes is organized to assist you in quickly finding relevant materials. To obtain the most recent revision of this documentation, please see

http://support.xilinx.com/xlnx/xil_tt_product.jsp?sProduct=Virtex-II+Pro.

Documentation Provided by Xilinx

Virtex-II Pro™ Platform FPGA Handbook

Virtex-II Pro™ Platform FPGA Developer's Kit Documentation:

Volume 1, Advance Product Specification

Volume 2, PowerPC® 405 Processor

Volume 3, Rocket I/O™ Transceiver User Guide

Volume 4, Design Flow

Volume 5, Software Development Tools

Volume 6, Reference Systems

Volume 7, Hardware IP Specifications

Volume 8, Software IP and Applications

IBM® CoreConnect™ Documentation

The Virtex-II Pro Developer's Kit integrates seamlessly with the IBM CoreConnect Toolkit. This toolkit is not included with the Developer's Kit, but is required if bus functional simulation is desired. The toolkit provides a number of features which enhance design productivity and allow you to get the most from the Developer's Kit. To obtain the toolkit, you must be a licensee of the IBM CoreConnect Bus Architecture. Licensing CoreConnect provides access to a wealth of documentation, Bus Functional Models, Hardware IP, and the toolkit.

Xilinx provides a Web-based licensing mechanism that allows you to obtain the CoreConnect toolkit from our website. To license CoreConnect, use an Internet browser to access http://www.xilinx.com/ipcenter/processor_central/register_coreconnect.htm. Once your request has been approved (typically within 24 hours), you will receive an e-mail granting access to a protected website. You may then download the toolkit.

If you prefer, you can also license CoreConnect directly from IBM.

If you would like further information on CoreConnect Bus Architecture, please see IBM's CoreConnect website at <http://www.ibm.com/chips/products/coreconnect>.

Once you have licensed the CoreConnect toolkit, and installed it with the Developer's Kit, the following documents will be available to you in the following locations:

IBM CoreConnect Bus Architecture Specifications

IBM CoreConnect Processor Local Bus (PLB) Architecture Specification
see [\\$CORECONNECT/published/corecon/64bitPlbBus.pdf](#)

IBM CoreConnect On-chip Peripheral Bus (OPB) Architecture Specification
see [\\$CORECONNECT/published/corecon/OpbBus.pdf](#)

IBM CoreConnect Device Control Register (DCR) Bus Architecture Specification
see [\\$CORECONNECT/published/corecon/DcrBus.pdf](#)

IBM CoreConnect Toolkit Documentation

PLB Bus Functional Model Toolkit - User's Manual
see [\\$CORECONNECT/published/corecon/PlbToolkit.pdf](#)

OPB Bus Functional Model Toolkit - User's Manual
see \$CORECONNECT/published/corecon/OpbToolkit.pdf

DCR Bus Functional Model Toolkit - User's Manual
see \$CORECONNECT/published/corecon/DcrToolkit.pdf

CoreConnect Test Generator - User's Manual
see \$CORECONNECT/published/corecon/ctg.pdf

Note: \$CORECONNECT is an environment variable that is created when installing the Developer's Kit or CoreConnect Toolkit.

Software Development Documentation

There are many sources of documentation available for those who wish to learn more about Software Development. It is recommended that a web search be conducted using a favorite search engine for keywords such as "PowerPC+Software+Development". Alternatively, a technical bookstore should be able to provide many valuable resources. The books listed below are a very small fraction of those available.

Books About Programming in C

There are many good books about the C and C++ programming languages. A few of these are listed below:

Books about C

Kernigham Brian, Ritchie Dennis. 1988. *The C Programming Language*. Prentice Hall

Wang Paul. 1992. *An Introduction to ANSI C on Unix*. Wadsworth.

Kumar Ram, Agrawal Rakesh. 1992. *Programming in ANSI C*. West Publishing Company.

Books about C++

Stroustrup Bjarne. 1991. *The C++ Programming Language*. Addison-Wesley.

Lafore Robert. 1995. *Object-Oriented Programming in C++*. Waite Group Press.

Online Documents About Programming in C

The Internet offers plenty of documentation about how to program in C and C++. One of the best approaches to finding documentation online is to use a search engine (such as Google, <http://www.google.com>) and search on "introduction to programming in C."

Volume 1: Advance Product Specification

Virtex-II Pro™ Platform FPGA Developer's Kit

March 2002 Release





The Xilinx logo shown above is a registered trademark of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

"Xilinx" and the Xilinx logo are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, XC5210 are registered Trademarks of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Benchner, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Benchner, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, Nano-Blaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, Rocket I/O, SelectI/O, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM IBM Logo PowerPC PowerPC Logo Blue Logic CoreConnect CodePack

All other trademarks are the property of their respective owners.

Xilinx does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 2002 Xilinx, Inc. All Rights Reserved.

Summary of Virtex-II Pro Features

- High-performance Platform FPGA solution including
 - Up to sixteen Rocket I/O™ embedded multi-gigabit transceiver blocks (based on Mindspeed's SkyRail™ technology)
 - Up to four IBM® PowerPC® RISC processor blocks
- Based on Virtex™-II Platform FPGA technology
 - Flexible logic resources
 - SRAM-based in-system configuration
 - Active Interconnect™ technology
 - SelectRAM™ memory hierarchy
 - Dedicated 18-bit x 18-bit multiplier blocks
 - High-performance clock management circuitry
 - SelectI/O™-Ultra technology
 - Digitally Controlled Impedance (DCI) I/O

The members and resources of the Virtex-II Pro family are shown in [Table 1](#).

Rocket I/O Features

- Full-duplex serial transceiver (SERDES) capable of baud rates from 622 Mb/s to 3.125 Gb/s
- 80 Gb/s duplex data rate (16 channels)
- Monolithic clock synthesis and clock recovery (CDR)
- Fibre Channel, Gigabit Ethernet, 10 Gb Attachment Unit Interface (XAUI), and Infiniband-compliant transceivers
- 8-, 16-, or 32-bit selectable internal FPGA interface

- 8B/10B encoder and decoder
- 50Ω / 75Ω on-chip selectable transmit and receive terminations
- Programmable comma detection
- Channel bonding support (two to sixteen channels)
- Rate matching via insertion/deletion characters
- Four levels of selectable pre-emphasis
- Five levels of output differential voltage
- Per-channel internal loopback modes
- 2.5V transceiver supply voltage

PowerPC RISC Core Features

- Embedded 300+ MHz Harvard architecture core
- Low power consumption: 0.9 mW/MHz
- Five-stage data path pipeline
- Hardware multiply/divide unit
- Thirty-two 32-bit general purpose registers
- 16 KB two-way set-associative instruction cache
- 16 KB two-way set-associative data cache
- Memory Management Unit (MMU)
 - 64-entry unified Translation Look-aside Buffers (TLB)
 - Variable page sizes (1 KB to 16 MB)
- Dedicated on-chip memory (OCM) interface
- Supports IBM CoreConnect™ bus architecture
- Debug and trace support
- Timer facilities

Table 1: Virtex-II Pro FPGA Family Members

Device	Rocket I/O Transceiver Blocks	PowerPC Processor Blocks	CLB (1 CLB = 4 slices = Max 128 bits)			18 X 18 Bit Multiplier Blocks	Block SelectRAM		DCMs	Max I/O Pads
			Array Row x Col	Slices	Maximum Distributed RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)		
XC2VP2	4	0	16 x 22	1,408	44	12	12	216	4	204
XC2VP4	4	1	40 x 22	3,008	94	28	28	504	4	348
XC2VP7	8	1	40 x 34	4,928	154	44	44	792	4	396
XC2VP20	8	2	56 x 46	9,280	290	88	88	1,584	8	564
XC2VP50	16	4	88 x 70	22,592	706	216	216	3,888	8	852

Virtex-II Pro Platform FPGA Technology

- SelectRAM memory hierarchy
 - Up to 4 Mb of True Dual-Port RAM in 18 Kb block SelectRAM resources
 - Up to 706 Kb of distributed SelectRAM resources
 - High-performance interfaces to external memory
- Arithmetic functions
 - Dedicated 18-bit x 18-bit multiplier blocks
 - Fast look-ahead carry logic chains
- Flexible logic resources
 - Up to 45,184 internal registers/latches with Clock Enable
 - Up to 45,184 look-up tables (LUTs) or cascadable variable (1 to 16 bits) shift registers
 - Wide multiplexers and wide-input function support
 - Horizontal cascade chain and Sum-of-Products support
 - Internal 3-state busing
- High-performance clock management circuitry
 - Up to eight Digital Clock Manager (DCM) modules
 - Precise clock de-skew
 - Flexible frequency synthesis
 - High-resolution phase shifting
 - 16 global clock multiplexer buffers in all parts
- Active Interconnect technology
 - Fourth-generation segmented routing structure
 - Fast, predictable routing delay, independent of fanout
 - Deep sub-micron noise immunity benefits
- SelectI/O-Ultra technology
 - Up to 852 user I/Os
 - Twenty two single-ended standards and five differential standards
 - Programmable LVTTL and LVCMOS sink/source current (2 mA to 24 mA) per I/O
 - Digitally Controlled Impedance (DCI) I/O: on-chip termination resistors for single-ended I/O standards
 - PCI support⁽¹⁾
 - Differential signaling
 - 840 Mb/s Low-Voltage Differential Signaling I/O (LVDS) with current mode drivers
 - Bus LVDS I/O
 - HyperTransport (LDT) I/O with current driver buffers
 - Built-in DDR input and output registers
 - Proprietary high-performance SelectLink technology for communications between Xilinx devices
 - High-bandwidth data path
 - Double Data Rate (DDR) link
 - Web-based HDL generation methodology
- SRAM-based in-system configuration
 - Fast SelectMAP™ configuration
 - Triple Data Encryption Standard (DES) security option (bitstream encryption)
 - IEEE1532 support
 - Partial reconfiguration
 - Unlimited reprogrammability
 - Readback capability
- Supported by Xilinx Foundation™ and Alliance™ series development systems
 - Integrated VHDL and Verilog design flows
 - ChipScope™ Integrated Logic Analyzer
- 0.13-μm, nine-layer copper process with 90 nm high-speed transistors
- 1.5V (V_{CCINT}) core power supply, dedicated 2.5V V_{CCAUX} auxiliary and V_{CCO} I/O power supplies
- IEEE 1149.1 compatible boundary-scan logic support
- Flip-Chip and Wire-Bond Ball Grid Array (BGA) packages in standard 1.00 mm pitch
- Each device 100% factory tested

General Description

The Virtex-II Pro family is a platform FPGA for designs that are based on IP cores and customized modules. The family incorporates multi-gigabit transceivers and PowerPC CPU cores in Virtex-II Pro Series FPGA architecture. It empowers complete solutions for telecommunication, wireless, networking, video, and DSP applications.

The leading-edge 0.13μm CMOS nine-layer copper process and the Virtex-II Pro architecture are optimized for high performance designs in a wide range of densities. Combining a wide variety of flexible features and IP cores, the Virtex-II Pro family enhances programmable logic design capabilities and is a powerful alternative to mask-programmed gate arrays.

1. PCI supported in some banks only.

Architecture

Virtex-II Pro Array Overview

Virtex-II Pro devices are user-programmable gate arrays with various configurable elements and embedded cores optimized for high-density and high-performance system designs. Virtex-II Pro devices implement the following functionality:

- Embedded high-speed serial transceivers enable data bit rate up to 3.125 Gb/s per channel.
- Embedded IBM PowerPC 405 RISC CPU cores provide performance of 300+ MHz.
- SelectI/O-Ultra blocks provide the interface between package pins and the internal configurable logic. Most popular and leading-edge I/O standards are supported by the programmable IOBs.
- Configurable Logic Blocks (CLBs) provide functional elements for combinatorial and synchronous logic, including basic storage elements. BUFTs (3-state buffers) associated with each CLB element drive dedicated segmentable horizontal routing resources.
- Block SelectRAM memory modules provide large 18 Kb storage elements of True Dual-Port RAM.
- Embedded multiplier blocks are 18-bit x 18-bit dedicated multipliers.
- Digital Clock Manager (DCM) blocks provide self-calibrating, fully digital solutions for clock distribution delay compensation, clock multiplication and division, and coarse- and fine-grained clock phase shifting.

A new generation of programmable routing resources called Active Interconnect Technology interconnects all of these elements. The general routing matrix (GRM) is an array of routing switches. Each programmable element is tied to a switch matrix, allowing multiple connections to the general routing matrix. The overall programmable interconnection is hierarchical and designed to support high-speed designs.

All programmable elements, including the routing resources, are controlled by values stored in static memory cells. These values are loaded in the memory cells during configuration and can be reloaded to change the functions of the programmable elements.

Virtex-II Pro Features

This section briefly describes Virtex-II Pro features.

Rocket I/O Multi-Gigabit Transceiver Cores

The Rocket I/O Multi-Gigabit Transceiver core, based on Mindspeed's SkyRail technology, is a flexible parallel-to-serial and serial-to-parallel transceiver embedded core used for high-bandwidth interconnection between buses, backplanes, or other subsystems.

Multiple user instantiations in an FPGA are possible, providing up to 80 Gb/s of full-duplex raw data transfer. Each

channel can be operated at a maximum data transfer rate of 3.125 Gb/s.

Each Rocket I/O core implements the following functionality:

- Serializer and deserializer (SERDES)
- Monolithic clock synthesis and clock recovery (CDR)
- Fibre Channel, Gigabit Ethernet, XAUI, and Infiniband compliant transceivers
- 8-, 16-, or 32-bit selectable FPGA interface
- 8B/10B encoder and decoder with bypassing option on each channel
- Channel bonding support (two to sixteen channels)
 - Elastic buffers for inter-chip deskewing and channel-to-channel alignment
- Receiver clock recovery tolerance of up to 75 non-transitioning bits
- 50Ω / 75Ω on-chip selectable TX and RX terminations
- Programmable comma detection
- Rate matching via insertion/deletion characters
- Automatic lock-to-reference function
- Optional TX and RX data inversion
- Four levels of pre-emphasis support
- Per-channel serial and parallel transmitter-to-receiver internal loopback modes
- Cyclic Redundancy Check (CRC) support

PowerPC 405 Processor Block

The PPC405 RISC CPU can execute instructions at a sustained rate of one instruction per cycle. On-chip instruction and data cache reduce design complexity and improve system throughput.

The PPC405 features include:

- PowerPC RISC CPU
 - Implements the PowerPC User Instruction Set Architecture (UIA) and extensions for embedded applications
 - Thirty-two 32-bit general purpose registers (GPRs)
 - Static branch prediction
 - Five-stage pipeline with single-cycle execution of most instructions, including loads/stores
 - Unaligned and aligned load/store support to cache, main memory, and on-chip memory
 - Hardware multiply/divide for faster integer arithmetic (4-cycle multiply, 35-cycle divide)
 - Enhanced string and multiple-word handling
 - Big/little endian operation support
- Storage Control
 - Separate instruction and data cache units, both two-way set-associative and non-blocking
 - Eight words (32 bytes) per cache line
 - 16 KB array Instruction Cache Unit (ICU), 16 KB array Data Cache Unit (DCU)

- Operand forwarding during instruction cache line fill
- Copy-back or write-through DCU strategy
- Doubleword instruction fetch from cache improves branch latency
- Virtual mode memory management unit (MMU)
 - Translation of the 4 GB logical address space into physical addresses
 - Software control of page replacement strategy
 - Supports multiple simultaneous page sizes ranging from 1 KB to 16 MB
- OCM controllers provide dedicated interfaces between Block SelectRAM memory and processor core instruction and data paths for high-speed access
- PowerPC timer facilities
 - 64-bit time base
 - Programmable interval timer (PIT)
 - Fixed interval timer (FIT)
 - Watchdog timer (WDT)
- Debug Support
 - Internal debug mode
 - External debug mode
 - Debug Wait mode
 - Real Time Trace debug mode
 - Enhanced debug support with logical operators
 - Instruction trace and trace-back support
 - Forward or backward trace
- Two hardware interrupt levels support
- Advanced power management support

Input/Output Blocks (IOBs)

IOBs are programmable and can be categorized as follows:

- Input block with an optional single data rate (SDR) or double data rate (DDR) register
- Output block with an optional SDR or DDR register and an optional 3-state buffer to be driven directly or through an SDR or DDR register
- Bidirectional block (any combination of input and output configurations)

These registers are either edge-triggered D-type flip-flops or level-sensitive latches.

IOBs support the following single-ended I/O standards:

- LVTTTL
- LVCMOS (3.3V, 2.5V, 1.8V, and 1.5V)
- PCI (33 and 66 MHz)
- GTL and GTLP
- HSTL 1.5V and 1.8V (Class I, II, III, and IV)
- SSTL (3.3V and 2.5V, Class I and II)

The DCI I/O feature automatically provides on-chip termination for each single-ended I/O standard.

The IOB elements also support the following differential signaling I/O standards:

- LVDS and Extended LVDS (2.5V only)
- BLVDS (Bus LVDS)
- ULVDS
- LDT

Two adjacent pads are used for each differential pair. Two or four IOB blocks connect to one switch matrix to access the routing resources.

Configurable Logic Blocks (CLBs)

CLB resources include four slices and two 3-state buffers. Each slice is equivalent and contains:

- Two function generators (F & G)
- Two storage elements
- Arithmetic logic gates
- Large multiplexers
- Wide function capability
- Fast carry look-ahead chain
- Horizontal cascade chain (OR gate)

The function generators F & G are configurable as 4-input look-up tables (LUTs), as 16-bit shift registers, or as 16-bit distributed SelectRAM memory.

In addition, the two storage elements are either edge-triggered D-type flip-flops or level-sensitive latches.

Each CLB has internal fast interconnect and connects to a switch matrix to access general routing resources.

Block SelectRAM Memory

The block SelectRAM memory resources are 18 Kb of True Dual-Port RAM, programmable from 16K x 1 bit to 512 x 36 bit, in various depth and width configurations. Each port is totally synchronous and independent, offering three "read-during-write" modes. Block SelectRAM memory is cascadable to implement large embedded storage blocks. Supported memory configurations for dual-port and single-port modes are shown in [Table 2](#).

Table 2: Dual-Port and Single-Port Configurations

16K x 1 bit	4K x 4 bits	1K x 18 bits
8K x 2 bits	2K x 9 bits	512 x 36 bits

18 X 18 Bit Multipliers

A multiplier block is associated with each SelectRAM memory block. The multiplier block is a dedicated 18 x 18-bit 2s complement signed multiplier, and is optimized for operations based on the block SelectRAM content on one port. The 18 x 18 multiplier can be used independently of the block SelectRAM resource. Read/multiply/accumulate operations and DSP filter structures are extremely efficient.

Both the SelectRAM memory and the multiplier resource are connected to four switch matrices to access the general routing resources.

Global Clocking

The DCM and global clock multiplexer buffers provide a complete solution for designing high-speed clock schemes.

Up to eight DCM blocks are available. To generate deskewed internal or external clocks, each DCM can be used to eliminate clock distribution delay. The DCM also provides 90-, 180-, and 270-degree phase-shifted versions of its output clocks. Fine-grained phase shifting offers high-resolution phase adjustments in increments of $1/256$ of the clock period. Very flexible frequency synthesis provides a clock output frequency equal to a fractional or integer multiple of the input clock frequency. For exact timing parameters, see **Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics**.

Virtex-II Pro devices have 16 global clock MUX buffers, with up to eight clock nets per quadrant. Each clock MUX buffer can select one of the two clock inputs and switch glitch-free from one clock to the other. Each DCM can send up to four of its clock outputs to global clock buffers on the same edge. Any global clock pin can drive any DCM on the same edge.

Routing Resources

The IOB, CLB, block SelectRAM, multiplier, and DCM elements all use the same interconnect scheme and the same access to the global routing matrix. Timing models are shared, greatly improving the predictability of the performance of high-speed designs.

There are a total of 16 global clock lines, with eight available per quadrant. In addition, 24 vertical and horizontal long lines per row or column, as well as massive secondary and local routing resources, provide fast interconnect. Virtex-II Pro buffered interconnects are relatively unaffected by net fanout, and the interconnect layout is designed to minimize crosstalk.

Horizontal and vertical routing resources for each row or column include:

- 24 long lines
- 120 hex lines
- 40 double lines
- 16 direct connect lines (total in all four directions)

Boundary Scan

Boundary-scan instructions and associated data registers support a standard methodology for accessing and config-

uring Virtex-II Pro devices, complying with IEEE standards 1149.1 and 1532. A system mode and a test mode are implemented. In system mode, a Virtex-II Pro device will continue to function while executing non-test boundary-scan instructions. In test mode, boundary-scan test instructions control the I/O pins for testing purposes. The Virtex-II Pro Test Access Port (TAP) supports BYPASS, PRELOAD, SAMPLE, IDCODE, and USERCODE non-test instructions. The EXTEST, INTEST, and HIGHZ test instructions are also supported.

Configuration

Virtex-II Pro devices are configured by loading the bitstream into internal configuration memory using one of the following modes:

- Slave-serial mode
- Master-serial mode
- Slave SelectMAP mode
- Master SelectMAP mode
- Boundary-Scan mode (IEEE 1532)

A Data Encryption Standard (DES) decryptor is available on-chip to secure the bitstreams. One or two triple-DES key sets can be used to optionally encrypt the configuration data.

The Xilinx System Advanced Configuration Environment (System ACE) family offers high-capacity and flexible solution for FPGA configuration as well as program/data storage for the processor. See [DS080, System ACE Compact-Flash Solution](#) for more information.

Readback and Integrated Logic Analyzer

Configuration data stored in Virtex-II Pro configuration memory can be read back for verification. Along with the configuration data, the contents of all flip-flops/latches, distributed SelectRAM, and block SelectRAM memory resources can be read back. This capability is useful for real-time debugging.

The Xilinx ChipScope Integrated Logic Analyzer (ILA) cores and Integrated Bus Analyzer (IBA) cores, along with the ChipScope Pro Analyzer software, provide a complete solution for accessing and verifying user designs within Virtex-II Pro devices.

IP Core and Reference Support

Intellectual Property is part of the Platform FPGA solution. In addition to the existing FPGA fabric cores, the list below shows some of the currently available hardware and software intellectual properties specially developed for Virtex-II Pro by Xilinx. Each IP core is modular, portable, Real-Time Operating System (RTOS) independent, and CoreConnect compatible for ease of design migration. Refer to www.xilinx.com for the latest and most complete list of cores.

Hardware Cores

- Bus Infrastructure cores (arbiters, bridges, and more)

- Memory cores (Flash, SRAM, and more)
- Peripheral cores (UART, IIC, and more)
- Networking cores (ATM, Ethernet, and more)

Software Cores

- Boot code
- Test code
- Device drivers
- Protocol stacks
- RTOS integration
- Customized board support package

Virtex-II Pro Device/Package Combinations and Maximum I/Os

Offerings include ball grid array (BGA) packages with 1.0 mm pitch. In addition to traditional wire-bond interconnects, flip-chip interconnect is used in some of the BGA offerings. The use of flip-chip interconnect offers more I/Os than are possible in wire-bond versions of the similar packages. Flip-chip construction offers the combination of high pin count and excellent power dissipation.

The Virtex-II Pro device/package combination table ([Table 3](#)) details the maximum number of I/Os for each device and package using wire-bond or flip-chip technology.

- FG denotes wire-bond fine-pitch BGA (1.00 mm pitch).
- FF denotes flip-chip fine-pitch BGA (1.00 mm pitch).
- BF denotes flip-chip fine-pitch BGA (1.27 mm pitch).

Table 3: Virtex-II Pro Device/Package Combinations and Maximum Number of Available I/Os (Advance Information)

Package	Pitch (mm)	Size (mm)	User Available I/Os				
			XC2VP2	XC2VP4	XC2VP7	XC2VP20	XC2VP50
FG256	1.00	17 x 17	140	140			
FG456	1.00	23 x 23	156	248	248		
FF672	1.00	27 x 27	204	348	396		
FF896	1.00	31 x 31			396	556	
FF1152	1.00	35 x 35				564	692
FF1517	1.00	40 x 40					852
BF957	1.27	40 x 40				564	584

Virtex-II Pro Ordering Information

Virtex-II Pro ordering information is shown in Figure 1.

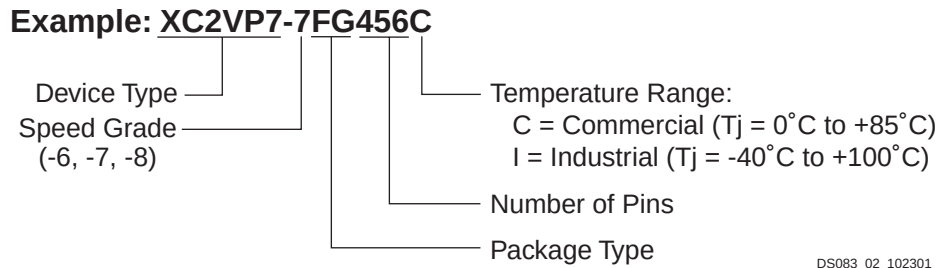


Figure 1: Virtex-II Pro Ordering Information

NOTE: Maximum serial transceiver baud rates for flipchip and wirebond packages are 3.125 Gb/s and 2.5 Gb/s respectively.

Revision History

This section records the change history for this module of the data sheet.

Date	Version	Revision
01/31/02	1.0	Initial Xilinx release.

Virtex-II Pro Data Sheet Modules

The Virtex-II Pro Data Sheet contains the following modules:

- **Virtex-II Pro Platform FPGAs: Introduction and Overview (Module 1)**
- **Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics (Module 3)**
- **Virtex-II Pro™ Platform FPGAs: Functional Description (Module 2)**
- **Virtex-II Pro Platform FPGAs: Pinout Information (Module 4)**

Virtex-II Pro Array Functional Description

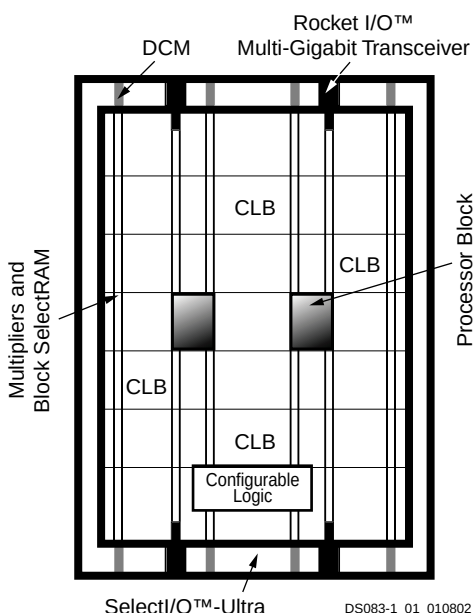


Figure 1: Virtex-II Pro Generic Architecture Overview

This module describes the following Virtex-II Pro functional components, as shown in Figure 1:

- Embedded Rocket I/O™ Multi-Gigabit Transceivers (MGTs)
- Processor Blocks containing embedded IBM® PowerPC® 405 RISC CPU (PPC405) cores and integration circuitry.

- FPGA fabric based on Virtex-II architecture.

For a detailed description of the PPC405 core programming models and internal core operations, refer to the *PowerPC 405 User Manual* and the *Processor Block Manual*.

For detailed Rocket I/O digital and analog design considerations, refer to the *Rocket I/O User Guide*.

All of the documents above, as well as a complete listing and description of Xilinx-developed Intellectual Property cores for Virtex-II Pro, are available on the Xilinx website at www.xilinx.com/virtex2pro.

Virtex-II Pro Compared to Virtex-II Devices

Virtex-II Pro is built on the Virtex-II FPGA architecture. Most FPGA features are identical to Virtex-II. The differences are described below:

- Virtex-II Pro is the first FPGA family incorporating embedded PPC405 cores and Rocket I/O MGTs.
- V_{CCAUX} , the auxiliary supply voltage, is 2.5V instead of 3.3V as for Virtex-II devices. Advanced processing at 0.13 μ m has resulted in a smaller die, faster speed, and lower power consumption.
- The Virtex-II Pro family is neither bitstream-compatible nor pin-compatible with the Virtex-II family. However, Virtex-II designs can be compiled into Virtex-II Pro devices.
- All banks support 2.5V (and below) I/O standards. 3.3V I/O standards including PCI are supported in certain banks only. (See Table 4-1, page 448.) LVPECL, LVDS_33, LVDSEXT_33, LVDCI_DV2_33, and AGP-2X are not supported.

Functional Description: Rocket I/O Multi-Gigabit Transceiver (MGT)

This section summarizes the features of the Rocket I/O multi-gigabit transceiver. For an in-depth discussion of the Rocket I/O MGT, refer to the *Rocket I/O User Guide*.

Overview

The embedded Rocket I/O multi-gigabit transceiver core is based on Mindspeed's SkyRail™ technology. Up to sixteen transceiver cores are available. The transceiver core is designed to operate at any baud rate in the range of

622 Mb/s to 3.125 Gb/s per channel. This includes specific baud rates used by various standards as listed in Table 1.

Table 1: Standards Supported by the Rocket I/O MGT

Mode	Channels (Lanes)	I/O Baud Rate (Gb/s)	Internal Clock Rate (REFCLK) (MHz)
Fibre Channel	1	1.06	53
		2.12	106
Gbit Ethernet	1	1.25	62.5
XAUI	4	3.125	156.25

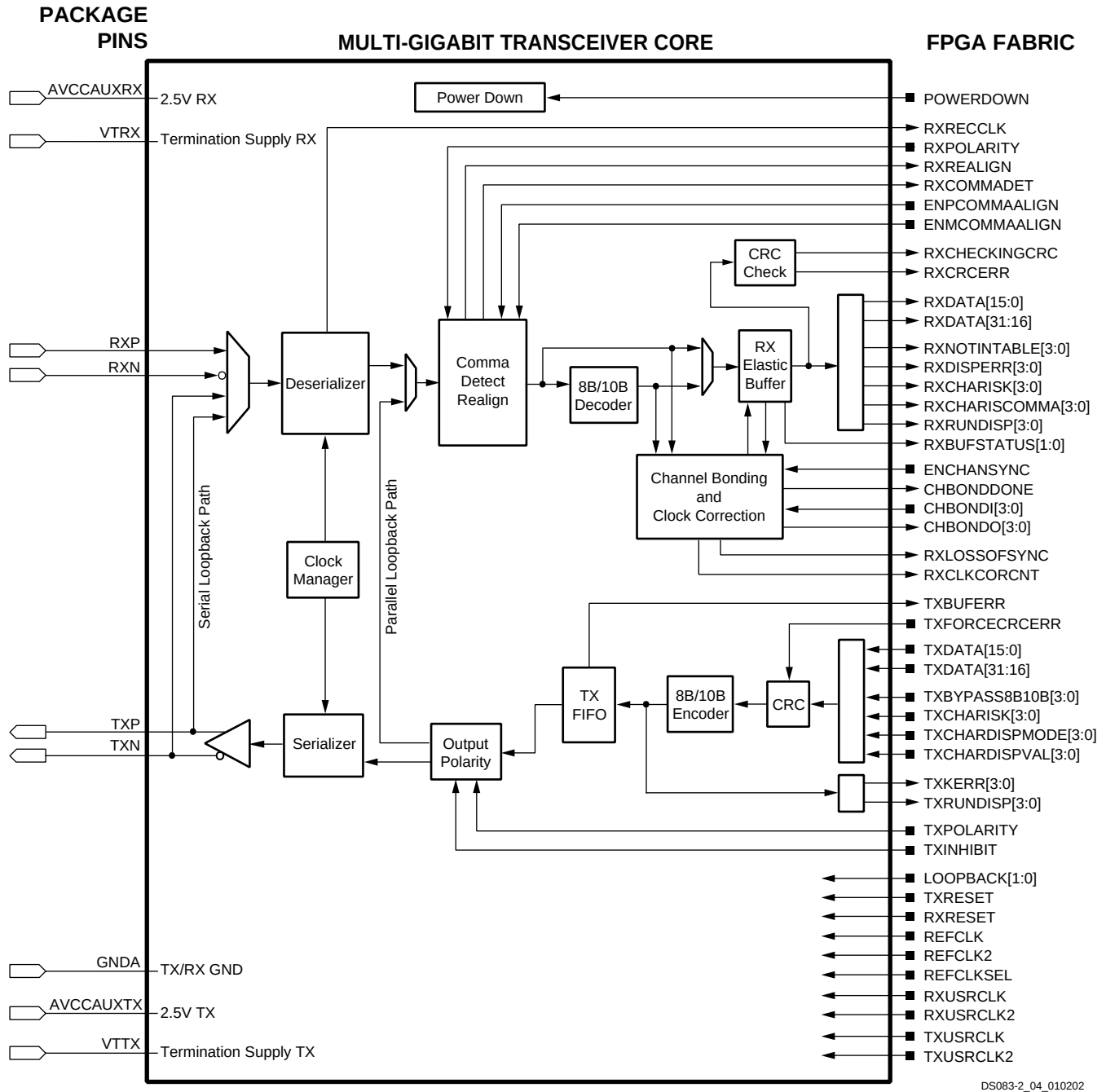
Table 1: Standards Supported by the Rocket I/O MGT

Mode	Channels (Lanes)	I/O Baud Rate (Gb/s)	Internal Clock Rate (REFCLK) (MHz)
Infiniband	1, 4, 12	2.5	125
Aurora (Xilinx)	1, 2, 3, 4, ...	0.840 - 3.125	42.00-156.25
Custom mode	1, 2, 3, 4, ...	up to 3.125	up to 156.25

The serial bit rate need not be configured in the transceiver, as the operating frequency is implied by the received data and reference clock applied.

The Rocket I/O transceiver core consists of the Physical Media Attachment (PMA) and Physical Coding Sublayer (PCS). The PMA contains the serializer and deserializer. The PCS contains the bypassable 8B/10B encoder/decoder, elastic buffers, and Cyclic Redundancy Check (CRC) units. The encoder and decoder handle the 8B/10B coding scheme. The elastic buffers support the clock correction (rate matching) and channel bonding features. The CRC units perform CRC generation and checking.

Figure 2 shows the Rocket I/O high-level block diagram and FPGA interface signals.



DS083-2_04_010202

Figure 2: Rocket I/O Block Diagram

Clock Synthesizer

Synchronous serial data reception is facilitated by a clock/data recovery circuit. This circuit uses a fully monolithic Phase Lock Loop (PLL), which does not require any external components. The clock/data recovery circuit extracts both phase and frequency from the incoming data stream. The recovered clock is presented on output RXRECCLK at 1/20 of the serial received data rate.

The gigabit transceiver multiplies the reference frequency provided on the reference clock input (REFCLK) by 20. The multiplication of the clock is achieved by using a fully monolithic PLL that does not require any external components.

No fixed phase relationship is assumed between REFCLK, RXRECCLK, and/or any other clock that is not tied to either of these clocks. When the 4-byte or 1-byte receiver data path is used, RXUSRCLK and RXUSRCLK2 have different frequencies, and each edge of the slower clock is aligned to a falling edge of the faster clock. The same relationships apply to TXUSRCLK and TXUSRCLK2.

Clock and Data Recovery

The clock/data recovery (CDR) circuits will lock to the reference clock automatically if the data is not present. For proper operation, the frequency of the reference clock must be within ± 100 ppm of the nominal frequency.

It is critical to keep power supply noise low in order to minimize common and differential noise modes into the clock/data recovery circuitry. Refer to the *Rocket I/O User Guide* for more details.

Transmitter

FPGA Transmit Interface

The FPGA can send either one, two, or four characters of data to the transmitter. Each character can be either 8 bits or 10 bits wide. If 8-bit data is applied, the additional inputs become control signals for the 8B/10B encoder. When the 8B/10B encoder is bypassed, the 10-bit character order is generated as follows:

```
TXCHARDISPMODE[0]      (first bit transmitted)
TXCHARDISPVAL[0]
TXDATA[7:0]             (last bit transmitted is TXDATA[0])
```

8B/10B Encoder

A bypassable 8B/10B encoder is included. The encoder uses the same 256 data characters and 12 control characters that are used for Gigabit Ethernet, Fibre Channel, and InfiniBand.

The encoder accepts 8 bits of data along with a K-character signal for a total of 9 bits per character applied, and generates a 10 bit character for transmission. If the K-character signal is High, the data is encoded into one of the twelve possible K-characters available in the 8B/10B code. If the K-character input is Low, the 8 bits are encoded

as standard data. If the K-character input is High, and a user applies other than one of the twelve possible combinations, TXKERR indicates the error.

Disparity Control

The 8B/10B encoder is initialized with a negative running disparity. Unique control allows forcing the current running disparity state.

TXRUNDISP signals its current running disparity. This may be useful in those cases where there is a need to manipulate the initial running disparity value.

Bits TXCHARDISPMODE and TXCHARDISPVAL control the generation of running disparity before each byte.

For example, the transceiver can generate the sequence

```
K28.5+ K28.5+ K28.5- K28.5-
or
K28.5- K28.5- K28.5+ K28.5+
```

by specifying inverted running disparity for the second and fourth bytes.

Transmit FIFO

Proper operation of the circuit is only possible if the FPGA clock (TXUSRCLK) is frequency-locked to the reference clock (REFCLK). Phase variations up to one clock cycle are allowable. The FIFO has a depth of four. Overflow or underflow conditions are detected and signaled at the interface. Bypassing of this FIFO is programmable.

Serializer

The multi-gigabit transceiver multiplies the reference frequency provided on the reference clock input (REFCLK) by 20. Clock multiplication is achieved by using a fully monolithic PLL requiring no external components. Data is converted from parallel to serial format and transmitted on the TXP and TXN differential outputs. Bit 0 is transmitted first and bit 19 is transmitted last.

The electrical connection of TXP and TXN can be interchanged through configuration. This option can be controlled by an input (TXPOLARITY) at the FPGA transmitter interface. This facilitates recovery from situations where printed circuit board traces have been reversed.

Transmit Termination

On-chip termination is provided at the transmitter, eliminating the need for external termination. Programmable options exist for 50 Ω (default) and 75 Ω termination.

Pre-Emphasis Circuit and Swing Control

Four selectable levels of pre-emphasis (10% [default], 20%, 25%, and 33%) are available. Optimizing this setting allows the transceiver to drive up to 20 inches of FR4 at the maximum baud rate.

The programmable output swing control can adjust the differential output level between 400 mV and 800 mV in four increments of 100 mV.

Receiver

Deserializer

The Rocket I/O transceiver core accepts serial differential data on its RXP and RXN inputs. The clock/data recovery circuit extracts the clock and retimes incoming data to this clock. It uses a fully monolithic PLL requiring no external components. The clock/data recovery circuitry extracts both phase and frequency from the incoming data stream. The recovered clock is presented on output RXRECCLK at 1/20 of the received serial data rate.

The receiver is capable of handling either transition-rich 8B/10B streams or scrambled streams, and can withstand a string of up to 75 non-transitioning bits without an error.

Word alignment is dependent on the state of comma detect bits. If comma detect is enabled, the transceiver will recognize up to two 10-bit preprogrammed characters. Upon detection of the character or characters, the comma detect output is driven high and the data is synchronously aligned. If a comma is detected and the data is aligned, no further alignment alteration will take place. If a comma is received and realignment is necessary, the data is realigned and an indication is given at the receiver interface. The realignment indicator is a distinct output. The transceiver will continuously monitor the data for the presence of the 10-bit character(s). Upon each occurrence of the 10-bit character, the data is checked for word alignment. If comma detect is disabled, the data will not be aligned to any particular pattern. The programmable option allows a user to align data on comma+, comma–, both, or a unique user-defined and programmed sequence.

The receiver can be configured to reverse the RXP and RXN inputs. This can be useful in the event that printed circuit board traces have been reversed.

Receiver Termination

On-chip termination is provided at the receiver, eliminating the need for external termination. The receiver includes programmable on-chip termination circuitry for 50Ω (default) or 75Ω impedance.

8B/10B Decoder

An optional 8B/10B decoder is included. A programmable option allows the decoder to be bypassed. When the 8B/10B decoder is bypassed, the 10-bit character order is, for example,

```
RXCHARISK[0]           (first bit received)
RXRUNDISP[0]
RXDATA[7:0]           (last bit received is RXDATA[0])
```

The decoder uses the same table that is used for Gigabit Ethernet, Fibre Channel, and InfiniBand. In addition to decoding all data and K-characters, the decoder has several extra features. The decoder separately detects both “disparity errors” and “out-of-band” errors. A disparity error is the reception of 10-bit character that exists within the

8B/10B table but has an incorrect disparity. An out-of-band error is the reception of a 10-bit character that does not exist within the 8B/10B table. It is possible to obtain an out-of-band error without having a disparity error. The proper disparity is always computed for both legal and illegal characters. The current running disparity is available at the RXRUNDISP signal.

The 8B/10B decoder performs a unique operation if out-of-band data is detected. If out-of-band data is detected, the decoder signals the error and passes the illegal 10-bits through and places them on the outputs. This can be used for debugging purposes if desired.

The decoder also signals the reception of one of the 12 valid K-characters. In addition, a programmable comma detect is included. The comma detect signal registers a comma on the receipt of any comma+, comma–, or both. Since the comma is defined as a 7-bit character, this includes several out-of-band characters. Another option allows the decoder to detect only the three defined commas (K28.1, K28.5, and K28.7) as comma+, comma–, or both. In total, there are six possible options, three for valid commas and three for “any comma.”

It should be noted that all bytes (1, 2, or 4) at the RX FPGA interface will each have their own individual 8B/10B indicators (K-character, disparity error, out-of-band error, current running disparity, and comma detect).

Loopback

In order to facilitate testing without having the need to either apply patterns or measure data at GHz rates, two programmable loop-back features are available.

One option, serial loopback, places the gigabit transceiver into a state where transmit data is directly fed back to the receiver. An important point to note is that the feedback path is at the output pads of the transmitter. This tests the entirety of the transmitter and receiver.

The second loopback path is a parallel path that checks the digital circuitry. When the parallel option is enabled, the serial loopback path is disabled. However, the transmitter outputs remain active and data is transmitted over a link. If TXINHIBIT is asserted, TXP is forced to 0 until TXINHIBIT is de-asserted.

Elastic and Transmitter Buffers

Both the transmitter and the receiver include buffers (FIFOs) in the datapath. This section gives the reasons for including the buffers and outlines their operation.

Receiver Buffer

The receiver buffer is required for two reasons:

- *Clock corection* to accommodate the slight difference in frequency between the recovered clock RXRECCLK and the internal FPGA user clock RXUSRCLK
- *Channel bonding* to allow realignment of the input

stream to ensure proper alignment of data being read through multiple transceivers

The receiver uses an *elastic buffer*, where "elastic" refers to the ability to modify the read pointer for clock correction and channel bonding.

Clock Correction

Clock RXRECCLK (the recovered clock) reflects the data rate of the incoming data. Clock RXUSRCLK defines the rate at which the FPGA fabric consumes the data. Ideally, these rates are identical. However, since the clocks typically have different sources, one of the clocks will be faster than the other. The receiver buffer accommodates this difference between the clock rates. See [Figure 3](#).

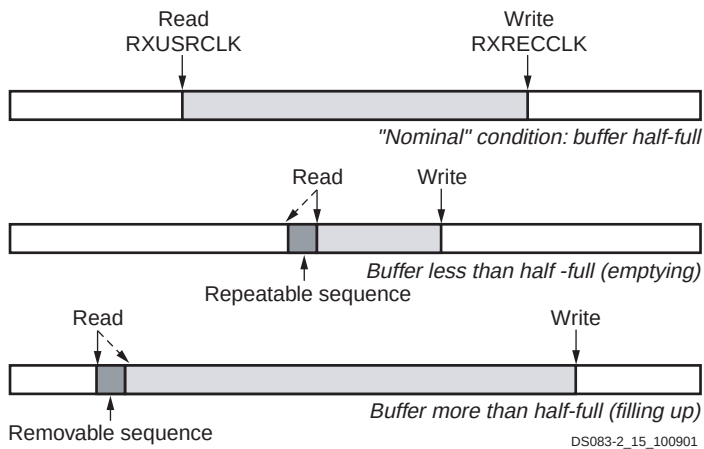


Figure 3: Clock Correction in Receiver

Nominally, the buffer is always half full. This is shown in the top buffer, [Figure 3](#), where the shaded area represents buffered data not yet read. Received data is inserted via the write pointer under control of RXRECCLK. The FPGA fabric reads data via the read pointer under control of RXUSRCLK. The half full/half empty condition of the buffer gives a cushion for the differing clock rates. This operation continues indefinitely, regardless of whether or not "meaningful" data is being received. When there is no meaningful data to be received, the incoming data will consist of IDLE characters or other padding.

If RXUSRCLK is faster than RXRECCLK, the buffer becomes more empty over time. The clock correction logic corrects for this by decrementing the read pointer to reread a repeatable byte sequence. This is shown in the middle buffer, [Figure 3](#), where the solid read pointer decrements to the value represented by the dashed pointer. By decrementing the read pointer instead of incrementing it in the usual fashion, the buffer is partially refilled. The transceiver design will repeat a single repeatable byte sequence when necessary to refill a buffer. If the byte sequence length is greater than one, and if attribute CLK_COR_REPEAT_WAIT is 0, then the transceiver may repeat the same sequence multiple times until the buffer is refilled to the desired extent.

Similarly, if RXUSRCLK is slower than RXRECCLK, the buffer will fill up over time. The clock correction logic corrects for this by incrementing the read pointer to skip over a removable byte sequence that need not appear in the final FPGA fabric byte stream. This is shown in the bottom buffer, [Figure 3](#), where the solid read pointer increments to the value represented by the dashed pointer. This accelerates the emptying of the buffer, preventing its overflow. The transceiver design will skip a single byte sequence when necessary to partially empty a buffer. If attribute CLK_COR_REPEAT_WAIT is 0, the transceiver may also skip two consecutive removable byte sequences in one step to further empty the buffer when necessary.

These operations require the clock correction logic to recognize a byte sequence that can be freely repeated or omitted in the incoming data stream. This sequence is generally an IDLE sequence, or other sequence comprised of special values that occur in the gaps separating packets of meaningful data. These gaps are required to occur sufficiently often to facilitate the timely execution of clock correction.

Channel Bonding

Some gigabit I/O standards such as Infiniband specify the use of multiple transceivers in parallel for even higher data rates. Words of data are split into bytes, with each byte sent over a separate channel (transceiver). See [Figure 4](#).

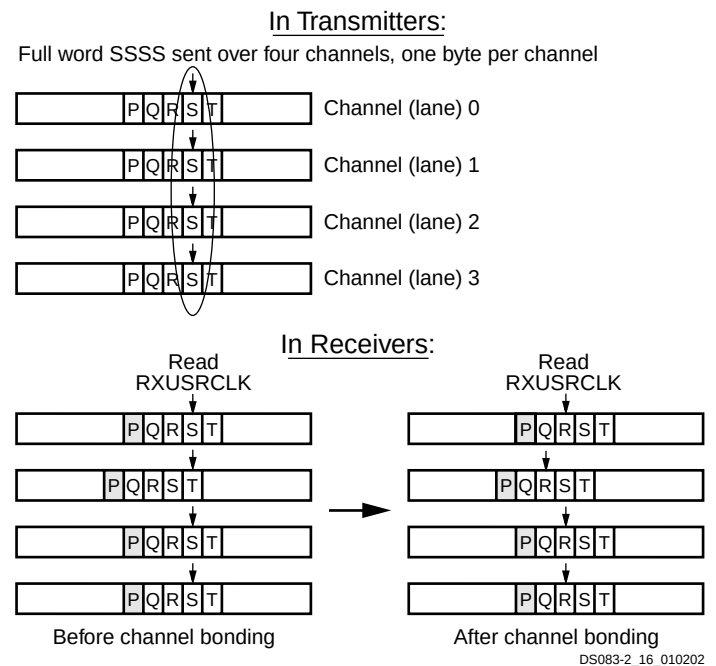


Figure 4: Channel Bonding (Alignment)

The top half of the figure shows the transmission of words split across four transceivers (channels or lanes). PPPP, QQQQ, RRRR, SSSS, and TTTT represent words sent over the four channels.

The bottom-left portion of the figure shows the initial situation in the FPGA's receivers at the other end of the four channels. Due to variations in transmission delay—especially if the channels are routed through repeaters—the FPGA fabric may not correctly assemble the bytes into complete words. The bottom-left illustration shows the incorrect assembly of data words PQPP, QRQQ, RSRR, etc.

To support correction of this misalignment, the data stream will include special byte sequences that define corresponding points in the several channels. In the bottom half of **Figure 4**, the shaded "P" bytes represent these special characters. Each receiver recognizes the "P" channel bonding character, and remembers its location in the buffer. At some point, one transceiver designated as the master instructs all the transceivers to align to the channel bonding character "P" (or to some location relative to the channel bonding character). After this operation, the words transmitted to the FPGA fabric will be properly aligned: RRRR, SSSS, TTTT, etc., as shown in the bottom-right portion of **Figure 4**. To ensure that the channels remain properly aligned following the channel bonding operation, the master transceiver must also control the clock correction operations described in the previous section for all channel-bonded transceivers.

Transmitter Buffer

The transmitter's buffer write pointer (TXUSRCLK) is frequency-locked to its read pointer (REFCLK). Therefore, clock correction and channel bonding are not required. The purpose of the transmitter's buffer is to accommodate a phase difference between TXUSRCLK and REFCLK. A simple FIFO suffices for this purpose. A FIFO depth of four will permit reliable operation with simple detection of overflow or underflow, which could occur if the clocks are not frequency-locked.

CRC

The Rocket I/O transceiver CRC logic supports the 32-bit invariant CRC calculation used by Infiniband, FibreChannel, and Gigabit Ethernet.

On the transmitter side, the CRC logic recognizes where the CRC bytes should be inserted and replaces four placeholder bytes at the tail of a data packet with the computed CRC. For Gigabit Ethernet and FibreChannel, transmitter CRC may adjust certain trailing bytes to generate the required running disparity at the end of the packet.

On the receiver side, the CRC logic verifies the received CRC value, supporting the same standards as above.

The CRC logic also supports a user mode, with a simple data packet structure beginning and ending with user-defined SOP and EOP characters.

Configuration

This section outlines functions that may be selected or con-

trolled by configuration. Xilinx implementation software supports 16 transceiver primitives, as shown in **Table 2**.

Table 2: Supported Rocket I/O Transceiver Primitives

GT_CUSTOM	Fully customizable by user
GT_FIBRE_CHAN_1	Fibre Channel, 1-byte data path
GT_FIBRE_CHAN_2	Fibre Channel, 2-byte data path
GT_FIBRE_CHAN_4	Fibre Channel, 4-byte data path
GT_ETHERNET_1	Gigabit Ethernet, 1-byte data path
GT_ETHERNET_2	Gigabit Ethernet, 2-byte data path
GT_ETHERNET_4	Gigabit Ethernet, 4-byte data path
GT_XAUI_1	10-gigabit Ethernet, 1-byte data path
GT_XAUI_2	10-gigabit Ethernet, 2-byte data path
GT_XAUI_4	10-gigabit Ethernet, 4-byte data path
GT_INFINIBAND_1	Infiniband, 1-byte data path
GT_INFINIBAND_2	Infiniband, 2-byte data path
GT_INFINIBAND_4	Infiniband, 4-byte data path
GT_AURORA_1	Xilinx protocol, 1-byte data path
GT_AURORA_2	Xilinx protocol, 2-byte data path
GT_AURORA_4	Xilinx protocol, 4-byte data path

Each of the above primitives defines default values for the configuration attributes, allowing some number of them to be modified by the user.

Refer to the *Rocket I/O User Guide* for more details.

Reset / Power Down

The receiver and transmitter have their own synchronous reset inputs. The transmitter reset recenters the transmission FIFO, and resets all transmitter registers and the 8B/10B decoder. The receiver reset recenters the receiver elastic buffer, and resets all receiver registers and the 8B/10B encoder. Neither reset signal has any effect on the PLLs.

The Power Down module is controlled by the POWER-DOWN input pin on the transceiver core. The Power down pin on the FPGA package has no effect on the transceiver core.

Power Sequencing

Although applying power in a random order does not damage the device, it is recommended to apply power in the following sequence to minimize power-on current:

1. Apply FPGA fabric power supplies (V_{CCINT} and V_{CCAUX}) in any order.
2. Apply AVCCAUXRX.
3. Apply AVCCAUTX, V_{TTX} , and V_{TRX} in any order.

Functional Description: Processor Block

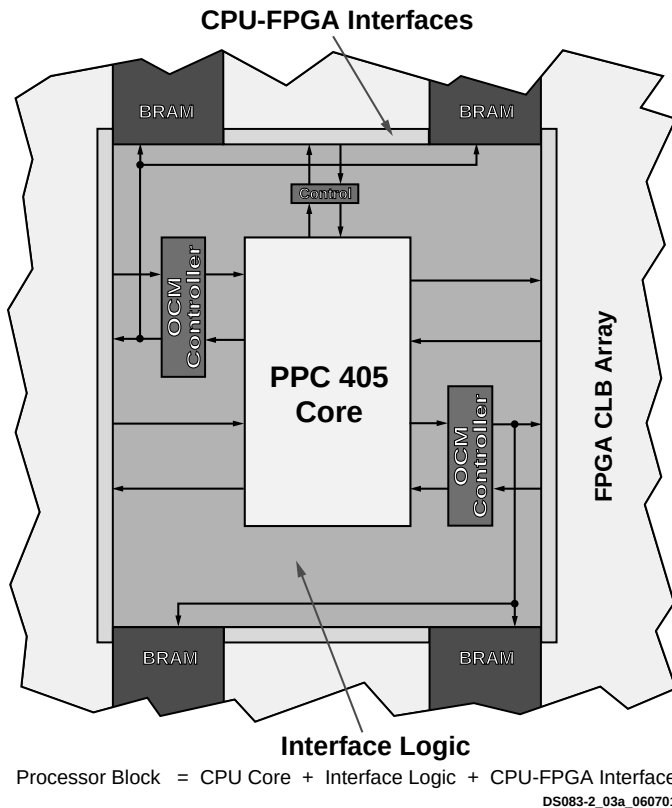


Figure 5: Processor Block Architecture

This section briefly describes the interfaces and components of the Processor Block. The subsequent section, **Functional Description: PowerPC 405 Core** beginning on page 93, offers a summary of major PPC405 core features. For an in-depth discussion on both Processor Block and PPC405, refer to the *Processor Block Manual* and the *PPC405 User Manual*.

Processor Block Overview

Figure 5 shows the internal architecture of the Processor Block.

Within the Virtex-II Pro Processor Block, there are four components:

- Embedded IBM PowerPC 405-D5 RISC CPU core
- On-Chip Memory (OCM) controllers and interfaces
- Clock/control interface logic
- CPU-FPGA Interfaces

Embedded PowerPC 405 RISC Core

The PowerPC 405D5 core is a 0.13 μm implementation of the IBM PowerPC 405D4 core. The advanced process technology enables the embedded PowerPC 405 (PPC405) core to operate at 300+ MHz while maintaining low power

consumption. Specially designed interface logic integrates the core with the surrounding CLBs, block RAMs, and general routing resources. Up to four Processor Blocks can be available in a single Virtex-II Pro device.

The PPC405 core implements the PowerPC User Instruction Set Architecture (UISA), user-level registers, programming model, data types, and addressing modes for 32-bit fixed-point operations. 64-bit operations, auxiliary processor operations, and floating-point operations are trapped and can be emulated in software.

Most of the PPC405 core features are compatible with the specifications for the PowerPC Virtual Environment Architecture (VEA) and Operating Environment Architecture (OEA). They also provide a number of optimizations and extensions to the lower layers of the PowerPC Architecture. The full architecture of the PPC405 is defined by the *PowerPC Embedded Environment* and the *PowerPC UISA*.

On-Chip Memory (OCM) Controllers

Introduction

The OCM controllers serve as dedicated interfaces between the block RAMs in the FPGA fabric (see **18 Kb Block SelectRAM Resources**, page 113) and OCM signals available on the embedded PPC405 core. The OCM signals on the PPC405 core are designed to provide very quick access to a fixed amount of instruction and data memory space. The OCM controller provides an interface to both the 64-bit Instruction-Side Block RAM (ISBRAM) and the 32-bit Data-Side Block RAM (DSBRAM). The designer can choose to implement:

- ISBRAM only
- DSBRAM only
- Both ISBRAM and DSBRAM
- No ISBRAM and no DSBRAM

One of OCM's primary advantages is that it guarantees a fixed latency of execution for a higher level of determinism. Additionally, it reduces cache pollution and thrashing, since the cache remains available for caching code from other memory resources.

Typical applications for DSOCM include scratch-pad memory, as well as use of the dual-port feature of block RAM to enable bidirectional data transfer between processor and FPGA. Typical applications for ISOCM include storage of interrupt service routines.

Functional Features

Common Features

- Separate Instruction and Data memory interface between Processor core and BRAMs in FPGA
- Dedicated interface to Device Control Register (DCR) bus for ISOCM and DSOCM
- Single-cycle and multi-cycle mode option for I-side and D-side interfaces

- Single cycle = one clock cycle; multi-cycle = minimum of two and maximum of eight clock cycles
- FPGA configurable DCR addresses within DSOCM and ISOCM
- Independent 16 MB logical memory space available within PPC405 memory map for each of the DSOCM and ISOCM. The number of block RAMs in the device may limit the maximum amount of OCM supported.
- Maximum of 64K and 128K bytes addressable from DSOCM and ISOCM interfaces, respectively, using address outputs from OCM directly without additional decoding logic

Data-Side OCM (DSOCM)

- 32-bit Data Read bus and 32-bit Data Write bus
- Byte write access to DSBRAM support
- Second port of dual port DSBRAM is available to read/write from an FPGA interface
- 22-bit address to DSBRAM port
- 8-bit DCR Registers: DSCNTL, DSARC
- Three alternatives to write into DSBRAM: BRAM initialization, CPU, FPGA H/W using second port

Instruction-Side OCM (ISOCM)

The ISOCM interface contains a 64-bit read only port, for instruction fetches, and a 32-bit write only port, to initialize or test the ISBRAM. When implementing the read only port, the user must deassert the write port inputs. The preferred method of initializing the ISBRAM is through the configuration bitstream.

- 64-bit Data Read Only bus (two instructions per cycle)
- 32-bit Data Write Only bus (through DCR)
- Separate 21-bit address to ISBRAM
- 8-bit DCR Registers: ISCNTL, ISARC
- 32-bit DCR Registers: ISINIT, ISFILL
- Two alternatives to write into ISBRAM: BRAM initialization, DCR and write instruction

Clock/Control Interface Logic

The clock/control interface logic provides proper initialization and connections for PPC405 clock/power management, resets, PLB cycle control, and OCM interfaces. It also couples user signals between the FPGA fabric and the PPC405 CPU core.

The processor clock connectivity is similar to CLB clock pins. It can connect either to global clock nets or general routing resources. Therefore the processor clock source can come from DCM, CLB, or user package pin.

CPU-FPGA Interfaces

All Processor Block user pins link up with the general FPGA routing resources through the CPU-FPGA interface. There-

fore processor signals have the same routability as other non-Processor Block user signals. Longlines and hex lines travel across the Processor Block both vertically and horizontally, allowing signals to route through the Processor Block.

Processor Local Bus (PLB) Interfaces

The PPC405 core accesses high-speed system resources through PLB interfaces on the instruction and data cache controllers. The PLB interfaces provide separate 32-bit address/64-bit data buses for the instruction and data sides.

The cache controllers are both PLB masters. PLB arbiters can be implemented on FPGA fabric and are available as soft IP cores.

Device Control Register (DCR) Bus Interface

The device control register (DCR) bus has 10 bits of address space for components external to the PPC405 core. Using the DCR bus to manage status and configuration registers reduces PLB traffic and improves system integrity. System resources on the DCR bus are protected or isolated from wayward code since the DCR bus is not part of the system memory map.

On-Chip Memory (OCM) Interfaces

Access to optional, user-configurable direct-mapped memory is through the OCM interfaces. The OCM interfaces can have the same access time as a cache hit, depending on the clock frequency and block RAM size. OCM may be attached to the PPC405 core through the instruction OCM interface and/or the data OCM interface.

Instruction side OCM is often used to hold critical code such as an interrupt handler that requires guaranteed low-latency deterministic access. Data side OCM offers the same fixed low-latency access and is used to hold critical data such as filter coefficients for a DSP application or packets for fast processing. Refer to **On-Chip Memory (OCM) Controllers**, page 90, for more information.

External Interrupt Controller (EIC) Interface

Two level-sensitive user interrupt pins (critical and non-critical) are available. They can be either driven by user defined logic or Xilinx soft interrupt controller IP core outside the Processor Block.

Clock/Power Management (CPM) Interface

The CPM interface supports several methods of clock distribution and power management. Three modes of operation that reduce power consumption below the normal operational level are available.

Reset Interface

There are three user reset input pins (core, chip, and system) and three user reset output pins for different levels of reset, if required.

Debug Interface

Debugging interfaces on the PPC405 core, consisting of the JTAG and Trace ports, offer access to resources internal to the core and assist in software development. The JTAG port provides basic JTAG chip testing functionality as well as the ability for external debug tools to gain control of the processor for debug purposes. The Trace port furnishes programmers with a mechanism for acquiring instruction execution traces.

The JTAG port complies with IEEE Std 1149.1, which defines a test access port (TAP) and boundary scan architecture. Extensions to the JTAG interface provide debuggers with processor control that includes stopping, starting, and stepping the PPC405 core. These extensions are compliant with the IEEE 1149.1 specifications for vendor-specific extensions.

The Trace port provides instruction execution trace information to an external trace tool. The PPC405 core is capable of back trace and forward trace. Back trace is the tracing of instructions prior to a debug event while forward trace is the tracing of instructions after a debug event.

The processor JTAG port can be accessed independently from the FPGA JTAG port, or the two can be programmatically linked together and accessed via the FPGA's dedicated JTAG pins.

CoreConnect™ Bus Architecture

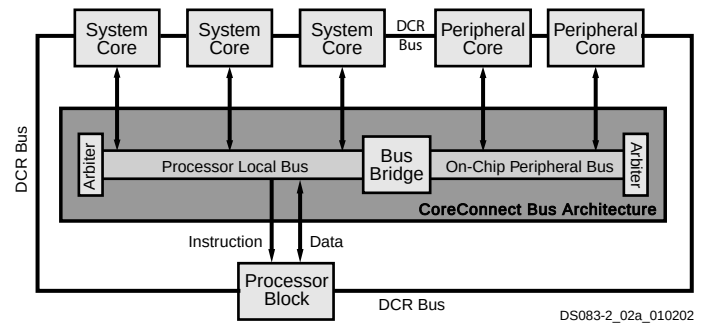


Figure 6: CoreConnect Block Diagram

The Processor Block is compatible with the CoreConnect™ bus architecture. Any CoreConnect compliant cores including Xilinx soft IP can integrate with the Processor Block through this high-performance bus architecture implemented on FPGA fabric.

The CoreConnect architecture provides three buses for interconnecting Processor Blocks, Xilinx soft IP, third party IP, and custom logic, as shown in Figure 6:

- Processor Local Bus (PLB)
- On-Chip Peripheral Bus (OPB)
- Device Control Register (DCR) bus

High-performance peripherals connect to the high-bandwidth, low-latency PLB. Slower peripheral cores connect to the OPB, which reduces traffic on the PLB, resulting in greater overall system performance.

For more information, refer to:

[http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect Bus Architecture/](http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect%20Bus%20Architecture/)

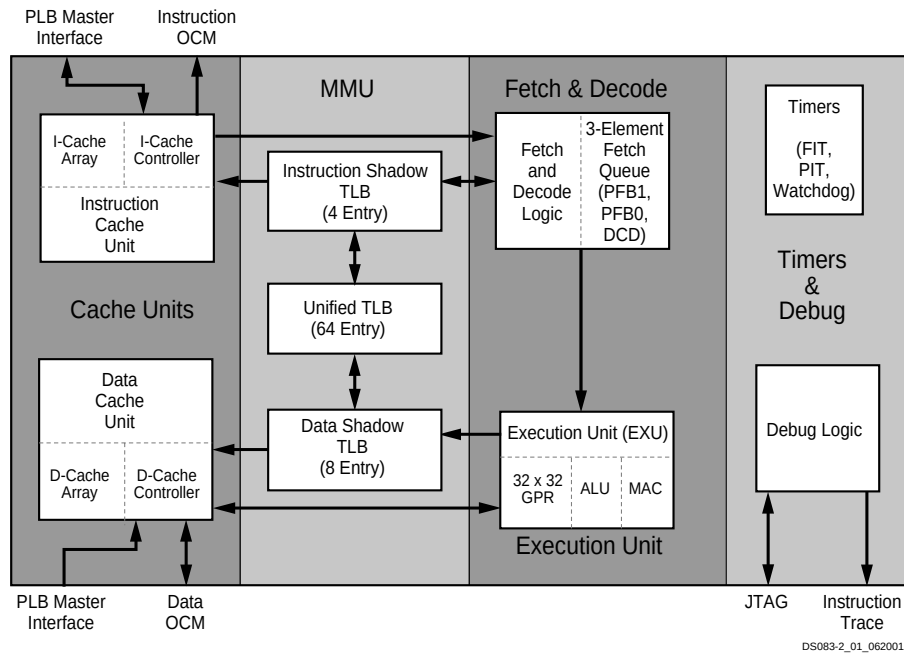


Figure 7: PPC405 Core Block Diagram

Functional Description: PowerPC 405 Core

This section offers a brief overview of the various functional blocks shown in Figure 7.

PPC405 Core

The PPC405 core is a 32-bit Harvard architecture processor. It consists of the following functional blocks as shown in Figure 7:

- Cache units
- Memory Management unit
- Fetch Decode unit
- Execution unit
- Timers
- Debug logic unit

It operates on instructions in a five stage pipeline consisting of a fetch, decode, execute, write-back, and load write-back stage. Most instructions execute in a single cycle, including loads and stores.

Instruction and Data Cache

The PPC405 core provides an instruction cache unit (ICU) and a data cache unit (DCU) that allow concurrent accesses and minimize pipeline stalls. The instruction and data cache array are 16 KB each. Both cache units are two-way set associative. Each way is organized into 256 lines of 32 bytes (eight words). The instruction set provides a rich assortment of cache control instructions, including instructions to read tag information and data arrays.

The PPC405 core accesses external memory through the instruction (ICU) and data cache units (DCU). The cache units each include a 64-bit PLB master interface, cache arrays, and a cache controller. The ICU and DCU handle cache misses as requests over the PLB to another PLB device such as an external bus interface unit. Cache hits are handled as single cycle memory accesses to the instruction and data caches.

Instruction Cache Unit (ICU)

The ICU provides one or two instructions per cycle to the instruction queue over a 64-bit bus. A line buffer (built into the output of the array for manufacturing test) enables the ICU to be accessed only once for every four instructions, to reduce power consumption by the array.

The ICU can forward any or all of the four or eight words of a line fill to the EXU to minimize pipeline stalls caused by cache misses. The ICU aborts speculative fetches abandoned by the EXU, eliminating unnecessary line fills and enabling the ICU to handle the next EXU fetch. Aborting abandoned requests also eliminates unnecessary external bus activity, thereby increasing external bus utilization.

Data Cache Unit (DCU)

The DCU transfers one, two, three, four, or eight bytes per cycle, depending on the number of byte enables presented by the CPU. The DCU contains a single-element command and store data queue to reduce pipeline stalls; this queue enables the DCU to independently process load/store and cache control instructions. Dynamic PLB request prioritization reduces pipeline stalls even further. When the DCU is busy with a low-priority request while a subsequent storage

operation requested by the CPU is stalled; the DCU automatically increases the priority of the current request to the PLB.

The DCU provides additional features that allow the programmer to tailor its performance for a given application. The DCU can function in write-back or write-through mode, as controlled by the Data Cache Write-through Register (DCWR) or the Translation Look-aside Buffer (TLB); the cache controller can be tuned for a balance of performance and memory coherency. Write-on-allocate, controlled by the store word on allocate (SWOA) field of the Core Configuration Register 0 (CCR0), can inhibit line fills caused by store misses, to further reduce potential pipeline stalls and unwanted external bus traffic.

Fetch and Decode Logic

The fetch and decode logic maintains a steady flow of instructions to the execution unit by placing up to two instructions in the fetch queue. The fetch queue consists of three buffers: pre-fetch buffer 1 (PFB1), pre-fetch buffer 0 (PFB0) and decode (DCD). The fetch logic ensures that instructions proceed directly to decode when the queue is empty.

Static branch prediction as implemented on the PPC405 core takes advantage of some standard statistical properties of code. Branches with negative address displacement are by default assumed taken. Branches that do not test the condition or count registers are also predicted as taken. The PPC405 core bases branch prediction upon these default conditions when a branch is not resolved and speculatively fetches along the predicted path. The default prediction can be overridden by software at assembly or compile time.

Branches are examined in the decode and pre-fetch buffer 0 fetch queue stages. Two branch instructions can be handled simultaneously. If the branch in decode is not taken, the fetch logic fetches along the predicted path of the branch instruction in pre-fetch buffer 0. If the branch in decode is taken, the fetch logic ignores the branch instruction in pre-fetch buffer 0.

Execution Unit

The PPC405 core has a single issue execution unit (EXU), which contains the register file, arithmetic logic unit (ALU), and the multiply-accumulate (MAC) unit. The execution unit performs all 32-bit PowerPC integer instructions in hardware.

The register file is comprised of thirty-two 32-bit general purpose registers (GPR), which are accessed with three read ports and two write ports. During the decode stage, data is read out of the GPRs and fed to the execution unit. Likewise, during the write-back stage, results are written to the GPR. The use of the five ports on the register file enables either a load or a store operation to execute in parallel with an ALU operation.

Memory Management Unit (MMU)

The PPC405 core has a 4 GB address space, which is presented as a flat address space.

The MMU provides address translation, protection functions, and storage attribute control for embedded applications. The MMU supports demand-paged virtual memory and other management schemes that require precise control of logical-to-physical address mapping and flexible memory protection. Working with appropriate system-level software, the MMU provides the following functions:

- Translation of the 4 GB effective address space into physical addresses
- Independent enabling of instruction and data translation/protection
- Page-level access control using the translation mechanism
- Software control of page replacement strategy
- Additional control over protection using zones
- Storage attributes for cache policy and speculative memory access control

The MMU can be disabled under software control. If the MMU is not used, the PPC405 core provides other storage control mechanisms.

Translation Look-Aside Buffer (TLB)

The Translation Look-Aside Buffer (TLB) is the hardware resource that controls translation and protection. It consists of 64 entries, each specifying a page to be translated. The TLB is fully associative; a given page entry can be placed anywhere in the TLB. The translation function of the MMU occurs pre-cache. Cache tags and indexing use physical addresses.

Software manages the establishment and replacement of TLB entries. This gives system software significant flexibility in implementing a custom page replacement strategy. For example, to reduce TLB thrashing or translation delays, software can reserve several TLB entries in the TLB for globally accessible static mappings. The instruction set provides several instructions used to manage TLB entries. These instructions are privileged and require the software to be executing in supervisor state. Additional TLB instructions are provided to move TLB entry fields to and from GPRs.

The MMU divides logical storage into pages. Eight page sizes (1 KB, 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, and 16 MB) are simultaneously supported, such that, at any given time, the TLB can contain entries for any combination of page sizes. In order for a logical to physical translation to exist, a valid entry for the page containing the logical address must be in the TLB. Addresses for which no TLB entry exists cause TLB-Miss exceptions.

To improve performance, four instruction-side and eight data-side TLB entries are kept in shadow arrays. The

shadow arrays allow single-cycle address translation and also help to avoid TLB contention between load/store and instruction fetch operations. Hardware manages the replacement and invalidation of shadow-TLB entries; no system software action is required.

Memory Protection

When address translation is enabled, the translation mechanism provides a basic level of protection.

The Zone Protection Register (ZPR) enables the system software to override the TLB access controls. For example, the ZPR provides a way to deny read access to application programs. The ZPR can be used to classify storage by type; access by type can be changed without manipulating individual TLB entries.

The PowerPC Architecture provides WIU0GE (write-back / write-through, cacheability, user-defined 0, guarded, endian) storage attributes that control memory accesses, using bits in the TLB or, when address translation is disabled, storage attribute control registers.

When address translation is enabled, storage attribute control bits in the TLB control the storage attributes associated with the current page. When address translation is disabled, bits in each storage attribute control register control the storage attributes associated with storage regions. Each storage attribute control register contains 32 fields. Each field sets the associated storage attribute for a 128 MB memory region.

Timers

The PPC405 core contains a 64-bit time base and three timers, as shown in **Figure 8**:

- Programmable Interval Timer (PIT)
- Fixed Interval Timer (FIT)
- Watchdog Timer (WDT)

The time base counter increments either by an internal signal equal to the CPU clock rate or by a separate external timer clock signal. No interrupts are generated when the time base rolls over. The three timers are synchronous with the time base.

The PIT is a 32-bit register that decrements at the same rate as the time base is incremented. The user loads the PIT register with a value to create the desired delay. When the register reaches zero, the timer stops decrementing and generates a PIT interrupt. Optionally, the PIT can be programmed to auto-reload the last value written to the PIT register, after which the PIT continues to decrement.

The FIT generates periodic interrupts based on one of four selectable bits in the time base. When the selected bit changes from 0 to 1, the PPC405 core generates a FIT interrupt.

The WDT provides a periodic critical-class interrupt based on a selected bit in the time base. This interrupt can be used

for system error recovery in the event of software or system lockups. Users may select one of four time periods for the interval and the type of reset generated if the WDT expires twice without an intervening clear from software. If enabled, the watchdog timer generates a reset unless an exception handler updates the WDT status bit before the timer has completed two of the selected timer intervals.

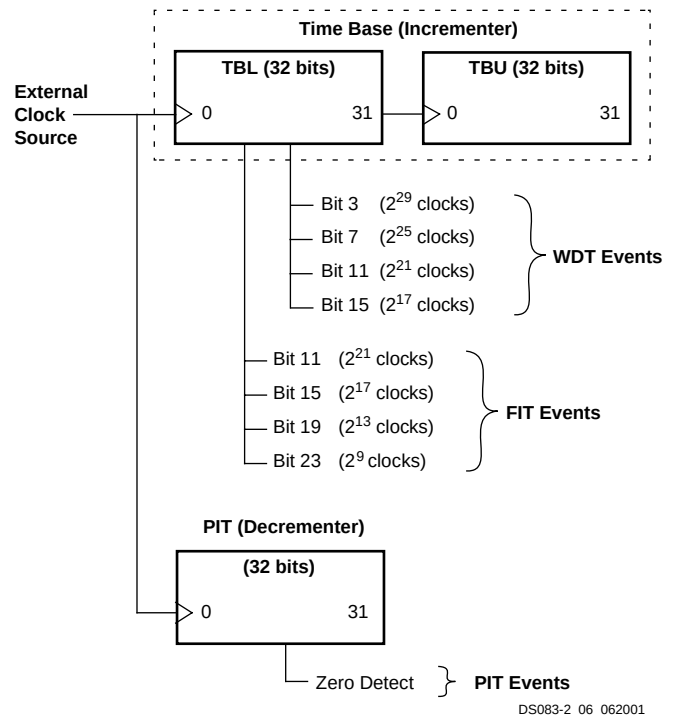


Figure 8: Relationship of Timer Facilities to Base Clock

Interrupts

The PPC405 provides an interface to an interrupt controller that is logically outside the PPC405 core. This controller combines the asynchronous interrupt inputs and presents them to the core as a single interrupt signal. The sources of asynchronous interrupts are external signals, the JTAG/debug unit, and any implemented peripherals.

Debug Logic

All architected resources on the PPC405 core can be accessed through the debug logic. Upon a debug event, the PPC405 core provides debug information to an external debug tool. Three different types of tools are supported depending on the debug mode: ROM monitors, JTAG debuggers, and instruction trace tools.

In internal (intrusive) debug mode, a debug event enables exception-handling software at a dedicated interrupt vector to take over the CPU core and communicate with a debug tool. The debug tool has read-write access to all registers and can set hardware or software breakpoints. ROM monitors typically use the internal debug mode.

In external (non-intrusive) debug mode, the CPU core enters stop state (stops instruction execution) when a debug event occurs. This mode offers a debug tool non-intrusive read-write access to all registers in the PPC405 core. Once the CPU core is in stop state, the debug tool can start the CPU core, step an instruction, freeze the timers, or set hardware or software break points. In addition to CPU core control, the debug logic is capable of writing instructions into the instruction cache, eliminating the need for external memory during initial board bring up. Communication to a debug tool using external debug mode is through the JTAG port.

Debug wait mode offers the same functionality as external debug mode with one exception. In debug wait mode, the CPU core goes into wait state instead of stop state after a debug event. Wait state is identical to stop state until an interrupt occurs. In wait state, the PPC405 core can vector to an exception handler, service an interrupt and return to wait state. This mode is particularly useful when debugging real time control systems.

Real-time trace debug mode is always enabled. The debug logic continuously broadcasts instruction trace information to the trace port. When a debug event occurs, the debug logic signals an external debug tool to save instruction trace information before and after the event. The number of instructions traced depends on the trace tool.

Debug events signal the debug logic to stop the CPU core, put the CPU core in debug wait state, cause a debug exception or save instruction trace information.

Big Endian and Little Endian Support

The PPC405 core supports big endian or little endian byte ordering for instructions stored in external memory. Since the PowerPC architecture is big endian internally, the ICU rearranges the instructions stored as little endian into the big endian format. Therefore, the instruction cache always contains instructions in big endian format so that the byte ordering is correct for the execution unit. This feature allows the 405 core to be used in systems designed to function in a little endian environment.

Functional Description: FPGA

Input/Output Blocks (IOBs)

Virtex-II Pro I/O blocks (IOBs) are provided in groups of two or four on the perimeter of each device. Each IOB can be used as input and/or output for single-ended I/Os. Two IOBs can be used as a differential pair. A differential pair is always connected to the same switch matrix, as shown in Figure 9.

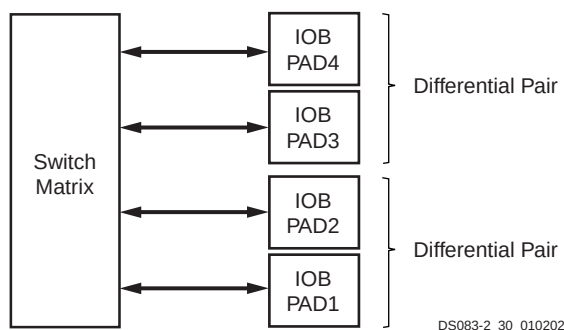


Figure 9: Virtex-II Pro Input/Output Tile

IOB blocks are designed for high-performance I/Os, supporting 22 single-ended standards, as well as differential signaling with LVDS, LDT, and bus LVDS.

Supported I/O Standards

Virtex-II Pro IOB blocks feature SelectI/O inputs and outputs that support a wide variety of I/O signaling standards. In addition to the internal supply voltage ($V_{CCINT} = 1.5V$), output driver supply voltage (V_{CCO}) is dependent on the I/O standard (see Table 3 and Table 4). An auxiliary supply voltage ($V_{CCAUX} = 2.5V$) is required, regardless of the I/O

standard used. For exact supply voltage absolute maximum ratings, see **Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics (Module 3)**.

Table 3: Supported Single-Ended I/O Standards

I/O Standard	Output V_{CCO}	Input V_{CCO}	Input V_{REF}	Board Termination Voltage (V_{TT})
LVTTTL	3.3	3.3	N/A	N/A
LVCNOS33	3.3	3.3	N/A	N/A
LVCNOS25	2.5	2.5	N/A	N/A
LVCNOS18	1.8	1.8	N/A	N/A
LVCNOS15	1.5	1.5	N/A	N/A
PCI33_3	3.3	3.3	N/A	N/A
PCI66_3	3.3	3.3	N/A	N/A
GTL	Note (1)	Note (1)	0.8	1.2
GTLP	Note (1)	Note (1)	1.0	1.5
HSTL_I	1.5	N/A	0.75	0.75
HSTL_II	1.5	N/A	0.75	0.75
HSTL_III	1.5	N/A	0.9	1.5
HSTL_IV	1.5	N/A	0.9	1.5
HSTL_I_18	1.8	N/A	0.9	0.9
HSTL_II_18	1.8	N/A	0.9	0.9
HSTL_III_18	1.8	N/A	1.08	1.8
HSTL_IV_18	1.8	N/A	1.08	1.8

Table 3: Supported Single-Ended I/O Standards

I/O Standard	Output V_{CCO}	Input V_{CCO}	Input V_{REF}	Board Termination Voltage (V_{TT})
SSTL2_I	2.5	N/A	1.25	1.25
SSTL2_II	2.5	N/A	1.25	1.25
SSTL3_I	3.3	N/A	1.5	1.5
SSTL3_II	3.3	N/A	1.5	1.5

Notes:

- V_{CCO} of GTL or GTLP should not be lower than the termination voltage or the voltage seen at the I/O pad.

Table 4: Supported Differential Signal I/O Standards

I/O Standard	Output V_{CCO}	Input V_{CCO}	Input V_{REF}	Output V_{OD}
LDT_25	2.5	N/A	N/A	0.500 - 0.740
LVDS_25	2.5	N/A	N/A	0.250 - 0.400
LVDS_25	2.5	N/A	N/A	0.330 - 0.700
BLVDS_25	2.5	N/A	N/A	0.250 - 0.450
ULVDS_25	2.5	N/A	N/A	0.500 - 0.740

All of the user IOBs have fixed-clamp diodes to V_{CCO} and to ground. The IOBs are not compatible or compliant with 5V I/O standards (not 5V tolerant).

Table 5 lists supported I/O standards with Digitally Controlled Impedance. See **Digitally Controlled Impedance (DCI)**, page 101.

Table 5: Supported DCI I/O Standards

I/O Standard	Output V_{CCO}	Input V_{CCO}	Input V_{REF}	Termination Type
LVDCI_33 ⁽¹⁾	3.3	3.3	N/A	Series
LVDCI_25	2.5	2.5	N/A	Series
LVDCI_DV2_25	2.5	2.5	N/A	Series
LVDCI_18	1.8	1.8	N/A	Series
LVDCI_DV2_18	1.8	1.8	N/A	Series
LVDCI_15	1.5	1.5	N/A	Series
LVDCI_DV2_15	1.5	1.5	N/A	Series
GTL_DCI	1.2	1.2	0.8	Single
GTLP_DCI	1.5	1.5	1.0	Single
HSTL_I_DCI	1.5	1.5	0.75	Split
HSTL_II_DCI	1.5	1.5	0.75	Split
HSTL_III_DCI	1.5	1.5	0.9	Single
HSTL_IV_DCI	1.5	1.5	0.9	Single

Table 5: Supported DCI I/O Standards (Continued)

I/O Standard	Output V_{CCO}	Input V_{CCO}	Input V_{REF}	Termination Type
HSTL_I_DCI_18	1.8	1.8	0.9	Split
HSTL_II_DCI_18	1.8	1.8	0.9	Split
HSTL_III_DCI_18	1.8	1.8	1.08	Single
HSTL_IV_DCI_18	1.8	1.8	1.08	Single
SSTL2_I_DCI ⁽²⁾	2.5	2.5	1.25	Split
SSTL2_II_DCI ⁽²⁾	2.5	2.5	1.25	Split
SSTL3_I_DCI ⁽²⁾	3.3	3.3	1.5	Split
SSTL3_II_DCI ⁽²⁾	3.3	3.3	1.5	Split

Notes:

- LVDCI_XX is LVCMOS controlled impedance buffers, matching the reference resistors or half of the reference resistors.
- These are SSTL compatible.

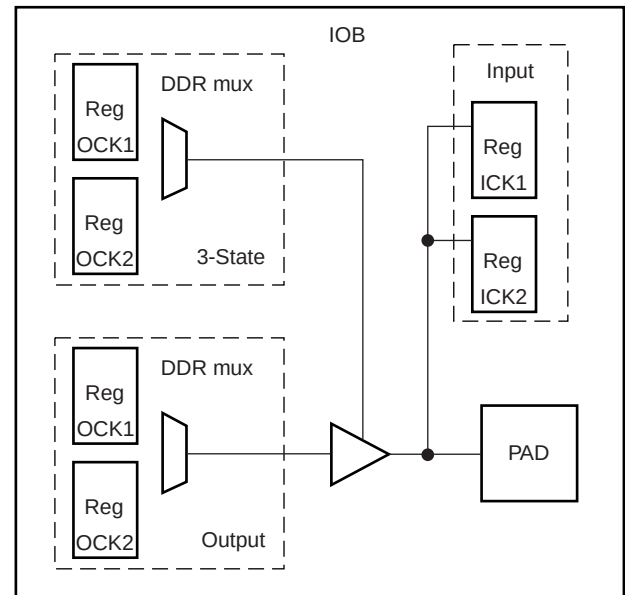


Figure 10: Virtex-II Pro IOB Block

Logic Resources

IOB blocks include six storage elements, as shown in Figure 10.

Each storage element can be configured either as an edge-triggered D-type flip-flop or as a level-sensitive latch. On the input, output, and 3-state path, one or two DDR registers can be used.

Double data rate is directly accomplished by the two registers on each path, clocked by the rising edges (or falling edges) from two different clock nets. The two clock signals are generated by the DCM and must be 180 degrees out of phase, as shown in Figure 11. There are two input, output, and 3-state data signals, each being alternately clocked out.

This DDR mechanism can be used to mirror a copy of the clock on the output. This is useful for propagating a clock along the data that has an identical delay. It is also useful for

multiple clock generation, where there is a unique clock driver for every clock load. Virtex-II Pro devices can produce many copies of a clock with very little skew.

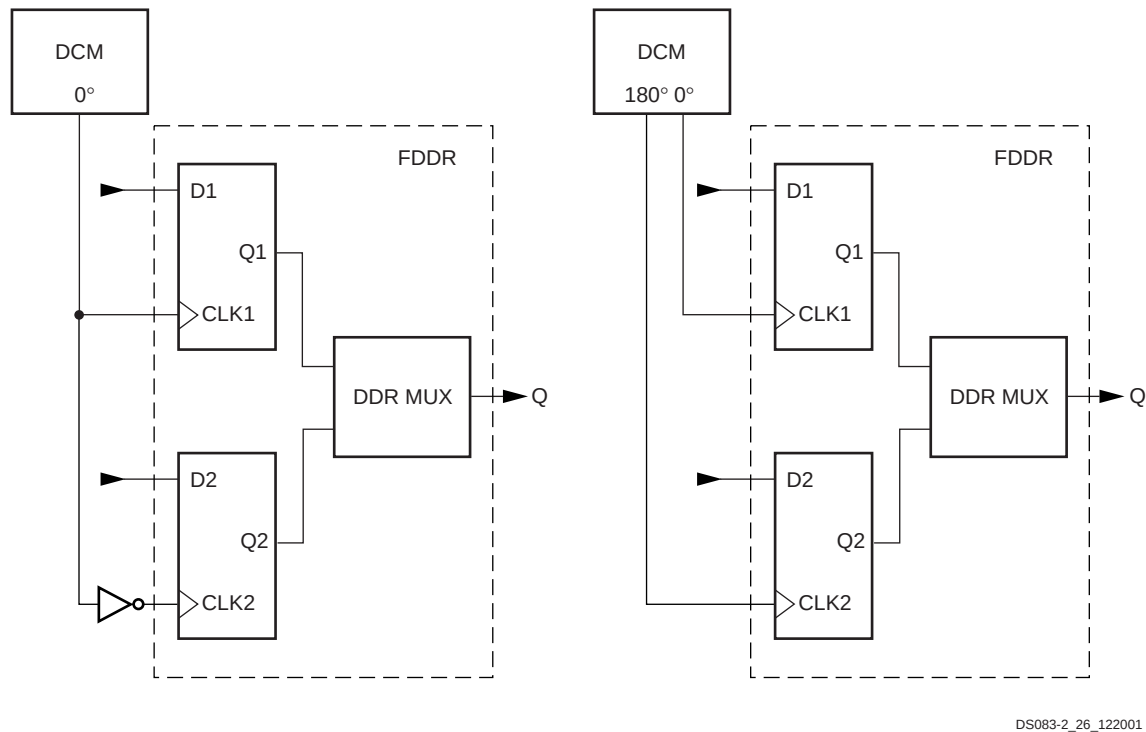


Figure 11: Double Data Rate Registers

Each group of two registers has a clock enable signal (ICE for the input registers, OCE for the output registers, and TCE for the 3-state registers). The clock enable signals are active High by default. If left unconnected, the clock enable for that storage element defaults to the active state.

Each IOB block has common synchronous or asynchronous set and reset (SR and REV signals).

SR forces the storage element into the state specified by the SRHIGH or SRLOW attribute. SRHIGH forces a logic 1. SRLOW forces a logic "0". When SR is used, a second input (REV) forces the storage element into the opposite state. The reset condition predominates over the set condition. The initial state after configuration or global initialization state is defined by a separate INIT0 and INIT1 attribute. By default, the SRLOW attribute forces INIT0, and the SRHIGH attribute forces INIT1.

For each storage element, the SRHIGH, SRLOW, INIT0, and INIT1 attributes are independent. Synchronous or asynchronous set / reset is consistent in an IOB block.

All the control signals have independent polarity. Any inverter placed on a control input is automatically absorbed.

Each register or latch, independent of all other registers or latches, can be configured as follows:

- No set or reset
- Synchronous set
- Synchronous reset
- Synchronous set and reset
- Asynchronous set (preset)
- Asynchronous reset (clear)
- Asynchronous set and reset (preset and clear)

The synchronous reset overrides a set, and an asynchronous clear overrides a preset.

Refer to [Figure 12](#).

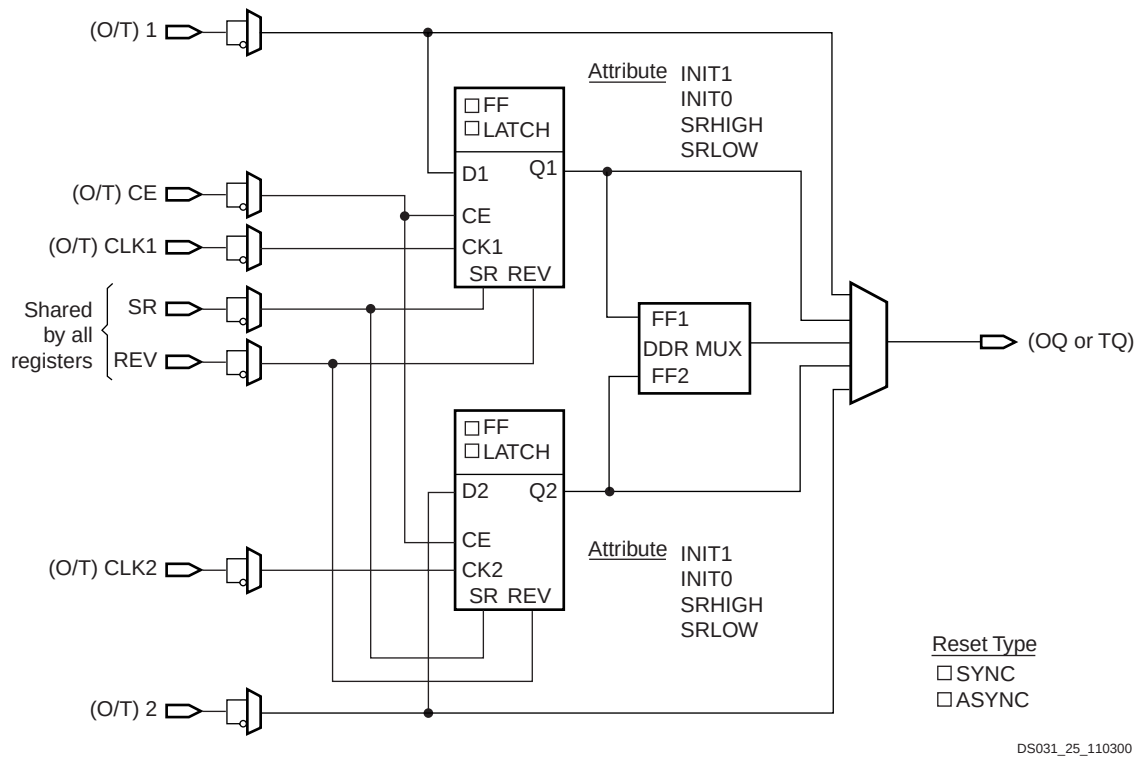


Figure 12: Register / Latch Configuration in an IOB Block

Input/Output Individual Options

Each device pad has optional pull-up/pull-down resistors and weak-keeper circuit in the LVCMOS SelectI/O configuration, as illustrated in Figure 13. Values of the optional pull-up and pull-down resistors fall within a range of 40 K Ω to 120 K Ω when $V_{CCO} = 2.5V$ (from 2.38V to 2.63V only). The clamp diode is always present, even when power is not.

The optional weak-keeper circuit is connected to each output. When selected, the circuit monitors the voltage on the pad and weakly drives the pin High or Low. If the pin is connected to a multiple-source signal, the weak-keeper holds the signal in its last state if all drivers are disabled. Maintaining a valid logic level in this way eliminates bus chatter. An enabled pull-up or pull-down overrides the weak-keeper circuit.

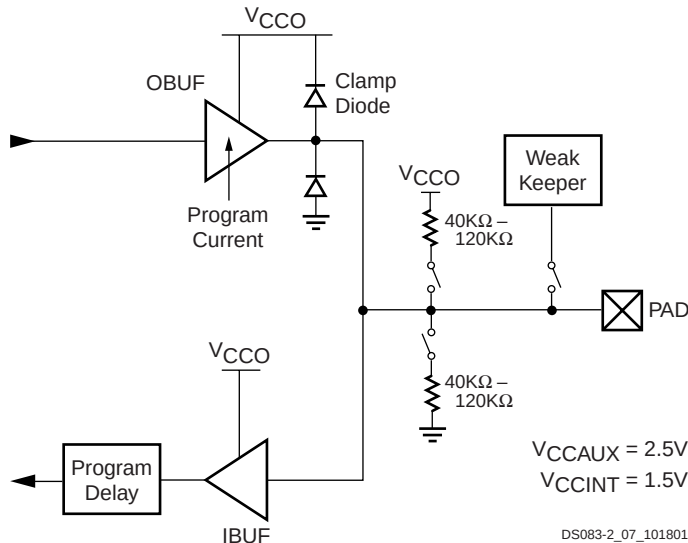


Figure 13: LVCMOS SelectI/O Standard

LVTTL sinks and sources current up to 24 mA. The current is programmable for LVTTL and LVCMOS SelectI/O standards (see Table 6). Drive strength and slew rate controls

for each output driver minimize bus transients. For LVDCI and LVDCI_DV2 standards, drive strength and slew rate controls are not available.

Table 6: LVTTL and LVCMOS Programmable Currents (Sink and Source)

SelectI/O	Programmable Current (Worst-Case Guaranteed Minimum)						
LVTTL	2 mA	4 mA	6 mA	8 mA	12 mA	16 mA	24 mA
LVCMOS33	2 mA	4 mA	6 mA	8 mA	12 mA	16 mA	24 mA
LVCMOS25	2 mA	4 mA	6 mA	8 mA	12 mA	16 mA	24 mA
LVCMOS18	2 mA	4 mA	6 mA	8 mA	12 mA	16 mA	n/a
LVCMOS15	2 mA	4 mA	6 mA	8 mA	12 mA	16 mA	n/a

Figure 14 shows the SSTL2 and HSTL configurations. HSTL can sink current up to 48 mA. (HSTL IV)

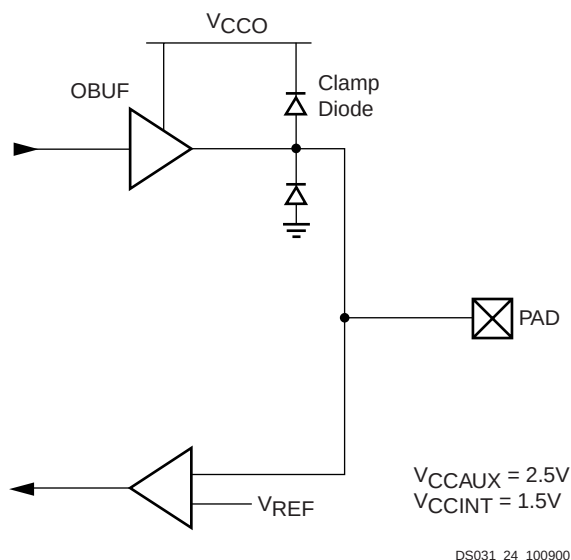


Figure 14: SSTL or HSTL SelectI/O Standards

All pads are protected against damage from electrostatic discharge (ESD) and from over-voltage transients. Virtex-II Pro uses two memory cells to control the configuration of an I/O as an input. This is to reduce the probability of an I/O configured as an input from flipping to an output when subjected to a single event upset (SEU) in space applications.

Prior to configuration, all outputs not involved in configuration are forced into their high-impedance state. The pull-down resistors and the weak-keeper circuits are inactive. The dedicated pin HSWAP_EN controls the pull-up resistors prior to configuration. By default, HSWAP_EN is set High, which disables the pull-up resistors on user I/O pins. When HSWAP_EN is set Low, the pull-up resistors are activated on user I/O pins.

All Virtex-II Pro IOBs (except Rocket I/O pins) support IEEE 1149.1 and IEEE 1532 compatible boundary scan testing.

Input Path

The Virtex-II Pro IOB input path routes input signals directly to internal logic and / or through an optional input flip-flop or latch, or through the DDR input registers. An optional delay element at the D-input of the storage element eliminates pad-to-pad hold time. The delay is matched to the internal clock-distribution delay of the Virtex-II Pro device, and when used, assures that the pad-to-pad hold time is zero.

Each input buffer can be configured to conform to any of the low-voltage signaling standards supported. In some of these standards the input buffer utilizes a user-supplied threshold voltage, V_{REF} . The need to supply V_{REF} imposes constraints on which standards can be used in the same bank. See I/O banking description.

Output Path

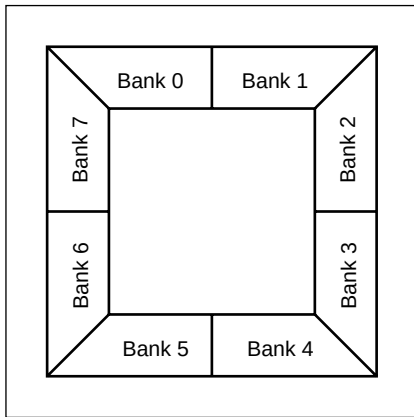
The output path includes a 3-state output buffer that drives the output signal onto the pad. The output and / or the 3-state signal can be routed to the buffer directly from the internal logic or through an output / 3-state flip-flop or latch, or through the DDR output / 3-state registers.

Each output driver can be individually programmed for a wide range of low-voltage signaling standards. In most signaling standards, the output High voltage depends on an externally supplied V_{CCO} voltage. The need to supply V_{CCO} imposes constraints on which standards can be used in the same bank. See I/O banking description.

I/O Banking

Some of the I/O standards described above require V_{CCO} and V_{REF} voltages. These voltages are externally supplied and connected to device pins that serve groups of IOB blocks, called banks. Consequently, restrictions exist about which I/O standards can be combined within a given bank.

Eight I/O banks result from dividing each edge of the FPGA into two banks, as shown in Figure 15 and Figure 16. Each bank has multiple V_{CCO} pins, all of which must be connected to the same voltage. This voltage is determined by the output standards in use.

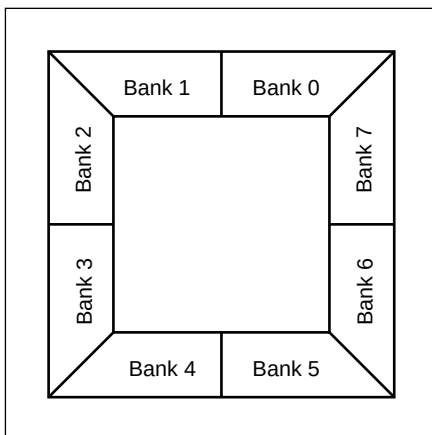


ug002_c2_014_112900

Figure 15: Virtex-II Pro I/O Banks: Top View for Wire-Bond Packages (CS, FG, and BG)

Within a bank, output standards can be mixed only if they use the same V_{CCO} . Compatible standards are shown in Table 7. GTL and GTLP appear under all voltages because their open-drain outputs do not depend on V_{CCO} .

Some input standards require a user-supplied threshold voltage, V_{REF} . In this case, certain user-I/O pins are automatically configured as inputs for the V_{REF} voltage. Approximately one in six of the I/O pins in the bank assume this role.



ds031_66_112900

Figure 16: Virtex-II Pro I/O Banks: Top View for Flip-Chip Packages (FF and BF)

V_{REF} pins within a bank are interconnected internally, and consequently only one V_{REF} voltage can be used within each bank. However, for correct operation, all V_{REF} pins in the bank must be connected to the external reference voltage source.

The V_{CCO} and the V_{REF} pins for each bank appear in the device pinout tables. Within a given package, the number of V_{REF} and V_{CCO} pins can vary depending on the size of

device. In larger devices, more I/O pins convert to V_{REF} pins. Since these are always a superset of the V_{REF} pins used for smaller devices, it is possible to design a PCB that permits migration to a larger device if necessary.

Table 7: Compatible Output Standards

V_{CCO}	Compatible Standards ⁽¹⁾
3.3V ⁽²⁾	PCI ⁽³⁾ , LVTTTL, SSTL3 (I & II), LVCMOS33, LVDCI_33, SSTL3_DCI (I & II) ⁽¹⁾
2.5V	SSTL2 (I & II), LVCMOS25, GTL, GTLP, LVDS_25, LVDSEXT_25, LVDCI_25, LVDCI_DV2_25, SSTL2_DCI (I & II), LDT, ULVDS, BLVDS
1.8V	HSTL (I, II, III, & IV), HSTL_DCI (I,II, III & IV), LVCMOS18, GTL, GTLP, LVDCI_18, LVDCI_DV2_18
1.5V	HSTL (I, II, III, & IV), HSTL_DCI (I,II, III & IV), LVCMOS15, GTL, GTLP, LVDCI_15, LVDCI_DV2_15, GTLP_DCI
1.2V	GTL_DCI

Notes:

1. LVPECL, LVDS_33, LVDSEXT_33, and AGP-2X are not supported.
2. Perfect impedance matching is required for 3.3V standards.
3. For optimum performance, it is recommended that PCI be used in conjunction with LVDCI_33. Contact Xilinx for more details.

All V_{REF} pins for the largest device anticipated must be connected to the V_{REF} voltage and not used for I/O. In smaller devices, some V_{CCO} pins used in larger devices do not connect within the package. These unconnected pins can be left unconnected externally, or, if necessary, they can be connected to the V_{CCO} voltage to permit migration to a larger device.

Digitally Controlled Impedance (DCI)

Today's chip output signals with fast edge rates require termination to prevent reflections and maintain signal integrity. High pin count packages (especially ball grid arrays) can not accommodate external termination resistors.

Virtex-II Pro DCI provides controlled impedance drivers and on-chip termination for single-ended I/Os. This eliminates the need for external resistors, and improves signal integrity. The DCI feature can be used on any IOB by selecting one of the DCI I/O standards.

When applied to inputs, DCI provides input parallel termination. When applied to outputs, DCI provides controlled impedance drivers (series termination) or output parallel termination.

DCI operates independently on each I/O bank. When a DCI I/O standard is used in a particular I/O bank, external reference resistors must be connected to two dual-function pins

on the bank. These resistors, voltage reference of N transistor (VRN) and the voltage reference of P transistor (VRP) are shown in Figure 17.

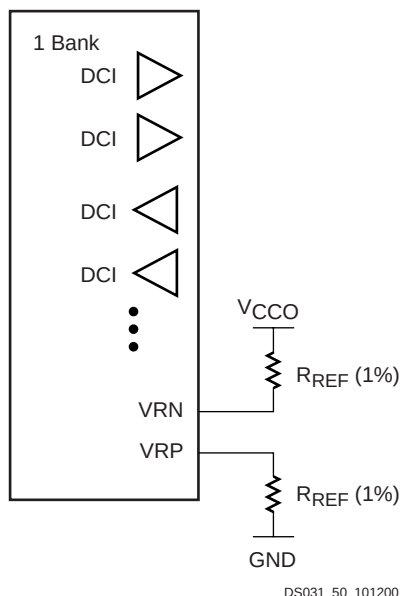


Figure 17: DCI in a Virtex-II Pro Bank

When used with a terminated I/O standard, the value of the resistors are specified by the standard (typically 50Ω). When used with a controlled impedance driver, the resistors set the output impedance of the driver within the specified range (20Ω to 100Ω). For all series and parallel terminations listed in Table 8 and Table 9, the reference resistors must have the same value for any given bank. One percent resistors are recommended.

The DCI system adjusts the I/O impedance to match the two external reference resistors, or half of the reference resistors, and compensates for impedance changes due to voltage and/or temperature fluctuations. The adjustment is done by turning parallel transistors in the IOB on or off.

Controlled Impedance Drivers (Series Termination)

DCI can be used to provide a buffer with a controlled output impedance. It is desirable for this output impedance to match the transmission line impedance (Z_0). Virtex-II Pro input buffers also support LVDCI and LVDCI_DV2 I/O standards.

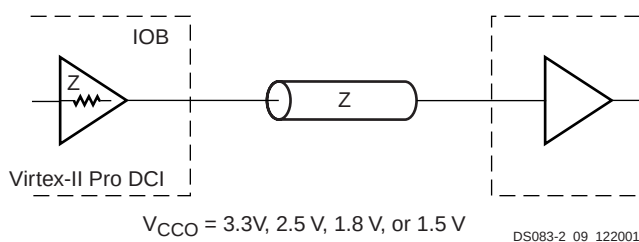


Figure 18: Internal Series Termination

Table 8: SelectI/O Controlled Impedance Buffers

V _{CCO}	DCI	DCI Half Impedance
3.3V	LVDCI_33	N/A
2.5V	LVDCI_25	LVDCI_DV2_25
1.8V	LVDCI_18	LVDCI_DV2_18
1.5V	LVDCI_15	LVDCI_DV2_15

Controlled Impedance Terminations (Parallel Termination)

DCI also provides on-chip termination for SSTL3, SSTL2, HSTL (Class I, II, III, or IV), and GTL/GTLP receivers or transmitters on bidirectional lines.

Table 9 lists the on-chip parallel terminations available in Virtex-II Pro devices. V_{CCO} must be set according to Table 5. Note that there is a V_{CCO} requirement for GTL_DCI and GTLP_DCI, due to the on-chip termination resistor.

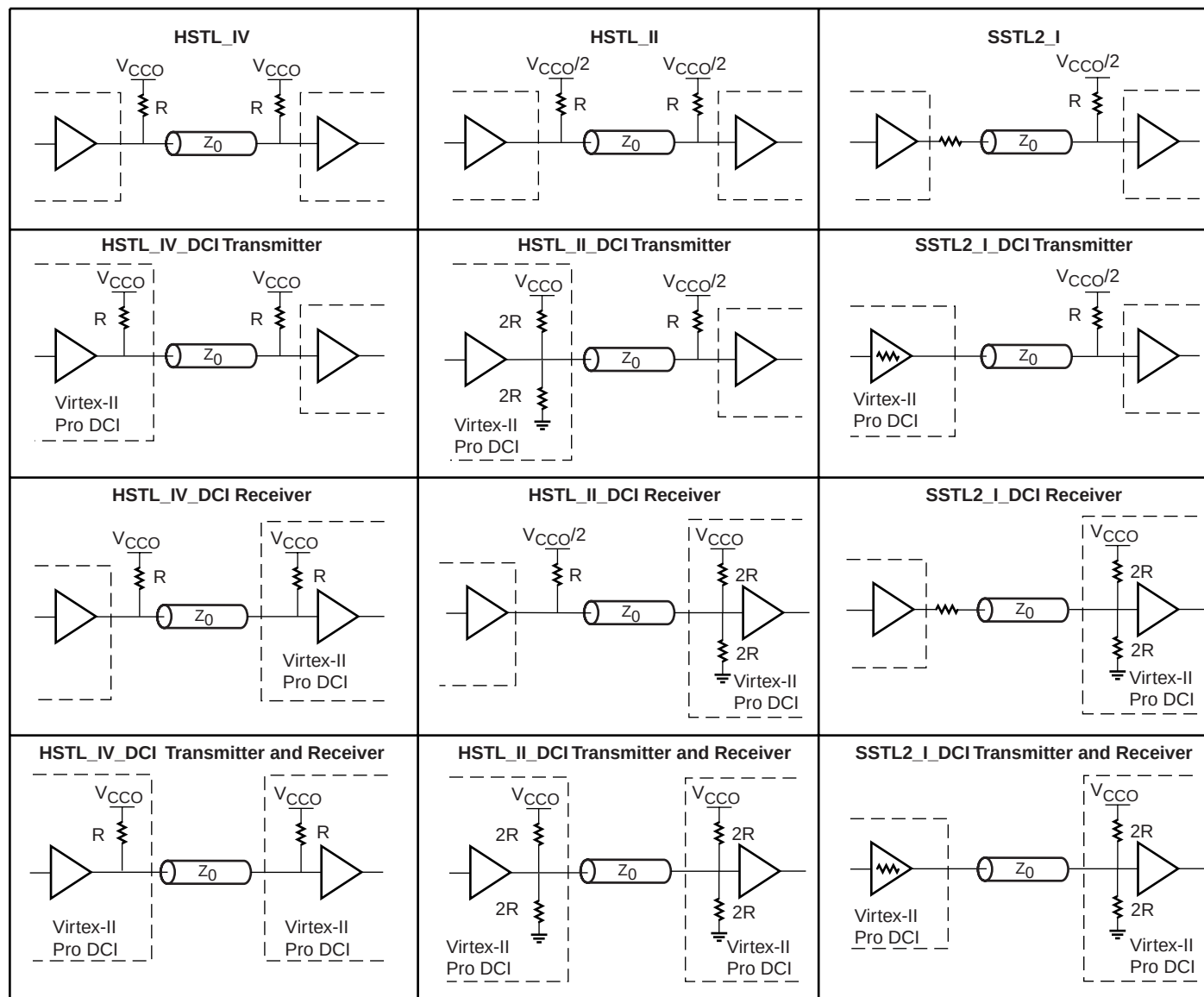
Table 9: SelectI/O Buffers With On-Chip Parallel Termination

I/O Standard	External Termination	On-Chip Termination
SSTL3 Class I	SSTL3_I	SSTL3_I_DCI ⁽¹⁾
SSTL3 Class II	SSTL3_II	SSTL3_II_DCI ⁽¹⁾
SSTL2 Class I	SSTL2_I	SSTL2_I_DCI ⁽¹⁾
SSTL2 Class II	SSTL2_II	SSTL2_II_DCI ⁽¹⁾
HSTL Class I	HSTL_I	HSTL_I_DCI
	HSTL_I_18	HSTL_I_DCI_18
HSTL Class II	HSTL_II	HSTL_II_DCI
	HSTL_II_18	HSTL_II_DCI_18
HSTL Class III	HSTL_III	HSTL_III_DCI
	HSTL_III_18	HSTL_III_DCI_18
HSTL Class IV	HSTL_IV	HSTL_IV_DCI
	HSTL_IV_18	HSTL_IV_DCI_18
GTL	GTL	GTL_DCI
GTLP	GTLP	GTLP_DCI

Notes:

1. SSTL Compatible

Figure 19 provides examples illustrating the use of the HSTL_IV_DCI, HSTL_II_DCI, and SSTL2_I_DCI I/O standards.



DS083-2_08_122001

Figure 19: DCI Usage Examples

Configurable Logic Blocks (CLBs)

The Virtex-II Pro configurable logic blocks (CLB) are organized in an array and are used to build combinational and synchronous logic designs. Each CLB element is tied to a switch matrix to access the general routing matrix, as shown in Figure 20. A CLB element comprises 4 similar slices, with fast local feedback within the CLB. The four slices are split in two columns of two slices with two independent carry logic chains and one common shift chain.

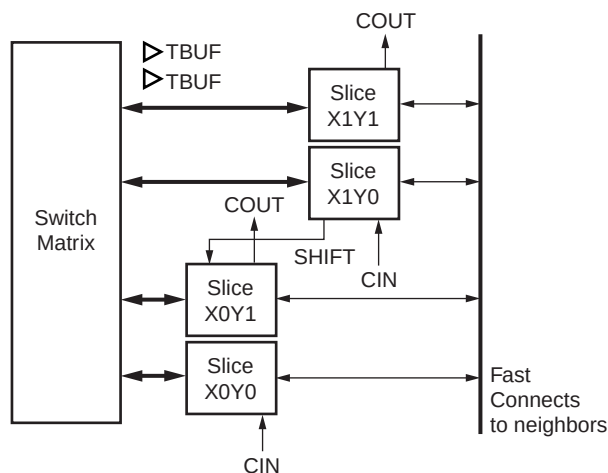


Figure 20: Virtex-II Pro CLB Element

Slice Description

Each slice includes two 4-input function generators, carry logic, arithmetic logic gates, wide function multiplexers and two storage elements. As shown in Figure 21, each 4-input function generator is programmable as a 4-input LUT, 16 bits of distributed SelectRAM memory, or a 16-bit variable-tap shift register element.

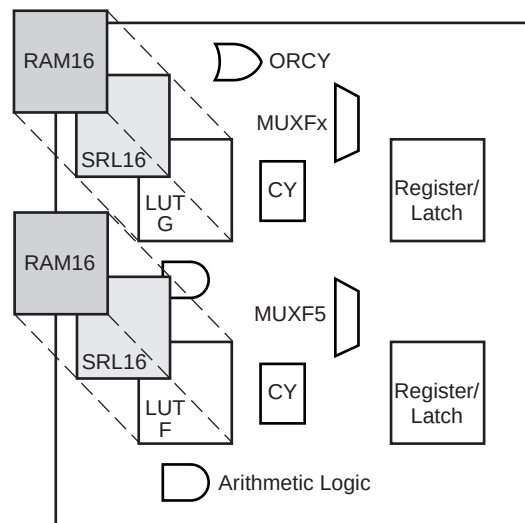
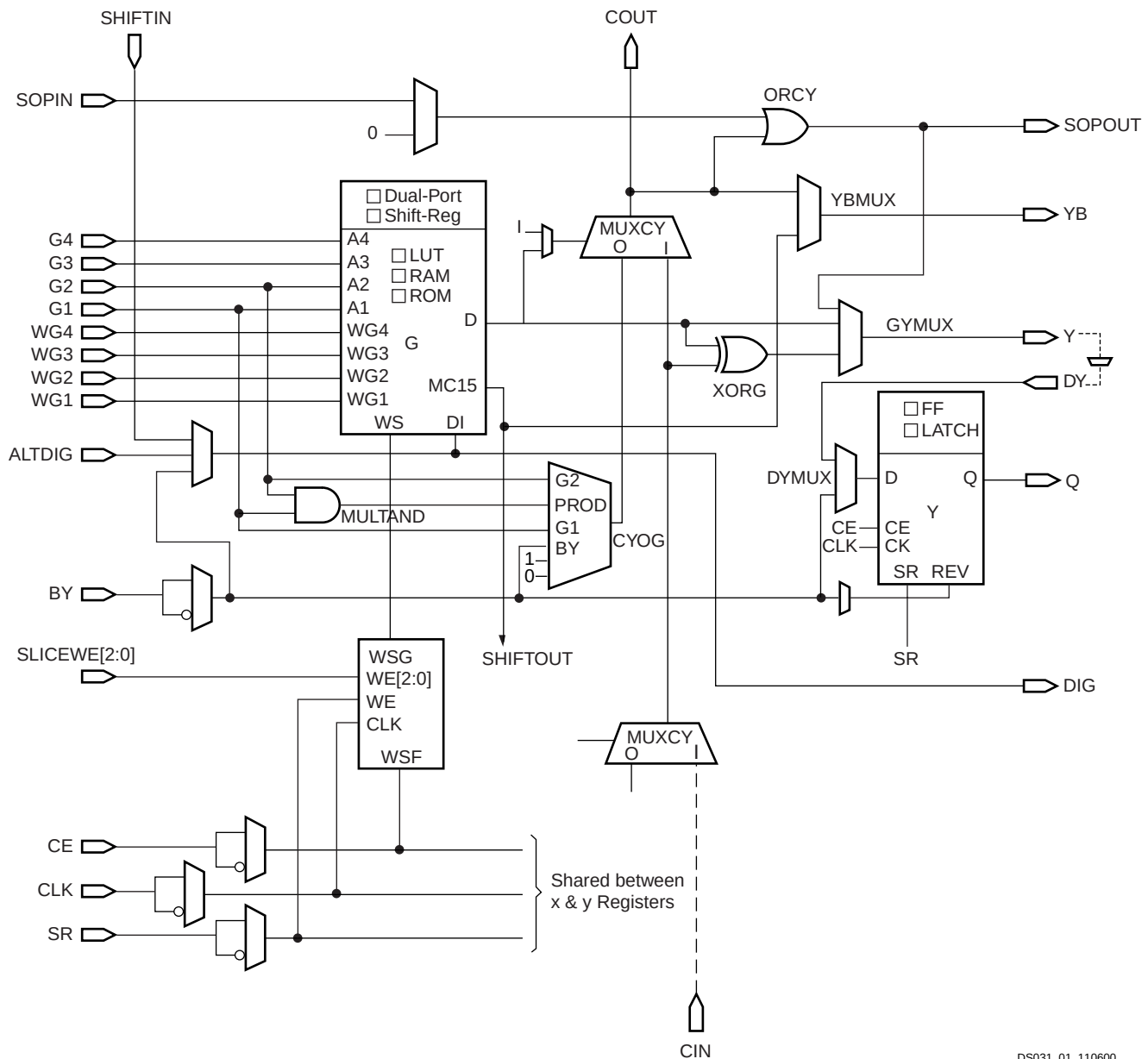


Figure 21: Virtex-II Pro Slice Configuration

The output from the function generator in each slice drives both the slice output and the D input of the storage element. **Figure 22** shows a more detailed view of a single slice.



DS031_01_110600

Figure 22: Virtex-II Pro Slice (Top Half)

Configurations

Look-Up Table

Virtex-II Pro function generators are implemented as 4-input look-up tables (LUTs). Four independent inputs are provided to each of the two function generators in a slice (F and G). These function generators are each capable of implementing any arbitrarily defined boolean function of four inputs. The propagation delay is therefore independent of the function implemented. Signals from the function generators can exit the slice (X or Y output), can input the XOR dedicated gate (see arithmetic logic), or input the carry-logic multiplexer (see fast look-ahead carry logic), or feed the D

input of the storage element, or go to the MUXF5 (not shown in **Figure 22**).

In addition to the basic LUTs, the Virtex-II Pro slice contains logic (MUXF5 and MUXFX multiplexers) that combines function generators to provide any function of five, six, seven, or eight inputs. The MUXFX is either MUXF6, MUXF7, or MUXF8 according to the slice considered in the CLB. Selected functions up to nine inputs (MUXF5 multiplexer) can be implemented in one slice. The MUXFX can also be a MUXF6, MUXF7, or MUXF8 multiplexer to map any function of six, seven, or eight inputs and selected wide logic functions.

Register/Latch

The storage elements in a Virtex-II Pro slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D input can be directly driven by the X or Y output via the DX or DY input, or by the slice inputs bypassing the function generators via the BX or BY input. The clock enable signal (CE) is active High by default. If left unconnected, the clock enable for that storage element defaults to the active state.

In addition to clock (CK) and clock enable (CE) signals, each slice has set and reset signals (SR and BY slice inputs). SR forces the storage element into the state specified by the attribute SRHIGH or SRLOW. SRHIGH forces a logic 1 when SR is asserted. SRLOW forces a logic 0. When SR is used, an optional second input (BY) forces the storage element into the opposite state via the REV pin. The reset condition is predominant over the set condition. (See [Figure 23.](#))

The initial state after configuration or global initial state is defined by a separate INIT0 and INIT1 attribute. By default, setting the SRLOW attribute sets INIT0, and setting the SRHIGH attribute sets INIT1.

For each slice, set and reset can be set to be synchronous or asynchronous. Virtex-II Pro devices also have the ability to set INIT0 and INIT1 independent of SRHIGH and SRLOW.

The control signals clock (CLK), clock enable (CE) and set/reset (SR) are common to both storage elements in one slice. All of the control signals have independent polarity. Any inverter placed on a control input is automatically absorbed.

The set and reset functionality of a register or a latch can be configured as follows:

- No set or reset
- Synchronous set
- Synchronous reset
- Synchronous set and reset
- Asynchronous set (preset)
- Asynchronous reset (clear)
- Asynchronous set and reset (preset and clear)

The synchronous reset has precedence over a set, and an asynchronous clear has precedence over a preset.

Distributed SelectRAM Memory

Each function generator (LUT) can implement a 16 x 1-bit synchronous RAM resource called a distributed SelectRAM element. The SelectRAM elements are configurable within a CLB to implement the following:

- Single-Port 16 x 8-bit RAM
- Single-Port 32 x 4-bit RAM
- Single-Port 64 x 2-bit RAM
- Single-Port 128 x 1-bit RAM
- Dual-Port 16 x 4-bit RAM
- Dual-Port 32 x 2-bit RAM
- Dual-Port 64 x 1-bit RAM

Distributed SelectRAM memory modules are synchronous (write) resources. The combinatorial read access time is extremely fast, while the synchronous write simplifies high-speed designs. A synchronous read can be implemented with a storage element in the same slice. The distributed SelectRAM memory and the storage element share the same clock input. A Write Enable (WE) input is active High, and is driven by the SR input.

[Table 10](#) shows the number of LUTs (2 per slice) occupied by each distributed SelectRAM configuration.

Table 10: Distributed SelectRAM Configurations

RAM	Number of LUTs
16 x 1S	1
16 x 1D	2
32 x 1S	2
32 x 1D	4
64 x 1S	4
64 x 1D	8
128 x 1S	8

Notes:
1. S = single-port configuration; D = dual-port configuration

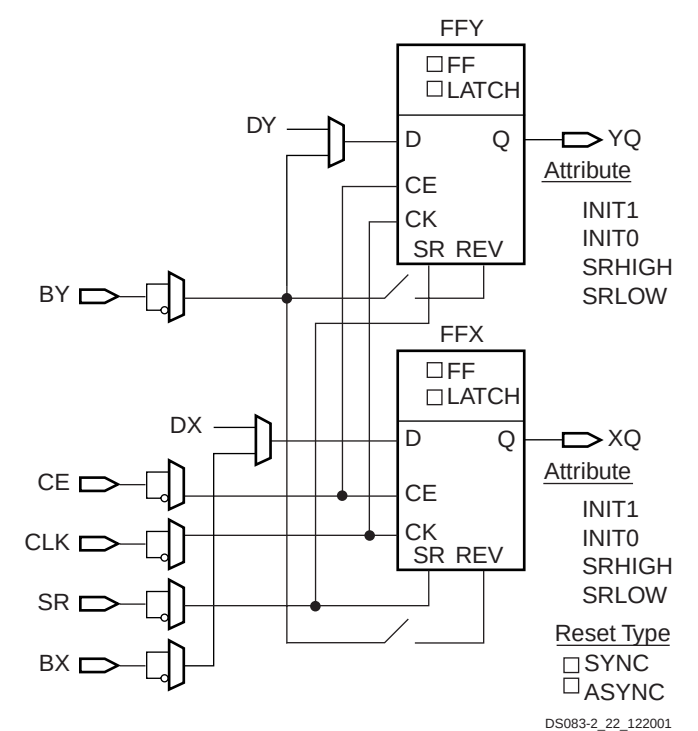


Figure 23: Register / Latch Configuration in a Slice

For single-port configurations, distributed SelectRAM memory has one address port for synchronous writes and asynchronous reads.

For dual-port configurations, distributed SelectRAM memory has one port for synchronous writes and asynchronous reads and another port for asynchronous reads. The function generator (LUT) has separated read address inputs (A1, A2, A3, A4) and write address inputs (WG1/WF1, WG2/WF2, WG3/WF3, WG4/WF4).

In single-port mode, read and write addresses share the same address bus. In dual-port mode, one function generator (R/W port) is connected with shared read and write addresses. The second function generator has the A inputs (read) connected to the second read-only port address and the W inputs (write) shared with the first read/write port address.

Figure 24, Figure 25, and Figure 26 illustrate various example configurations.

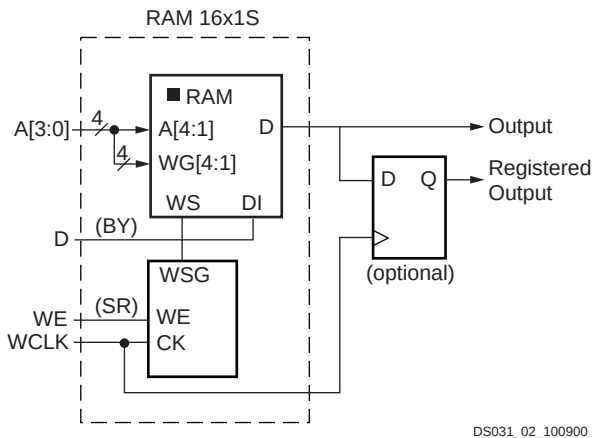


Figure 24: Distributed SelectRAM (RAM16x1S)

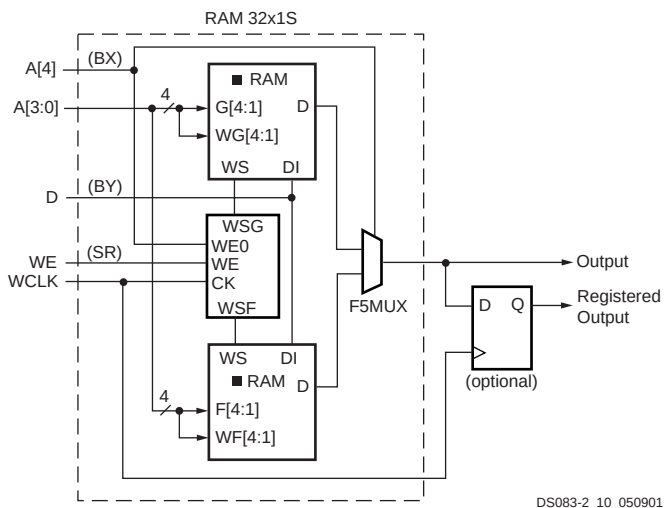


Figure 25: Single-Port Distributed SelectRAM (RAM32x1S)

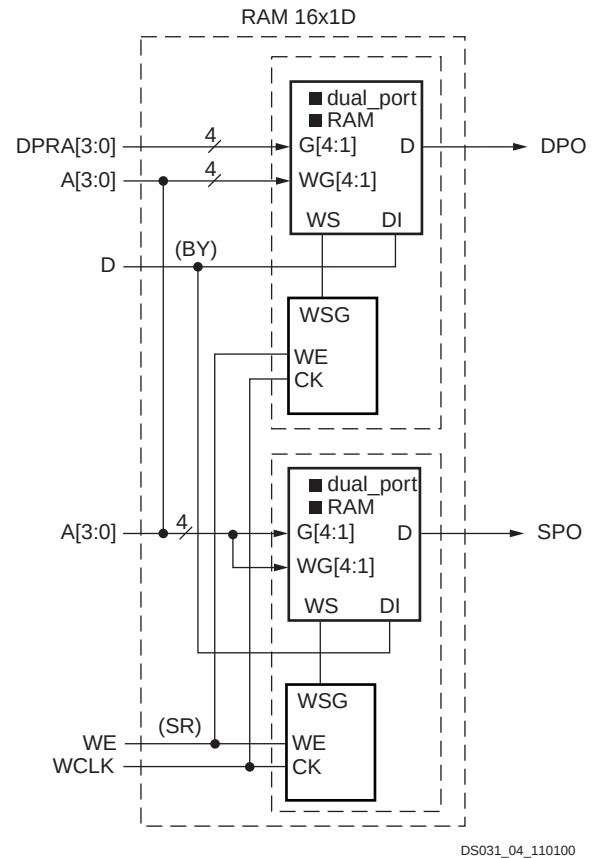


Figure 26: Dual-Port Distributed SelectRAM (RAM16x1D)

Similar to the RAM configuration, each function generator (LUT) can implement a 16 x 1-bit ROM. Five configurations are available: ROM16x1, ROM32x1, ROM64x1, ROM128x1, and ROM256x1. The ROM elements are cascadable to implement wider or/and deeper ROM. ROM contents are loaded at configuration. Table 11 shows the number of LUTs occupied by each configuration.

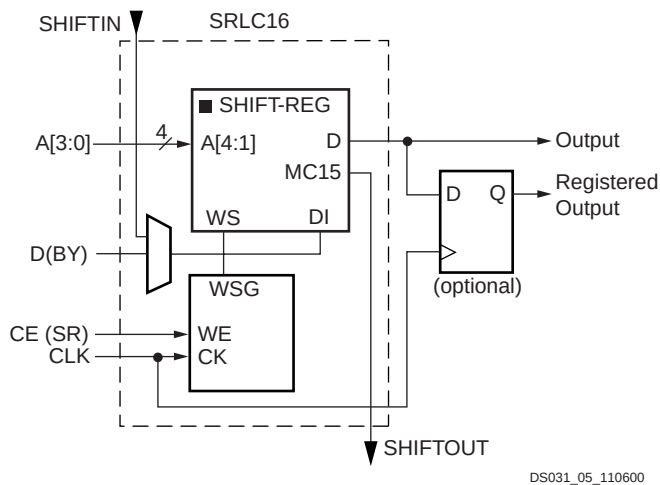
Table 11: ROM Configuration

ROM	Number of LUTs
16 x 1	1
32 x 1	2
64 x 1	4
128 x 1	8 (1 CLB)
256 x 1	16 (2 CLBs)

Shift Registers

Each function generator can also be configured as a 16-bit shift register. The write operation is synchronous with a clock input (CLK) and an optional clock enable, as shown in **Figure 27**. A dynamic read access is performed through the 4-bit address bus, A[3:0]. The configurable 16-bit shift regis-

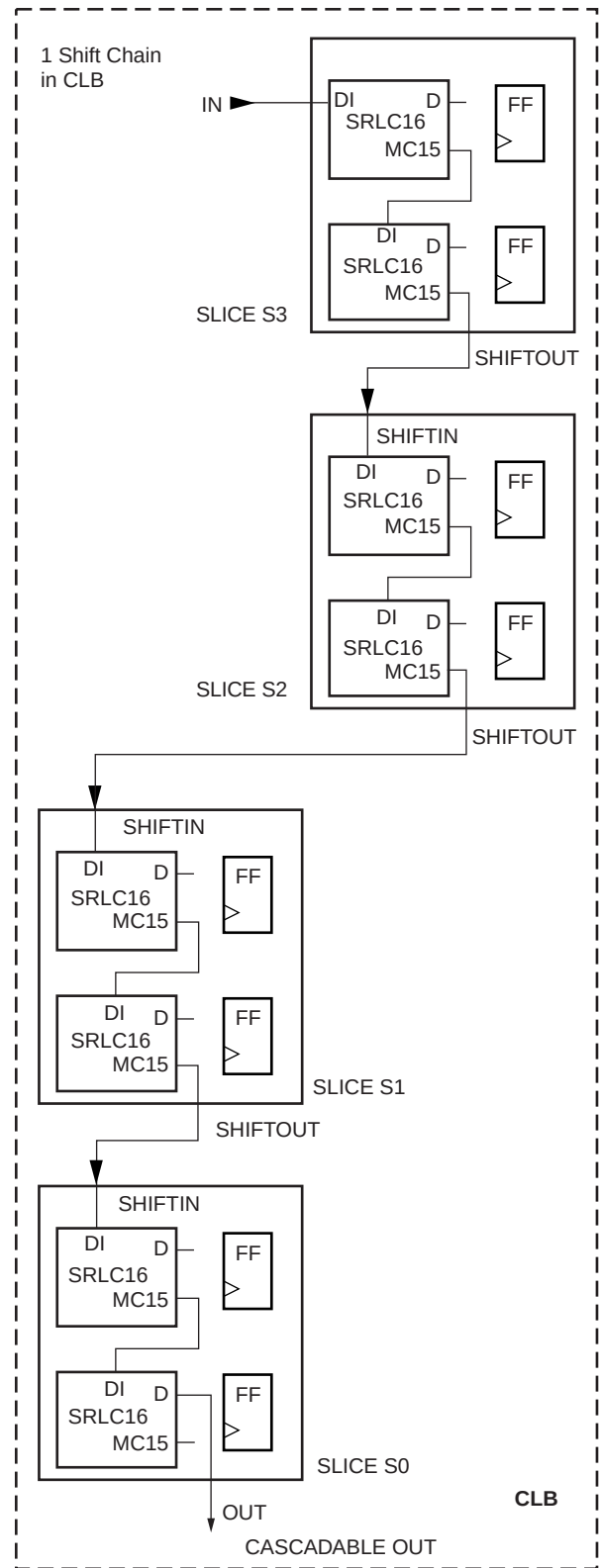
ter cannot be set or reset. The read is asynchronous; however, the storage element or flip-flop is available to implement a synchronous read. Any of the 16 bits can be read out asynchronously by varying the address. The storage element should always be used with a constant address. For example, when building an 8-bit shift register and configuring the addresses to point to the 7th bit, the 8th bit can be the flip-flop. The overall system performance is improved by using the superior clock-to-out of the flip-flops.



DS031_05_110600

Figure 27: Shift Register Configurations

An additional dedicated connection between shift registers allows connecting the last bit of one shift register to the first bit of the next, without using the ordinary LUT output. (See [Figure 28](#).) Longer shift registers can be built with dynamic access to any bit in the chain. The shift register chaining and the MUXF5, MUXF6, and MUXF7 multiplexers allow up to a 128-bit shift register with addressable access to be implemented in one CLB.



DS031_06_110200

Figure 28: Cascadable Shift Register

Multiplexers

Virtex-II Pro function generators and associated multiplexers can implement the following:

- 4:1 multiplexer in one slice
- 8:1 multiplexer in two slices
- 16:1 multiplexer in one CLB element (4 slices)
- 32:1 multiplexer in two CLB elements (8 slices)

Each Virtex-II Pro slice has one MUXF5 multiplexer and one MUXFX multiplexer. The MUXFX multiplexer implements the MUXF6, MUXF7, or MUXF8, as shown in **Figure 29**. Each CLB element has two MUXF6 multiplexers, one MUXF7 multiplexer and one MUXF8 multiplexer. Examples of multiplexers are shown in the *Virtex-II Pro User Guide*. Any LUT can implement a 2:1 multiplexer.

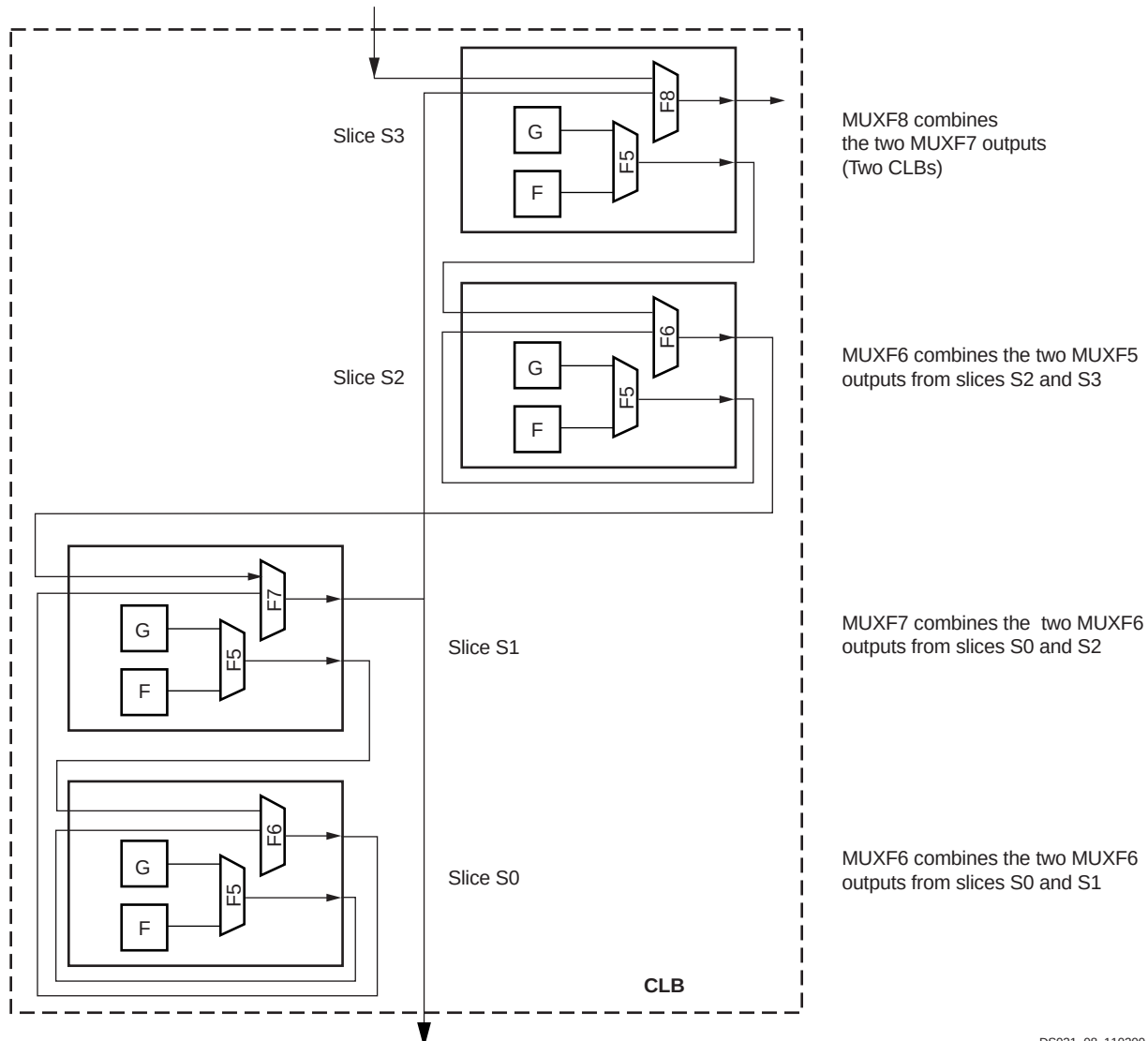


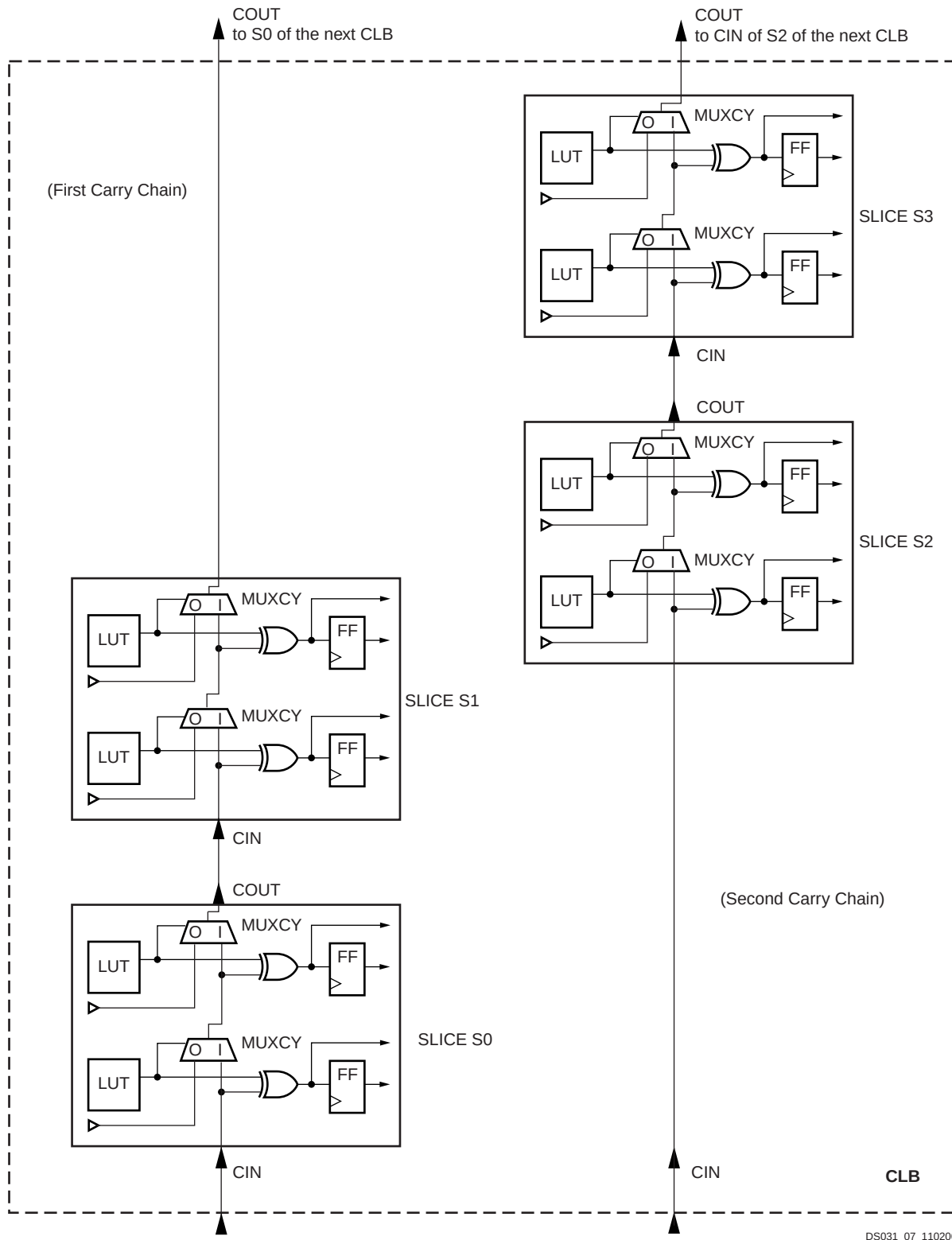
Figure 29: MUXF5 and MUXFX multiplexers

DS031_08_110200

Fast Lookahead Carry Logic

Dedicated carry logic provides fast arithmetic addition and subtraction. The Virtex-II Pro CLB has two separate carry chains, as shown in the **Figure 30**.

The height of the carry chains is two bits per slice. The carry chain in the Virtex-II Pro device is running upward. The dedicated carry path and carry multiplexer (MUXCY) can also be used to cascade function generators for implementing wide logic functions.



DS031_07_110200

Figure 30: Fast Carry Logic Path

Arithmetic Logic

The arithmetic logic includes an XOR gate that allows a 2-bit full adder to be implemented within a slice. In addition,

a dedicated AND (MULT_AND) gate (shown in [Figure 22](#)) improves the efficiency of multiplier implementation.

Sum of Products

Each Virtex-II Pro slice has a dedicated OR gate named ORCY, ORing together outputs from the slices carryout and the ORCY from an adjacent slice. The ORCY gate with the dedicated Sum of Products (SOP) chain are designed for

implementing large, flexible SOP chains. One input of each ORCY is connected through the fast SOP chain to the output of the previous ORCY in the same slice row. The second input is connected to the output of the top MUXCY in the same slice, as shown in [Figure 31](#).

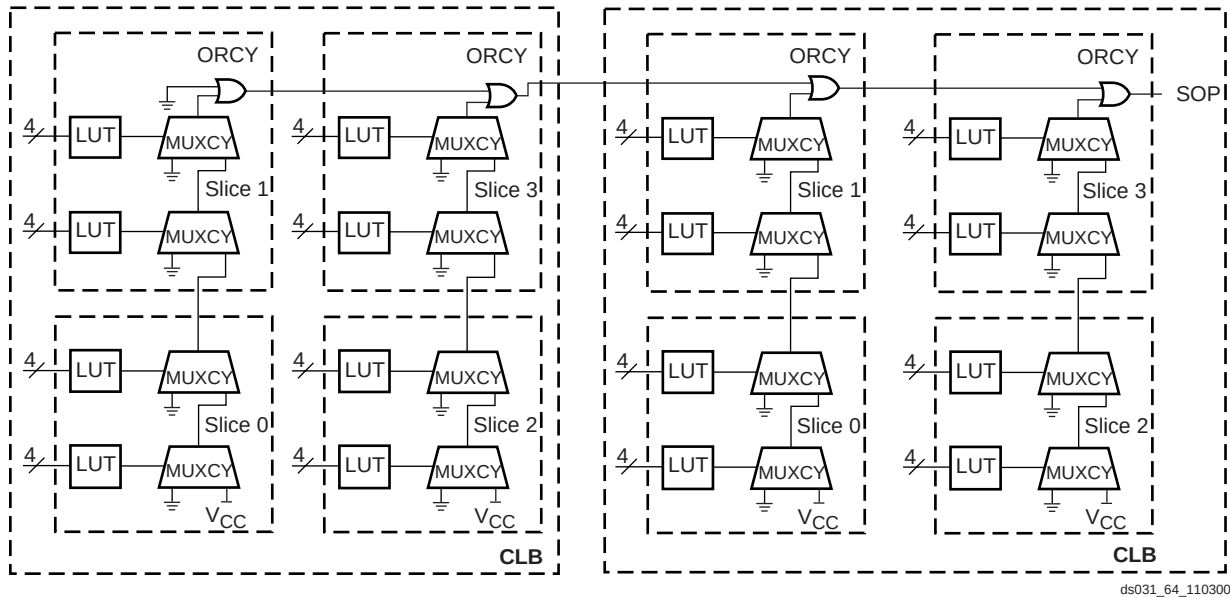


Figure 31: Horizontal Cascade Chain

LUTs and MUXCYs can implement large AND gates or other combinatorial logic functions. **Figure 32** illustrates

LUT and MUXCY resources configured as a 16-input AND gate.

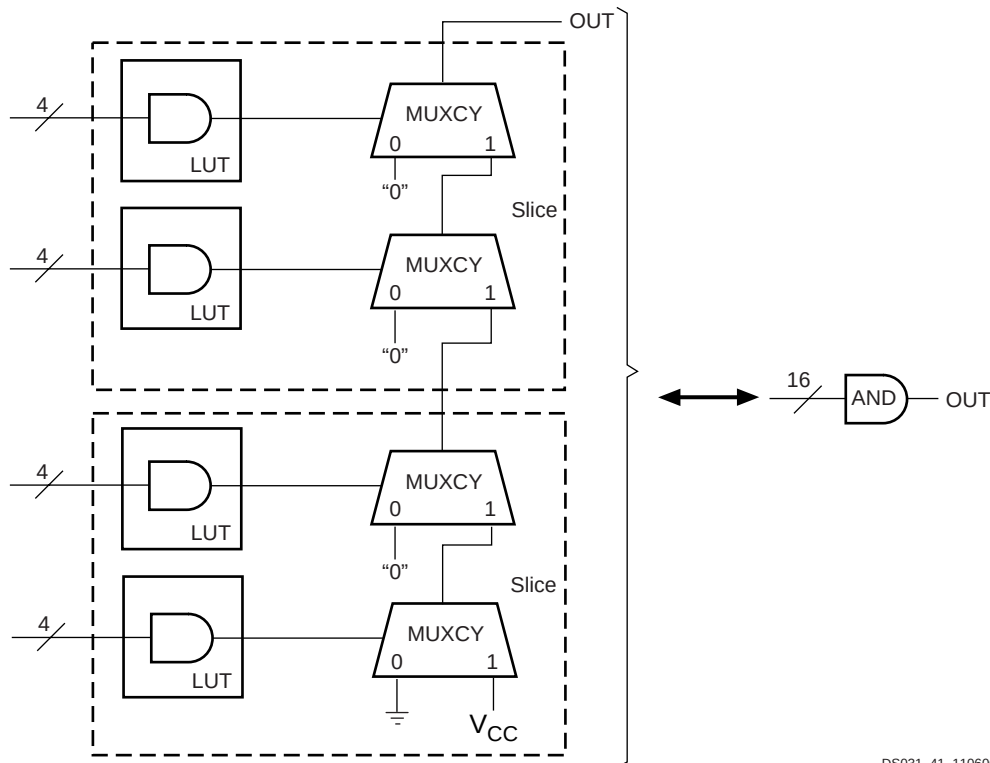


Figure 32: Wide-Input AND Gate (16 Inputs)

3-State Buffers

Introduction

Each Virtex-II Pro CLB contains two 3-state drivers (TBUFs) that can drive on-chip buses. Each 3-state buffer has its own 3-state control pin and its own input pin.

Each of the four slices have access to the two 3-state buffers through the switch matrix, as shown in Figure 33. TBUFs in neighboring CLBs can access slice outputs by direct connects. The outputs of the 3-state buffers drive horizontal routing resources used to implement 3-state buses.

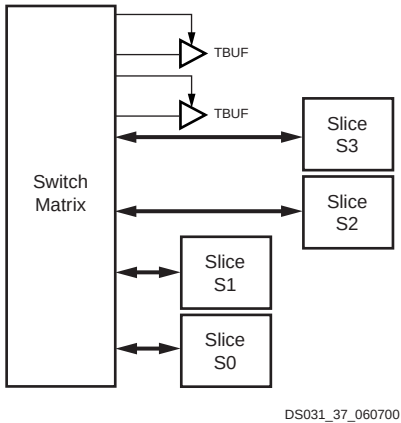


Figure 33: Virtex-II Pro 3-State Buffers

The 3-state buffer logic is implemented using AND-OR logic rather than 3-state drivers, so that timing is more predictable and less load dependant especially with larger devices.

Locations / Organization

Four horizontal routing resources per CLB are provided for on-chip 3-state buses. Each 3-state buffer has access alternately to two horizontal lines, which can be partitioned as shown in Figure 34. The switch matrices corresponding to SelectRAM memory and multiplier or I/O blocks are skipped.

Number of 3-State Buffers

Table 12 shows the number of 3-state buffers available in each Virtex-II Pro device. The number of 3-state buffers is twice the number of CLB elements.

Table 12: Virtex-II Pro 3-State Buffers

Device	3-State Buffers per Row	Total Number of 3-State Buffers
XC2VP2	44	704
XC2VP4	44	1,760
XC2VP7	68	2,720
XC2VP20	92	5,152
XC2VP50	140	12,320

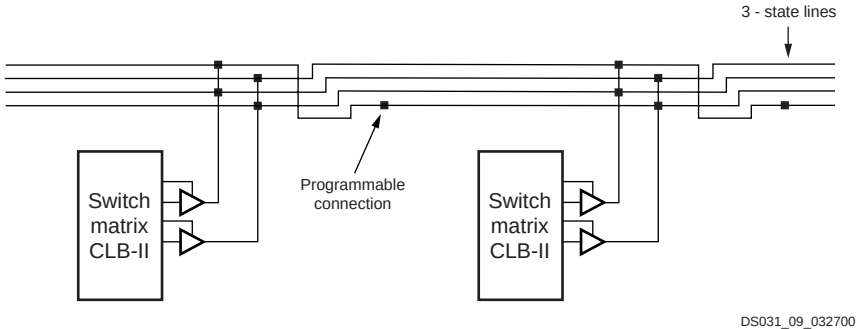


Figure 34: 3-State Buffer Connection to Horizontal Lines

CLB/Slice Configurations

Table 13 summarizes the logic resources in one CLB. All of the CLBs are identical and each CLB or slice can be imple-

mented in one of the configurations listed. Table 14 shows the available resources in all CLBs.

Table 13: Logic Resources in One CLB

Slices	LUTs	Flip-Flops	MULT_ANDs	Arithmetic & Carry-Chains	SOP Chains	Distributed SelectRAM	Shift Registers	TBUF
4	8	8	8	2	2	128 bits	128 bits	2

Table 14: Virtex-II Pro Logic Resources Available in All CLBs

Device	CLB Array: Row x Column	Number of Slices	Number of LUTs	Max Distributed SelectRAM or Shift Register (bits)	Number of Flip-Flops	Number of Carry Chains ⁽¹⁾	Number of SOP Chains ⁽¹⁾
XC2VP2	16 x 22	1,408	2,816	45,056	2,816	44	32
XC2VP4	40 x 22	3,008	6,016	96,256	6,016	44	80
XC2VP7	40 x 34	4,928	9,856	157,696	9,856	68	80
XC2VP20	56 x 46	9,280	18,560	296,960	18,560	92	112
XC2VP50	88 x 70	22,592	45,184	722,944	45,184	140	176

Notes:

1. The carry-chains and SOP chains can be split or cascaded.

18 Kb Block SelectRAM Resources

Introduction

Virtex-II Pro devices incorporate large amounts of 18 Kb block SelectRAM. These complement the distributed SelectRAM resources that provide shallow RAM structures implemented in CLBs. Each Virtex-II Pro block SelectRAM is an 18 Kb true dual-port RAM with two independently clocked and independently controlled synchronous ports that access a common storage area. Both ports are functionally identical. CLK, EN, WE, and SSR polarities are defined through configuration.

Each port has the following types of inputs: Clock and Clock Enable, Write Enable, Set/Reset, and Address, as well as separate Data/parity data inputs (for write) and Data/parity data outputs (for read).

Operation is synchronous; the block SelectRAM behaves like a register. Control, address and data inputs must (and need only) be valid during the set-up time window prior to a rising (or falling, a configuration option) clock edge. Data outputs change as a result of the same clock edge.

Configuration

The Virtex-II Pro block SelectRAM supports various configurations, including single- and dual-port RAM and various data/address aspect ratios. Supported memory configurations for single- and dual-port modes are shown in Table 15.

Table 15: Dual- and Single-Port Configurations

16K x 1 bit	2K x 9 bits
8K x 2 bits	1K x 18 bits
4K x 4 bits	512 x 36 bits

Single-Port Configuration

As a single-port RAM, the block SelectRAM has access to the 18 Kb memory locations in any of the 2K x 9-bit,

1K x 18-bit, or 512 x 36-bit configurations and to 16 Kb memory locations in any of the 16K x 1-bit, 8K x 2-bit, or 4K x 4-bit configurations. The advantage of the 9-bit, 18-bit and 36-bit widths is the ability to store a parity bit for each eight bits. Parity bits must be generated or checked externally in user logic. In such cases, the width is viewed as 8 + 1, 16 + 2, or 32 + 4. These extra parity bits are stored and behave exactly as the other bits, including the timing parameters. Video applications can use the 9-bit ratio of Virtex-II Pro block SelectRAM memory to advantage.

Each block SelectRAM cell is a fully synchronous memory as illustrated in Figure 35. Input data bus and output data bus widths are identical.

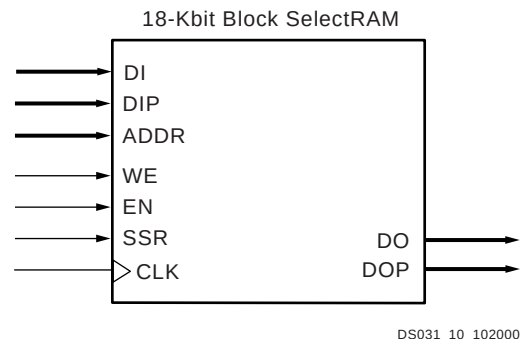


Figure 35: 18 Kb Block SelectRAM Memory in Single-Port Mode

Dual-Port Configuration

As a dual-port RAM, each port of block SelectRAM has access to a common 18 Kb memory resource. These are fully synchronous ports with independent control signals for each port. The data widths of the two ports can be configured independently, providing built-in bus-width conversion.

Table 16 illustrates the different configurations available on ports A and B.

Table 16: Dual-Port Mode Configurations

Port A	16K x 1	16K x 1	16K x 1	16K x 1	16K x 1	16K x 1
Port B	16K x 1	8K x 2	4K x 4	2K x 9	1K x 18	512 x 36
Port A	8K x 2	8K x 2	8K x 2	8K x 2	8K x 2	
Port B	8K x 2	4K x 4	2K x 9	1K x 18	512 x 36	
Port A	4K x 4	4K x 4	4K x 4	4K x 4		
Port B	4K x 4	2K x 9	1K x 18	512 x 36		
Port A	2K x 9	2K x 9	2K x 9			
Port B	2K x 9	1K x 18	512 x 36			
Port A	1K x 18	1K x 18				
Port B	1K x 18	512 x 36				
Port A	512 x 36					
Port B	512 x 36					

If both ports are configured in either 2K x 9-bit, 1K x 18-bit, or 512 x 36-bit configurations, the 18 Kb block is accessible from port A or B. If both ports are configured in either 16K x 1-bit, 8K x 2-bit, or 4K x 4-bit configurations, the 16 K-bit block is accessible from Port A or Port B. All other configurations result in one port having access to an 18 Kb memory block and the other port having access to a 16 K-bit subset of the memory block equal to 16 Kbs.

includes dedicated routing resources to provide an efficient interface with CLBs, block SelectRAM, and multipliers.

Table 17: 18 Kb Block SelectRAM Port Aspect Ratio

Width	Depth	Address Bus	Data Bus	Parity Bus
1	16,384	ADDR[13:0]	DATA[0]	N/A
2	8,192	ADDR[12:0]	DATA[1:0]	N/A
4	4,096	ADDR[11:0]	DATA[3:0]	N/A
9	2,048	ADDR[10:0]	DATA[7:0]	Parity[0]
18	1,024	ADDR[9:0]	DATA[15:0]	Parity[1:0]
36	512	ADDR[8:0]	DATA[31:0]	Parity[3:0]

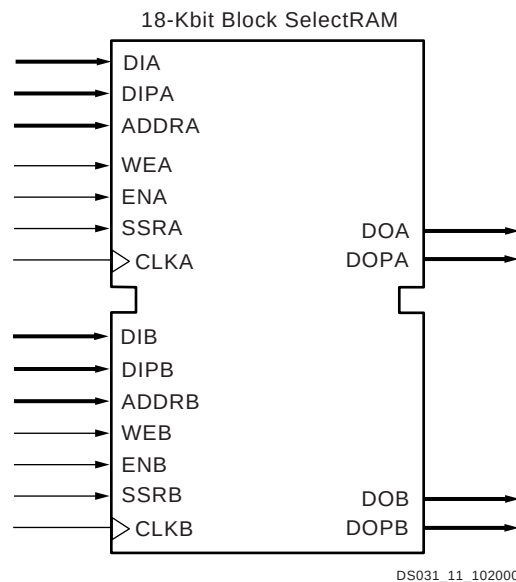


Figure 36: 18 Kb Block SelectRAM in Dual-Port Mode

Each block SelectRAM cell is a fully synchronous memory, as illustrated in Figure 36. The two ports have independent inputs and outputs and are independently clocked.

Port Aspect Ratios

Table 17 shows the depth and the width aspect ratios for the 18 Kb block SelectRAM. Virtex-II Pro block SelectRAM also

Read/Write Operations

The Virtex-II Pro block SelectRAM read operation is fully synchronous. An address is presented, and the read operation is enabled by control signal ENA or ENB. Then, depending on clock polarity, a rising or falling clock edge causes the stored data to be loaded into output registers.

The write operation is also fully synchronous. Data and address are presented, and the write operation is enabled by control signals WEA and WEB in addition to ENA or ENB. Then, again depending on the clock input mode, a rising or falling clock edge causes the data to be loaded into the memory cell addressed.

A write operation performs a simultaneous read operation. Three different options are available, selected by configuration:

1. WRITE_FIRST

The WRITE_FIRST option is a transparent mode. The same clock edge that writes the data input (DI) into the

memory also transfers DI into the output registers DO, as shown in [Figure 37](#).

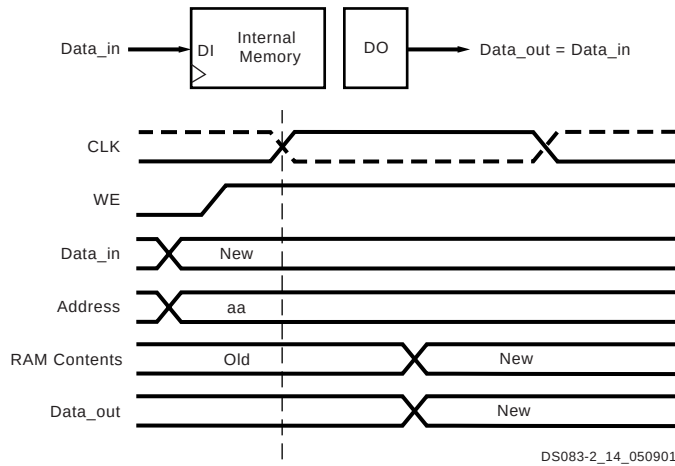


Figure 37: **WRITE_FIRST** Mode

2. READ_FIRST

The READ_FIRST option is a read-before-write mode.

The same clock edge that writes data input (DI) into the memory also transfers the prior content of the memory cell addressed into the data output registers DO, as shown in [Figure 38](#).

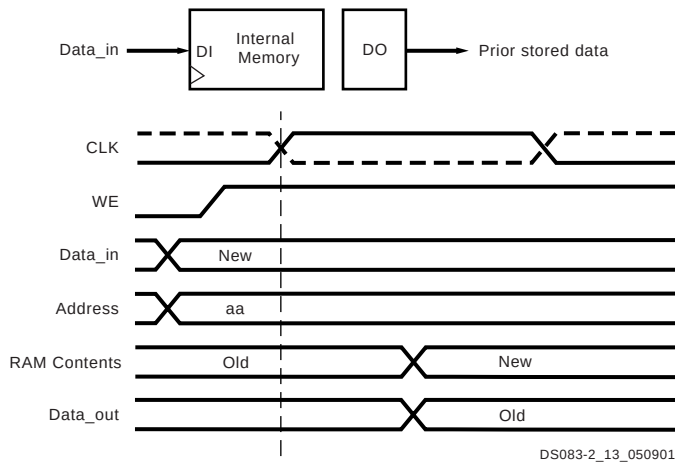


Figure 38: **READ_FIRST** Mode

3. NO_CHANGE

The NO_CHANGE option maintains the content of the output registers, regardless of the write operation. The clock edge during the write mode has no effect on the content of the data output register DO. When the port is configured as

NO_CHANGE, only a read operation loads a new value in the output register DO, as shown in [Figure 39](#).

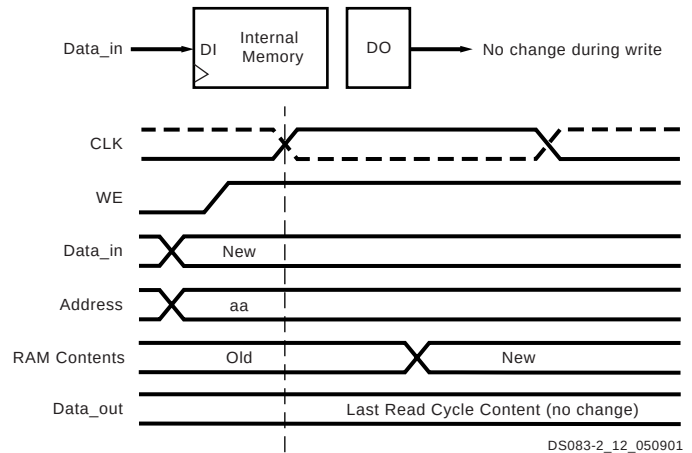


Figure 39: **NO_CHANGE** Mode

Control Pins and Attributes

Virtex-II Pro SelectRAM memory has two independent ports with the control signals described in [Table 18](#). All control inputs including the clock have an optional inversion.

Table 18: Control Functions

Control Signal	Function
CLK	Read and Write Clock
EN	Enable affects Read, Write, Set, Reset
WE	Write Enable
SSR	Set DO register to SRVAL (attribute)

Initial memory content is determined by the INIT_xx attributes. Separate attributes determine the output register value after device configuration (INIT) and SSR is asserted (SRVAL). Both attributes (INIT_B and SRVAL) are available for each port when a block SelectRAM resource is configured as dual-port RAM.

Total Amount of SelectRAM Memory

Virtex-II Pro SelectRAM memory blocks are organized in multiple columns. The number of blocks per column depends on the row size, the number of Processor Blocks, and the number of Rocket I/O transceivers.

[Table 19](#) shows the number of columns as well as the total amount of block SelectRAM memory available for each Virtex-II Pro device. The 18 Kb SelectRAM blocks are cascadable to implement deeper or wider single- or dual-port memory resources.

Table 19: Virtex-II Pro SelectRAM Memory Available

Device	Columns	Total SelectRAM Memory		
		Blocks	in Kb	in Bits
XC2VP2	4	12	216	221,184
XC2VP4	4	28	504	516,096
XC2VP7	6	44	792	811,008
XC2VP20	8	88	1,584	1,622,016
XC2VP50	12	216	3,888	3,981,312

Figure 40 shows the layout of the block RAM columns in the XC2VP4 device.

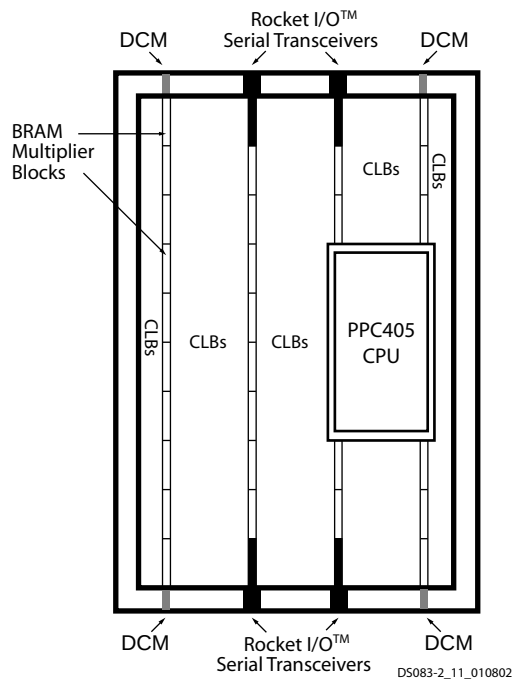


Figure 40: XC2VP4 Block RAM Column Layout

18-Bit x 18-Bit Multipliers

Introduction

A Virtex-II Pro multiplier block is an 18-bit by 18-bit 2's complement signed multiplier. Virtex-II Pro devices incorporate many embedded multiplier blocks. These multipliers can be associated with an 18 Kb block SelectRAM resource or can be used independently. They are optimized for high-speed operations and have a lower power consumption compared to an 18-bit x 18-bit multiplier in slices.

Each SelectRAM memory and multiplier block is tied to four switch matrices, as shown in Figure 41.

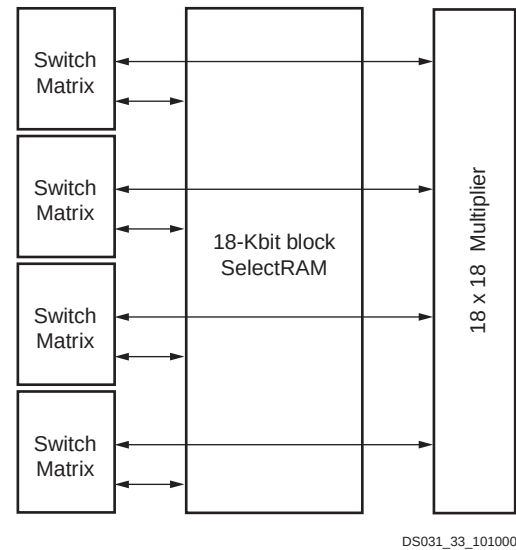


Figure 41: SelectRAM and Multiplier Blocks

Association With Block SelectRAM Memory

The interconnect is designed to allow SelectRAM memory and multiplier blocks to be used at the same time, but some interconnect is shared between the SelectRAM and the multiplier. Thus, SelectRAM memory can be used only up to 18 bits wide when the multiplier is used, because the multiplier shares inputs with the upper data bits of the SelectRAM memory.

This sharing of the interconnect is optimized for an 18-bit-wide block SelectRAM resource feeding the multiplier. The use of SelectRAM memory and the multiplier with an accumulator in LUTs allows for implementation of a digital signal processor (DSP) multiplier-accumulator (MAC) function, which is commonly used in finite and infinite impulse response (FIR and IIR) digital filters.

Configuration

The multiplier block is an 18-bit by 18-bit signed multiplier (2's complement). Both A and B are 18-bit-wide inputs, and the output is 36 bits. Figure 42 shows a multiplier block.

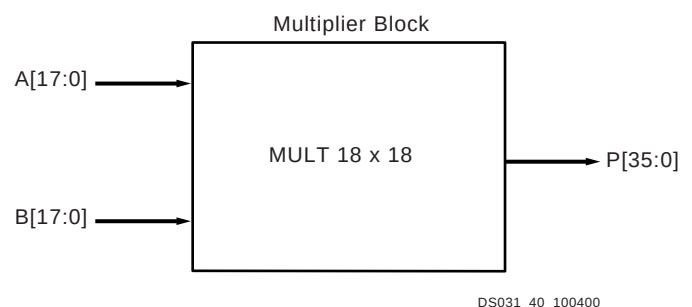


Figure 42: Multiplier Block

Locations / Organization

Multiplier organization is identical to the 18 Kb SelectRAM organization, because each multiplier is associated with an 18 Kb block SelectRAM resource.

Table 20: Multiplier Resources

Device	Columns	Total Multipliers
XC2VP2	4	12
XC2VP4	4	28
XC2VP7	6	44
XC2VP20	8	88
XC2VP50	12	216

In addition to the built-in multiplier blocks, the CLB elements have dedicated logic to implement efficient multipliers in logic. (Refer to **Configurable Logic Blocks (CLBs)**, page 104).

Global Clock Multiplexer Buffers

Virtex-II Pro devices have 16 clock input pins that can also be used as regular user I/Os. Eight clock pads center on both the top edge and the bottom edge of the device, as illustrated in Figure 43.

The global clock multiplexer buffer represents the input to dedicated low-skew clock tree distribution in Virtex-II Pro devices. Like the clock pads, eight global clock multiplexer buffers are on the top edge of the device and eight are on the bottom edge.

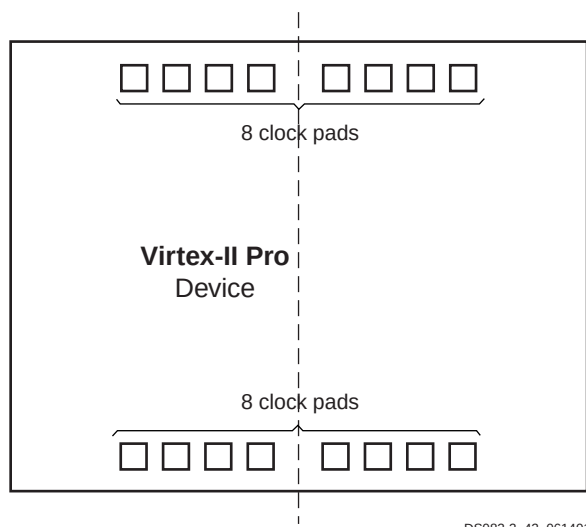
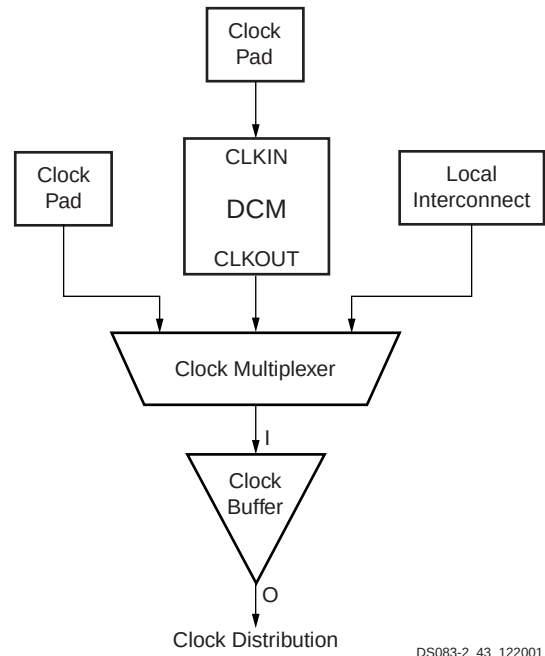


Figure 43: Virtex-II Pro Clock Pads

Each global clock multiplexer buffer can be driven either by the clock pad to distribute a clock directly to the device, or by the Digital Clock Manager (DCM), discussed in **Digital Clock Manager (DCM)**, page 119. Each global clock multiplexer buffer can also be driven by local interconnects. The DCM has clock output(s) that can be connected to global clock multiplexer buffer inputs, as shown in Figure 44.



DS083-2_43_122001

Figure 44: Virtex-II Pro Clock Multiplexer Buffer Configuration

Global clock buffers are used to distribute the clock to some or all synchronous logic elements (such as registers in CLBs and IOBs, and SelectRAM blocks).

Eight global clocks can be used in each quadrant of the Virtex-II Pro device. Designers should consider the clock distribution detail of the device prior to pin-locking and floor-planning. (See the Virtex-II Pro User Guide.)

Figure 45 shows clock distribution in Virtex-II Pro devices.

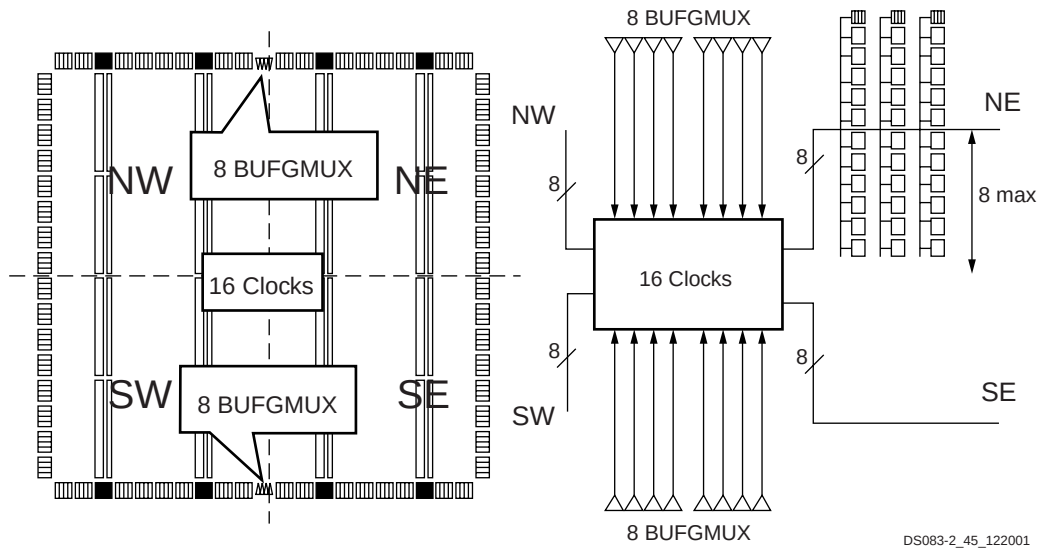


Figure 45: Virtex-II Pro Clock Distribution

In each quadrant, up to eight clocks are organized in clock rows. A clock row supports up to 16 CLB rows (eight up and eight down).

To reduce power consumption, any unused clock branches remain static.

Global clocks are driven by dedicated clock buffers (BUFG), which can also be used to gate the clock (BUFGCE) or to multiplex between two independent clock inputs (BUFGMUX).

The most common configuration option of this element is as a buffer. A BUFG function in this (global buffer) mode, is shown in Figure 46.

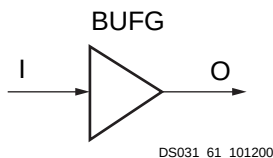


Figure 46: Virtex-II Pro BUFG Function

The Virtex-II Pro global clock buffer BUFG can also be configured as a clock enable/disable circuit (Figure 47), as well as a two-input clock multiplexer (Figure 48). A functional description of these two options is provided below. Each of them can be used in either of two modes, selected by configuration: rising clock edge or falling clock edge.

This section describes the rising clock edge option. For the opposite option, falling clock edge, just change all "rising" references to "falling" and all "High" references to "Low", except for the description of the CE and S levels. The rising clock edge option uses the BUFGCE and BUFGMUX prim-

itives. The falling clock edge option uses the BUFGCE_1 and BUFGMUX_1 primitives.

BUFGCE

If the CE input is active (High) prior to the incoming rising clock edge, this Low-to-High-to-Low clock pulse passes through the clock buffer. Any level change of CE during the incoming clock High time has no effect.

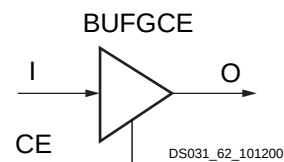


Figure 47: Virtex-II Pro BUFGCE Function

If the CE input is inactive (Low) prior to the incoming rising clock edge, the following clock pulse does not pass through the clock buffer, and the output stays Low. Any level change of CE during the incoming clock High time has no effect. CE must not change during a short setup window just prior to the rising clock edge on the BUFGCE input I. Violating this setup time requirement can result in an undefined runt pulse output.

BUFGMUX

BUFGMUX can switch between two unrelated, even asynchronous clocks. Basically, a Low on S selects the I_0 input, a High on S selects the I_1 input. Switching from one clock to the other is done in such a way that the output High and Low time is never shorter than the shortest High or Low time of

either input clock. As long as the presently selected clock is High, any level change of S has no effect.

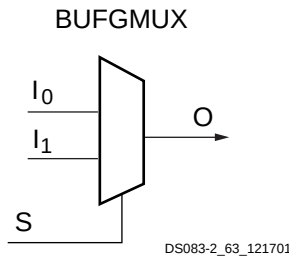


Figure 48: Virtex-II Pro BUFGMUX Function

If the presently selected clock is Low while S changes, or if it goes Low after S has changed, the output is kept Low until the other ("to-be-selected") clock has made a transition from High to Low. At that instant, the new clock starts driving the output.

The two clock inputs can be asynchronous with regard to each other, and the S input can change at any time, except for a short setup time prior to the rising edge of the presently selected clock; that is, prior to the rising edge of the BUFGMUX output O. Violating this setup time requirement can result in an undefined runt pulse output.

All Virtex-II Pro devices have 16 global clock multiplexer buffers.

Figure 49 shows a switchover from CLK0 to CLK1.

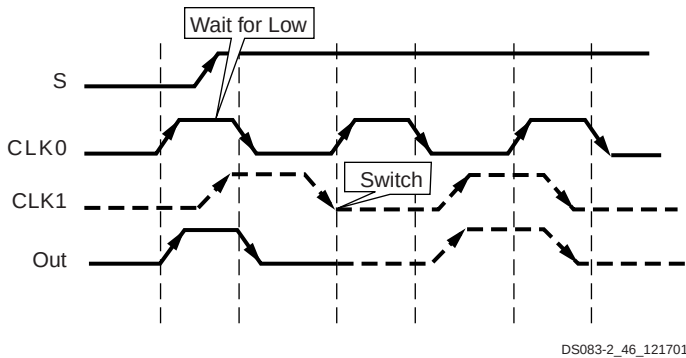


Figure 49: Clock Multiplexer Waveform Diagram

- The current clock is CLK0.
- S is activated High.
- If CLK0 is currently High, the multiplexer waits for CLK0 to go Low.
- Once CLK0 is Low, the multiplexer output stays Low until CLK1 transitions High to Low.
- When CLK1 transitions from High to Low, the output switches to CLK1.
- No glitches or short pulses can appear on the output.

Digital Clock Manager (DCM)

The Virtex-II Pro DCM offers a wide range of powerful clock management features.

- **Clock De-skew:** The DCM generates new system clocks (either internally or externally to the FPGA), which are phase-aligned to the input clock, thus eliminating clock distribution delays.
- **Frequency Synthesis:** The DCM generates a wide range of output clock frequencies, performing very flexible clock multiplication and division.
- **Phase Shifting:** The DCM provides both coarse phase shifting and fine-grained phase shifting with dynamic phase shift control.

The DCM utilizes fully digital delay lines allowing robust high-precision control of clock phase and frequency. It also utilizes fully digital feedback systems, operating dynamically to compensate for temperature and voltage variations during operation.

Up to four of the nine DCM clock outputs can drive inputs to global clock buffers or global clock multiplexer buffers simultaneously (see Figure 50). All DCM clock outputs can simultaneously drive general routing resources, including routes to output buffers.

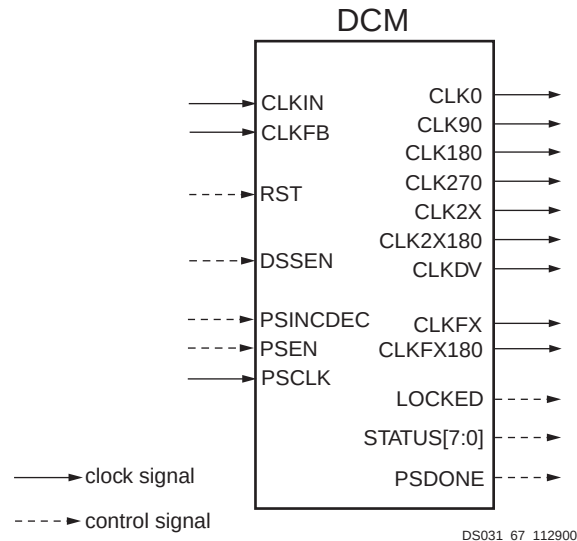


Figure 50: Digital Clock Manager

The DCM can be configured to delay the completion of the Virtex-II Pro configuration process until after the DCM has achieved lock. This guarantees that the chip does not begin operating until after the system clocks generated by the DCM have stabilized.

The DCM has the following general control signals:

- RST input pin: resets the entire DCM
- LOCKED output pin: asserted High when all enabled DCM circuits have locked.
- STATUS output pins (active High): shown in Table 21.

Table 21: DCM Status Pins

Status Pin	Function
0	Phase Shift Overflow
1	CLKIN Stopped
2	CLKFX Stopped
3	N/A
4	N/A
5	N/A
6	N/A
7	N/A

Clock De-skew

The DCM de-skews the output clocks relative to the input clock by automatically adjusting a digital delay line. Additional delay is introduced so that clock edges arrive at internal registers and block RAMs simultaneously with the clock edges arriving at the input clock pad. Alternatively, external clocks, which are also de-skewed relative to the input clock, can be generated for board-level routing. All DCM output clocks are phase-aligned to CLK0 and, therefore, are also phase-aligned to the input clock.

To achieve clock de-skew, the CLKFB input must be connected, and its source must be either CLK0 or CLK2X. Note that CLKFB must always be connected, unless only the CLKFX or CLKFX180 outputs are used and de-skew is not required.

Frequency Synthesis

The DCM provides flexible methods for generating new clock frequencies. Each method has a different operating frequency range and different AC characteristics. The CLK2X and CLK2X180 outputs double the clock frequency. The CLKDV output creates divided output clocks with division options of 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 7.5, 8, 9, 10, 11, 12, 13, 14, 15, and 16.

The CLKFX and CLKFX180 outputs can be used to produce clocks at the following frequency:

$$\text{FREQ}_{\text{CLKFX}} = (M/D) \bullet \text{FREQ}_{\text{CLKIN}}$$

where M and D are two integers. Specifications for M and D are provided under **DCM Timing Parameters**. By default, $M = 4$ and $D = 1$, which results in a clock output frequency four times faster than the clock input frequency (CLKIN).

CLK2X180 is phase shifted 180 degrees relative to CLK2X. CLKFX180 is phase shifted 180 degrees relative to CLKFX.

All frequency synthesis outputs automatically have 50/50 duty cycles, with the exception of the CLKDV output when performing a non-integer divide in high-frequency mode. See [Table 22](#) for more details.

Note that CLK2X and CLK2X180 are not available in high-frequency mode.

Table 22: CLKDV Duty Cycle for Non-integer Divides

CLKDV_DIVIDE	Duty Cycle
1.5	1/3
2.5	2/5
3.5	3/7
4.5	4/9
5.5	5/11
6.5	6/13
7.5	7/15

Phase Shifting

The DCM provides additional control over clock skew through either coarse or fine-grained phase shifting. The CLK0, CLK90, CLK180, and CLK270 outputs are each phase shifted by $\frac{1}{4}$ of the input clock period relative to each other, providing coarse phase control. Note that CLK90 and CLK270 are not available in high-frequency mode.

Fine-phase adjustment affects all nine DCM output clocks. When activated, the phase shift between the rising edges of CLKIN and CLKFB is a specified fraction of the input clock period.

In variable mode, the PHASE_SHIFT value can also be dynamically incremented or decremented as determined by PSINCDEC synchronously to PSCLK, when the PSEN input is active. [Figure 51](#) illustrates the effects of fine-phase shifting. For more information on DCM features, see the *Virtex-II Pro User Guide*.

[Table 23](#) lists fine-phase shifting control pins, when used in variable mode.

Table 23: Fine Phase Shifting Control Pins

Control Pin	Direction	Function
PSINCDEC	In	Increment or decrement
PSEN	In	Enable $\pm$ phase shift
PSCLK	In	Clock for phase shift
PSDONE	Out	Active when completed

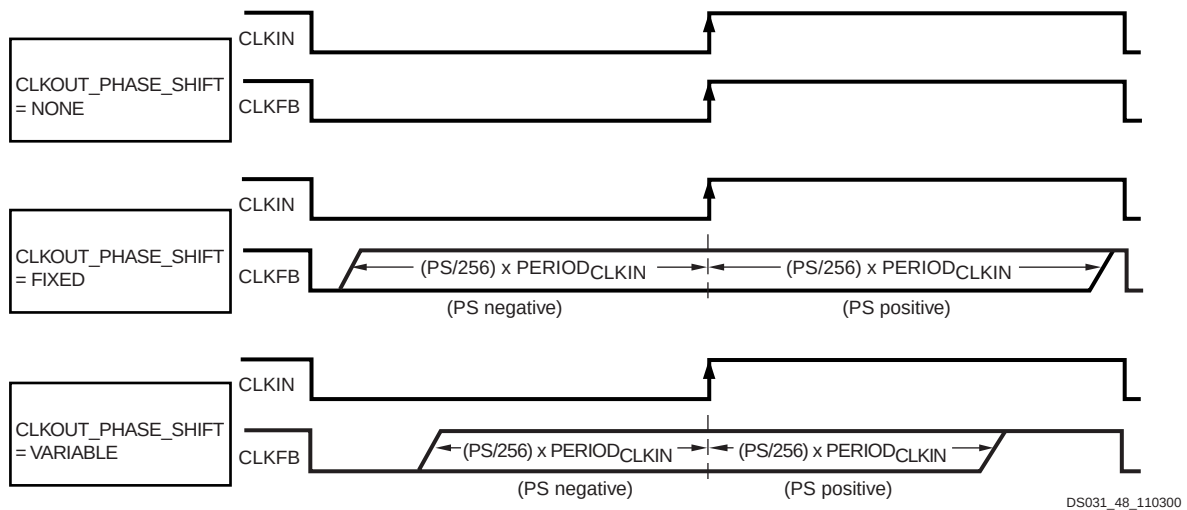


Figure 51: Fine-Phase Shifting Effects

Two separate components of the phase shift range must be understood:

- PHASE_SHIFT attribute range
- FINE_SHIFT_RANGE DCM timing parameter range

The PHASE_SHIFT attribute is the numerator in the following equation:

$$\text{Phase Shift (ns)} = (\text{PHASE\_SHIFT}/256) \times \text{PERIOD}_{\text{CLKIN}}$$

The full range of this attribute is always -255 to +255, but its practical range varies with CLKIN frequency, as constrained by the FINE_SHIFT_RANGE component, which represents the total delay achievable by the phase shift delay line. Total delay is a function of the number of delay taps used in the circuit. Across process, voltage, and temperature, this absolute range is guaranteed to be as specified under **DCM Timing Parameters**.

Absolute range (fixed mode) = $\pm \text{FINE\_SHIFT\_RANGE}$

Absolute range (variable mode) = $\pm \text{FINE\_SHIFT\_RANGE}/2$

The reason for the difference between fixed and variable modes is as follows. For variable mode to allow symmetric, dynamic sweeps from -255/256 to +255/256, the DCM sets the "zero phase skew" point as the middle of the delay line,

thus dividing the total delay line range in half. In fixed mode, since the PHASE_SHIFT value never changes after configuration, the entire delay line is available for insertion into either the CLKIN or CLKFB path (to create either positive or negative skew).

Taking both of these components into consideration, the following are some usage examples:

- If $\text{PERIOD}_{\text{CLKIN}} = 2 \times \text{FINE\_SHIFT\_RANGE}$, then PHASE_SHIFT in fixed mode is limited to ± 128 , and in variable mode it is limited to ± 64 .
- If $\text{PERIOD}_{\text{CLKIN}} = \text{FINE\_SHIFT\_RANGE}$, then PHASE_SHIFT in fixed mode is limited to ± 255 , and in variable mode it is limited to ± 128 .
- If $\text{PERIOD}_{\text{CLKIN}} \leq 0.5 \times \text{FINE\_SHIFT\_RANGE}$, then PHASE_SHIFT is limited to ± 255 in either mode.

Operating Modes

The frequency ranges of DCM input and output clocks depend on the operating mode specified, either low-frequency mode or high-frequency mode, according to Table 24. For actual values, see **Virtex-II Pro Switching Characteristics (Module 3)**. The CLK2X, CLK2X180,

Table 24: DCM Frequency Ranges

Output Clock	Low-Frequency Mode		High-Frequency Mode	
	CLKIN Input	CLK Output	CLKIN Input	CLK Output
CLK0, CLK180	CLKIN_FREQ_DLL_LF	CLKOUT_FREQ_1X_LF	CLKIN_FREQ_DLL_HF	CLKOUT_FREQ_1X_HF
CLK90, CLK270	CLKIN_FREQ_DLL_LF	CLKOUT_FREQ_1X_LF	NA	NA
CLK2X, CLK2X180	CLKIN_FREQ_DLL_LF	CLKOUT_FREQ_2X_LF	NA	NA
CLKDV	CLKIN_FREQ_DLL_LF	CLKOUT_FREQ_DV_LF	CLKIN_FREQ_DLL_HF	CLKOUT_FREQ_DV_HF
CLKFX, CLKFX180	CLKIN_FREQ_FX_LF	CLKOUT_FREQ_FX_LF	CLKIN_FREQ_FX_HF	CLKOUT_FREQ_FX_HF

CLK90, and CLK270 outputs are not available in high-frequency mode.

High or low-frequency mode is selected by an attribute.

Routing

DCM and MGT Locations/Organization

Virtex-II Pro DCMs and serial transceivers (MGTs) are placed on the top and bottom of each block RAM and multiplier column in some combination, as shown in [Table 25](#). The number of DCMs and Rocket I/O transceiver cores total to twice the number of columns in the device. Refer to [Figure 40, page 116](#) for an illustration of this in the XC2VP4 device.

Table 25: DCM Organization

Device	Columns	DCMs	MGTs
XC2VP2	4	4	4
XC2VP4	4	4	4
XC2VP7	6	4	8
XC2VP20	8	8	8
XC2VP50	12	8	16

Place-and-route software takes advantage of this regular array to deliver optimum system performance and fast compile times. The segmented routing resources are essential to guarantee IP cores portability and to efficiently handle an incremental design flow that is based on modular implementations. Total design time is reduced due to fewer and shorter design iterations.

Hierarchical Routing Resources

Most Virtex-II Pro signals are routed using the global routing resources, which are located in horizontal and vertical routing channels between each switch matrix.

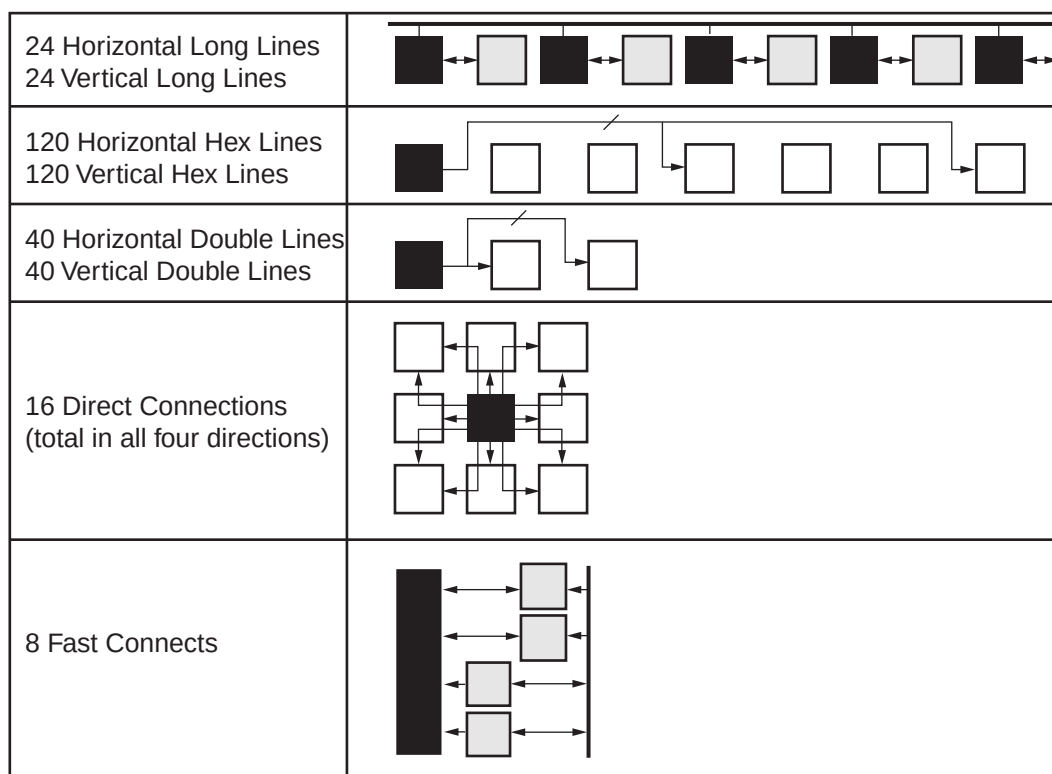
As shown in [Figure 52, page 123](#), Virtex-II Pro has fully buffered programmable interconnections, with a number of resources counted between any two adjacent switch matrix rows or columns. Fanout has minimal impact on the performance of each net.

- The long lines are bidirectional wires that distribute signals across the device. Vertical and horizontal long lines span the full height and width of the device.
- The hex lines route signals to every third or sixth block away in all four directions. Organized in a staggered pattern, hex lines can only be driven from one end. Hex-line signals can be accessed either at the endpoints or at the midpoint (three blocks from the source).
- The double lines route signals to every first or second block away in all four directions. Organized in a staggered pattern, double lines can be driven only at their endpoints. Double-line signals can be accessed either at the endpoints or at the midpoint (one block from the source).
- The direct connect lines route signals to neighboring blocks: vertically, horizontally, and diagonally.
- The fast connect lines are the internal CLB local interconnections from LUT outputs to LUT inputs.

Dedicated Routing

In addition to the global and local routing resources, dedicated signals are available.

- There are eight global clock nets per quadrant. (See [Global Clock Multiplexer Buffers, page 117](#).)
- Horizontal routing resources are provided for on-chip 3-state buses. Four partitionable bus lines are provided per CLB row, permitting multiple buses within a row. (See [3-State Buffers, page 112](#).)
- Two dedicated carry-chain resources per slice column (two per CLB column) propagate carry-chain MUXCY output signals vertically to the adjacent slice. (See [CLB/Slice Configurations, page 112](#).)
- One dedicated SOP chain per slice row (two per CLB row) propagate ORCY output logic signals horizontally to the adjacent slice. (See [Sum of Products, page 111](#).)
- One dedicated shift-chain per CLB connects the output of LUTs in shift-register mode to the input of the next LUT in shift-register mode (vertically) inside the CLB. (See [Shift Registers, page 107](#).)



DS031_60_110200

Figure 52: Hierarchical Routing Resources

Configuration

Virtex-II Pro devices are configured by loading application specific configuration data into the internal configuration memory. Configuration is carried out using a subset of the device pins, some of which are dedicated, while others can be re-used as general purpose inputs and outputs once configuration is complete.

Depending on the system design, several configuration modes are supported, selectable via mode pins. The mode pins M2, M1 and M0 are dedicated pins. An additional pin, HSWAP_EN is used in conjunction with the mode pins to select whether user I/O pins have pull-ups during configuration. By default, HSWAP_EN is tied High (internal pull-up) which shuts off the pull-ups on the user I/O pins during configuration. When HSWAP_EN is tied Low, user I/Os have pull-ups during configuration. Other dedicated pins are CCLK (the configuration clock pin), DONE, PROG_B, and the boundary-scan pins: TDI, TDO, TMS, and TCK. Depending on the configuration mode chosen, CCLK can be an output generated by the FPGA, or an input accepting an externally generated clock. The configuration pins and boundary scan pins are independent of the V_{CCO} . The auxiliary power supply (V_{CCAUX}) of 2.5V is used for these pins. See **Virtex-II Pro Switching Characteristics (Module 3)**.

A persist option is available which can be used to force the configuration pins to retain their configuration function even after device configuration is complete. If the persist option is

not selected then the configuration pins with the exception of CCLK, PROG_B, and DONE can be used as user I/O in normal operation. The persist option does not apply to the boundary-scan related pins. The persist feature is valuable in applications which employ partial reconfiguration or reconfiguration on the fly.

Virtex-II Pro supports the following five configuration modes:

- **Slave-Serial Mode**
- **Master-Serial Mode**
- **Slave SelectMAP Mode**
- **Master SelectMAP Mode**
- **Boundary-Scan (JTAG, IEEE 1532) Mode**

Refer to **Table 26, page 124**.

A detailed description of configuration modes is provided in the *Virtex-II Pro User Guide*.

Slave-Serial Mode

In slave-serial mode, the FPGA receives configuration data in bit-serial form from a serial PROM or other serial source of configuration data. The CCLK pin on the FPGA is an input in this mode. The serial bitstream must be setup at the DIN input pin a short time before each rising edge of the externally generated CCLK.

Multiple FPGAs can be daisy-chained for configuration from a single source. After a particular FPGA has been config-

ured, the data for the next device is routed internally to the DOUT pin. The data on the DOUT pin changes on the rising edge of CCLK.

Slave-serial mode is selected by applying [111] to the mode pins (M2, M1, M0). A weak pull-up on the mode pins makes slave serial the default mode if the pins are left unconnected.

Master-Serial Mode

In master-serial mode, the CCLK pin is an output pin. It is the Virtex-II Pro FPGA device that drives the configuration clock on the CCLK pin to a Xilinx Serial PROM which in turn feeds bit-serial data to the DIN input. The FPGA accepts this data on each rising CCLK edge. After the FPGA has been loaded, the data for the next device in a daisy-chain is presented on the DOUT pin after the rising CCLK edge.

The interface is identical to slave serial except that an internal oscillator is used to generate the configuration clock (CCLK). A wide range of frequencies can be selected for CCLK which always starts at a slow default frequency. Configuration bits then switch CCLK to a higher frequency for the remainder of the configuration.

Slave SelectMAP Mode

The SelectMAP mode is the fastest configuration option. Byte-wide data is written into the Virtex-II Pro FPGA device with a BUSY flag controlling the flow of data. An external data source provides a byte stream, CCLK, an active Low Chip Select (CS_B) signal and a Write signal (RDWR_B). If BUSY is asserted (High) by the FPGA, the data must be held until BUSY goes Low. Data can also be read using the SelectMAP mode. If RDWR_B is asserted, configuration data is read out of the FPGA as part of a readback operation.

After configuration, the pins of the SelectMAP port can be used as additional user I/O. Alternatively, the port can be retained to permit high-speed 8-bit readback using the persist option.

Multiple Virtex-II Pro FPGAs can be configured using the SelectMAP mode, and be made to start-up simultaneously. To configure multiple devices in this way, wire the individual CCLK, Data, RDWR_B, and BUSY pins of all the devices in parallel. The individual devices are loaded separately by deasserting the CS_B pin of each device in turn and writing the appropriate data.

Master SelectMAP Mode

This mode is a master version of the SelectMAP mode. The device is configured byte-wide on a CCLK supplied by the Virtex-II Pro FPGA device. Timing is similar to the Slave SerialMAP mode except that CCLK is supplied by the Virtex-II Pro FPGA.

Boundary-Scan (JTAG, IEEE 1532) Mode

In boundary-scan mode, dedicated pins are used for configuring the Virtex-II Pro device. The configuration is done entirely through the IEEE 1149.1 Test Access Port (TAP). Virtex-II Pro device configuration using Boundary scan is compliant with IEEE 1149.1-1993 standard and the new IEEE 1532 standard for In-System Configurable (ISC) devices. The IEEE 1532 standard is backward compliant with the IEEE 1149.1-1993 TAP and state machine. The IEEE Standard 1532 for In-System Configurable (ISC) devices is intended to be programmed, reprogrammed, or tested on the board via a physical and logical protocol. Configuration through the boundary-scan port is always available, independent of the mode selection. Selecting the boundary-scan mode simply turns off the other modes.

Table 26: Virtex-II Pro Configuration Mode Pin Settings

Configuration Mode ⁽¹⁾	M2	M1	M0	CCLK Direction	Data Width	Serial D _{OUT} ⁽²⁾
Master Serial	0	0	0	Out	1	Yes
Slave Serial	1	1	1	In	1	Yes
Master SelectMAP	0	1	1	Out	8	No
Slave SelectMAP	1	1	0	In	8	No
Boundary Scan	1	0	1	N/A	1	No

Notes:

1. The HSWAP_EN pin controls the pullups. Setting M2, M1, and M0 selects the configuration mode, while the HSWAP_EN pin controls whether or not the pullups are used.
2. Daisy chaining is possible only in modes where Serial D_{OUT} is used. For example, in SelectMAP modes, the first device does NOT support daisy chaining of downstream devices.

Table 27 lists the total number of bits required to configure each device.

Table 27: Virtex-II Pro Bitstream Lengths

Device	Number of Configuration Bits
XC2VP2	1,305,440
XC2VP4	3,006,560
XC2VP7	4,485,472
XC2VP20	8,214,624
XC2VP50	19,021,408

Configuration Sequence

The configuration of Virtex-II Pro devices is a three-phase process. First, the configuration memory is cleared. Next, configuration data is loaded into the memory, and finally, the logic is activated by a start-up process.

Configuration is automatically initiated on power-up unless it is delayed by the user. The INIT_B pin can be held Low using an open-drain driver. An open-drain is required since INIT_B is a bidirectional open-drain pin that is held Low by a Virtex-II Pro FPGA device while the configuration memory is being cleared. Extending the time that the pin is Low causes the configuration sequencer to wait. Thus, configuration is delayed by preventing entry into the phase where data is loaded.

The configuration process can also be initiated by asserting the PROG_B pin. The end of the memory-clearing phase is signaled by the INIT_B pin going High, and the completion of the entire process is signaled by the DONE pin going High. The Global Set/Reset (GSR) signal is pulsed after the last frame of configuration data is written but before the start-up sequence. The GSR signal resets all flip-flops on the device.

The default start-up sequence is that one CCLK cycle after DONE goes High, the global 3-state signal (GTS) is released. This permits device outputs to turn on as necessary. One CCLK cycle later, the Global Write Enable (GWE) signal is released. This permits the internal storage elements to begin changing state in response to the logic and the user clock.

The relative timing of these events can be changed via configuration options in software. In addition, the GTS and GWE events can be made dependent on the DONE pins of multiple devices all going High, forcing the devices to start synchronously. The sequence can also be paused at any stage, until lock has been achieved on any or all DCMs, as well as DCI.

Readback

In this mode, configuration data from the Virtex-II Pro FPGA device can be read back. Readback is supported only in the SelectMAP (master and slave) and Boundary Scan mode.

Along with the configuration data, it is possible to read back the contents of all registers, distributed SelectRAM, and block RAM resources. This capability is used for real-time debugging. For more detailed configuration information, see the *Virtex-II Pro User Guide*.

Bitstream Encryption

Virtex-II Pro devices have an on-chip decryptor using one or two sets of three keys for triple-key Data Encryption Standard (DES) operation. Xilinx software tools offer an optional encryption of the configuration data (bitstream) with a triple-key DES determined by the designer.

The keys are stored in the FPGA by JTAG instruction and retained by a battery connected to the V_{BATT} pin, when the device is not powered. Virtex-II Pro devices can be configured with the corresponding encrypted bitstream, using any of the configuration modes described previously.

A detailed description of how to use bitstream encryption is provided in the *Virtex-II Pro User Guide*. Your local FAE can also provide specific information on this feature.

Partial Reconfiguration

Partial reconfiguration of Virtex-II Pro devices can be accomplished in either Slave SelectMAP mode or Boundary-Scan mode. Instead of resetting the chip and doing a full configuration, new data is loaded into a specified area of the chip, while the rest of the chip remains in operation. Data is loaded on a column basis, with the smallest load unit being a configuration “frame” of the bitstream (device size dependent).

Partial reconfiguration is useful for applications that require different designs to be loaded into the same area of a chip, or that require the ability to change portions of a design without having to reset or reconfigure the entire chip.

Revision History

This section records the change history for this module of the data sheet.

Date	Version	Revision
01/31/02	1.0	Initial Xilinx release.

Virtex-II Pro Data Sheet Modules

The Virtex-II Pro Data Sheet contains the following modules:

- **Virtex-II Pro™ Platform FPGAs: Introduction and Overview (Module 1)**
- **Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics (Module 3)**
- **Virtex-II Pro Platform FPGAs: Functional Description (Module 2)**
- **Virtex-II Pro Platform FPGAs: Pinout Information (Module 4)**

Virtex-II Pro Electrical Characteristics

Virtex-II Pro devices are provided in -8, -7, and -6 speed grades, with -8 having the highest performance.

Virtex-II Pro DC and AC characteristics are specified for both commercial and industrial grades. Except the operating temperature range or unless otherwise noted, all the DC and AC electrical parameters are the same for a particular speed grade (that is, the timing characteristics of a -6 speed grade industrial device are the same as for a -6 speed grade

commercial device). However, only selected speed grades and/or devices might be available in the industrial range.

All supply voltage and junction temperature specifications are representative of worst-case conditions. The parameters included are common to popular designs and typical applications. Contact Xilinx for design considerations requiring more detailed information.

All specifications are subject to change without notice.

Virtex-II Pro DC Characteristics

Table 1: Absolute Maximum Ratings

Symbol	Description		Units
V_{CCINT}	Internal supply voltage relative to GND	-0.5 to 1.65	V
V_{CCAUX}	Auxiliary supply voltage relative to GND	-0.5 to 3.45	V
V_{CCO}	Output drivers supply voltage relative to GND	-0.5 to 3.45	V
V_{BATT}	Key memory battery backup supply	-0.5 to 3.45	V
V_{REF}	Input reference voltage	-0.5 to 3.45	V
V_{IN}	Input voltage relative to GND (user and dedicated I/Os)	-0.5 ⁽²⁾ to 3.45 ⁽⁴⁾	V
V_{TS}	Voltage applied to 3-state output (user and dedicated I/Os)	-0.5 ⁽³⁾ to 3.45 ⁽⁵⁾	V
$V_{CCAUXRX}$	Auxilliary supply voltage relative to analog ground, GNDA (Rocket I/O pins)	-0.5 to 3.45	V
$V_{CCAUXTX}$	Auxilliary supply voltage relative to analog ground, GNDA (Rocket I/O pins)	-0.5 to 3.45	V
V_{TTX}	Terminal transmit supply voltage relative to GND (Rocket I/O pins)	-0.5 to 3.45	V
V_{TRX}	Terminal receive supply voltage relative to GND (Rocket I/O pins)	-0.5 to 3.45	V
T_{STG}	Storage temperature (ambient)	-65 to +150	°C
T_{SOL}	Maximum soldering temperature	+220	°C
T_J	Operating junction temperature	+125	°C

Notes:

- Stresses beyond those listed under Absolute Maximum Ratings might cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those listed under Operating Conditions is not implied. Exposure to Absolute Maximum Ratings conditions for extended periods of time might affect device reliability.
- For 3.3V I/O standards only, I/O input pin voltage, including negative undershoot, must not fall below 0.0V, either on a continuous or transient basis (i.e., no negative undershoot is allowed). See [Table 6, page 130](#).
- For 3.3V I/O standards only, I/O output pin voltage while in 3-state mode must not fall below 0.0V, either on a continuous or transient basis. See [Table 6, page 130](#).
- I/O input pin voltage, including overshoot, must not exceed 3.45V, either on a continuous or transient basis.
- I/O output pin voltage while in 3-state mode must not exceed 3.45V, either on a continuous or transient basis.

Table 2: Recommended Operating Conditions

Symbol	Description		Min	Max	Units
V_{CCINT}	Internal supply voltage relative to GND, $T_J = 0^\circ\text{C}$ to $+85^\circ\text{C}$	Commercial	1.425	1.575	V
	Internal supply voltage relative to GND, $T_J = -40^\circ\text{C}$ to $+100^\circ\text{C}$	Industrial	1.425	1.575	V
$V_{CCAUX}^{(1)}$	Auxiliary supply voltage relative to GND, $T_J = 0^\circ\text{C}$ to $+85^\circ\text{C}$	Commercial	2.375	2.625	V
	Auxiliary supply voltage relative to GND, $T_J = -40^\circ\text{C}$ to $+100^\circ\text{C}$	Industrial	2.375	2.625	V
$V_{CCO}^{(2)}$	Supply voltage relative to GND, $T_J = 0^\circ\text{C}$ to $+85^\circ\text{C}$	Commercial	1.2	3.45 ⁽⁴⁾	V
	Supply voltage relative to GND, $T_J = -40^\circ\text{C}$ to $+100^\circ\text{C}$	Industrial	1.2	3.45 ⁽⁴⁾	V
$V_{BATT}^{(3)}$	Battery voltage relative to GND, $T_J = 0^\circ\text{C}$ to $+85^\circ\text{C}$	Commercial	1.0	2.63	V
	Battery voltage relative to GND, $T_J = -40^\circ\text{C}$ to $+100^\circ\text{C}$	Industrial	1.0	2.63	V
$V_{CCAUXRX}$, V_{CCAUTX}	Auxiliary supply voltage relative to GNDA	Commercial	2.375	2.625	V
	Auxiliary supply voltage relative to GNDA	Industrial	2.375	2.625	V
V_{TTX} , V_{TRX}	Terminal supply voltage relative to GND	Commercial	1.8	2.625	V
	Terminal supply voltage relative to GND	Industrial	1.8	2.625	V

Notes:

1. For LVDS operation, V_{CCAUX} min is 2.37V and max is 2.63V.
2. Configuration data is retained even if V_{CCO} drops to 0V.
3. If battery is not used, do not connect V_{BATT} .
4. For 3.3V operation, see [Table 4-1, page 448](#), for banking information.

Table 3: DC Characteristics Over Recommended Operating Conditions

Symbol	Description	Device	Min	Typ	Max	Units
V_{DRINT}	Data retention V_{CCINT} voltage (below which configuration data might be lost)	All	1.2			V
V_{DRI}	Data retention V_{CCAUX} voltage (below which configuration data might be lost)	All				V
I_{REF}	V_{REF} current per bank	All				μA
I_L	Input or output leakage current per pin	All				μA
C_{IN}	Input capacitance (sample tested)	All				pF
I_{RPU}	Pad pull-up (when selected) @ $V_{in} = 0\text{V}$, $V_{CCO} = 3.3\text{V}$ (sample tested)	All	Note (1)			mA
I_{RPD}	Pad pull-down (when selected) @ $V_{in} = 3.6\text{V}$ (sample tested)	All	Note (1)			mA
I_{CCAUTX}	Operating V_{CCAUTX} supply current			60		mA
$I_{CCAUXRX}$	Operating $V_{CCAUXRX}$ supply current			35		mA
I_{TTX}	Operating I_{TTX} supply current when transmitter is AC coupled			30		mA
	Operating I_{TTX} supply current when transmitter is DC coupled			15		mA
I_{TRX}	Operating I_{TRX} supply current when receiver is AC coupled			TBD		mA
	Operating I_{TRX} supply current when receiver is DC coupled			15		mA

Table 3: DC Characteristics Over Recommended Operating Conditions (Continued)

Symbol	Description	Device	Min	Typ	Max	Units
P_{CPU}	Power dissipation of PowerPC® 405 processor block					mW / MHz
P_{RXTX}	Power dissipation of Rocket I/O @ 3.125 Gb/s per channel			350		mW
	Power dissipation of Rocket I/O @ 2.5 Gb/s per channel			310		mW
	Power dissipation of Rocket I/O @ 1.25 Gb/s per channel			230		mW

Notes:

1. Internal pull-up and pull-down resistors guarantee valid logic levels at unconnected input pins. These pull-up and pull-down resistors do not guarantee valid logic levels when input pins are connected to other circuits.

Table 4: Quiescent Supply Current

Symbol	Description	Device	Min	Typ	Max	Units
I_{CCINTQ}	Quiescent V_{CCINT} supply current	XC2VP2				mA
		XC2VP4				mA
		XC2VP7				mA
		XC2VP20				mA
		XC2VP50				mA
I_{CCOQ}	Quiescent V_{CCO} supply current	XC2VP2				mA
		XC2VP4				mA
		XC2VP7				mA
		XC2VP20				mA
		XC2VP50				mA
I_{CCAUXQ}	Quiescent V_{CCAUX} supply current	XC2VP2				mA
		XC2VP4				mA
		XC2VP7				mA
		XC2VP20				mA
		XC2VP50				mA

Notes:

1. With no output current loads, no active input pull-up resistors, all I/O pins are 3-state and floating.
2. If DCI or differential signaling is used, more accurate quiescent current estimates can be obtained by using the Power Estimator or XPOWER™.

Power-On Power Supply Requirements

Xilinx FPGAs require a certain amount of supply current during power-on to insure proper device operation. The actual current consumed depends on the power-on ramp rate of the power supply.

The V_{CCINT} , V_{CCAUX} , and V_{CCO} power supplies must ramp on no faster than 100 μ s and no slower than 50 ms. Ramp on is defined as: 0 V_{DC} to minimum supply voltages (see Table 2, page 128).

V_{CCAUX} and V_{CCO} for bank 4 must be connected together (2.5 V_{DC}) to meet the following specification.

Table 5, page 130, shows the minimum current required by Virtex-II Pro devices for proper power on and configuration.

Power supplies can be turned on in any sequence, as long as V_{CCAUX} and V_{CCO} are connected together for bank 4.

If any V_{CCO} bank powers up before V_{CCAUX} , then each bank draws up to 600 mA, worst case, until the V_{CCAUX} powers on. This does not harm the device. (Note that the 600 mA is *peak transient current*, which eventually dissipates even if V_{CCAUX} does not power on.)

If the currents minimums shown in Table 5 are met, the device powers on properly after all three supplies have passed through their power-on reset threshold voltages.

Once initialized and configured, use the power calculator to estimate current drain on these supplies.

Table 5: Power-On Current for Virtex-II Pro Devices

Symbol	Device					Units
	XC2VP2	XC2VP4	XC2VP7	XC2VP20	XC2VP50	
$I_{CCINTMIN}$	250	250	250	250	500	mA
$I_{CCAUXMIN}$	250	250	250	250	250	mA
I_{CCOMIN}	10	10	10	10	10	mA

SelectI/O DC Input and Output Levels

Values for V_{IL} and V_{IH} are recommended input voltages. Values for I_{OL} and I_{OH} are guaranteed over the recommended operating conditions at the V_{OL} and V_{OH} test points. Only selected standards are tested. These are cho-

sen to ensure that all standards meet their specifications. The selected standards are tested at minimum V_{CCO} with the respective V_{OL} and V_{OH} voltage levels shown. Other standards are sample tested.

Table 6: DC Input and Output Levels

Input/Output Standard	V_{IL}		V_{IH}		V_{OL}	V_{OH}	I_{OL}	I_{OH}
	V, min	V, max	V, min	V, max	V, Max	V, Min	mA	mA
LVTTL ⁽¹⁾	0.0	0.8	2.0	V_{CCO}	0.4	2.4	24	-24
LVC MOS33	0.0	0.8	2.0	V_{CCO}	0.4	$V_{CCO} - 0.4$	24	-24
LVC MOS25	-0.5	0.7	1.7	$V_{CCO} + 0.4$	0.4	$V_{CCO} - 0.4$	24	-24
LVC MOS18	-0.5	20% V_{CCO}	70% V_{CCO}	$V_{CCO} + 0.4$	0.4	$V_{CCO} - 0.45$	16	-16
LVC MOS15	-0.5	20% V_{CCO}	70% V_{CCO}	$V_{CCO} + 0.4$	0.4	$V_{CCO} - 0.45$	16	-16
PCI33_3 ⁽²⁾	0.0	30% V_{CCO}	50% V_{CCO}	V_{CCO}	10% V_{CCO}	90% V_{CCO}		
PCI66_3 ⁽²⁾	0.0	30% V_{CCO}	50% V_{CCO}	V_{CCO}	10% V_{CCO}	90% V_{CCO}		
GTLP	-0.5	$V_{REF} - 0.1$	$V_{REF} + 0.1$	$V_{CCO} + 0.4$	0.6	n/a	36	n/a
GTL	-0.5	$V_{REF} - 0.05$	$V_{REF} + 0.05$	$V_{CCO} + 0.4$	0.4	n/a	40	n/a
HSTL I	-0.5	$V_{REF} - 0.1$	$V_{REF} + 0.1$	$V_{CCO} + 0.4$	0.4 ⁽³⁾	$V_{CCO} - 0.4$	8 ⁽³⁾	-8 ⁽³⁾
HSTL II	-0.5	$V_{REF} - 0.1$	$V_{REF} + 0.1$	$V_{CCO} + 0.4$	0.4 ⁽³⁾	$V_{CCO} - 0.4$	16 ⁽³⁾	-16 ⁽³⁾
HSTL III	-0.5	$V_{REF} - 0.1$	$V_{REF} + 0.1$	$V_{CCO} + 0.4$	0.4 ⁽³⁾	$V_{CCO} - 0.4$	24 ⁽³⁾	-8 ⁽³⁾
HSTL IV	-0.5	$V_{REF} - 0.1$	$V_{REF} + 0.1$	$V_{CCO} + 0.4$	0.4 ⁽³⁾	$V_{CCO} - 0.4$	48 ⁽³⁾	-8 ⁽³⁾
SSTL3 I	0.0	$V_{REF} - 0.2$	$V_{REF} + 0.2$	V_{CCO}	$V_{REF} - 0.6$	$V_{REF} + 0.6$	8	-8
SSTL3 II	0.0	$V_{REF} - 0.2$	$V_{REF} + 0.2$	V_{CCO}	$V_{REF} - 0.8$	$V_{REF} + 0.8$	16	-16
SSTL2 I	-0.5	$V_{REF} - 0.2$	$V_{REF} + 0.2$	$V_{CCO} + 0.4$	$V_{REF} - 0.61$	$V_{REF} + 0.65$	7.6	-7.6
SSTL2 II	-0.5	$V_{REF} - 0.2$	$V_{REF} + 0.2$	$V_{CCO} + 0.4$	$V_{REF} - 0.80$	$V_{REF} + 0.80$	15.2	-15.2

Notes:

1. V_{OL} and V_{OH} for lower drive currents are sample tested. The DONE pin is always CMOS 2.5 12 mA.
2. For optimum performance, it is recommended that PCI be used in conjunction with LVDCI_33. Contact Xilinx for more details.
3. This applies to 1.5V and 1.8V HSTL.

LDT DC Specifications (LDT_25)

Table 7: LDT DC Specifications

DC Parameter	Symbol	Conditions	Min	Typ	Max	Units
Supply Voltage	V_{CCO}		2.38	2.5	2.63	V
Differential Output Voltage	V_{OD}	$R_T = 100$ ohm across Q and $\overline{Q}$ signals	500	600	700	mV
Change in V_{OD} Magnitude	ΔV_{OD}		-15		15	mV
Output Common Mode Voltage	V_{OCM}	$R_T = 100$ ohm across Q and $\overline{Q}$ signals	560	600	640	mV
Change in V_{OS} Magnitude	ΔV_{OCM}		-15		15	mV
Input Differential Voltage	V_{ID}		200	600	1000	mV
Change in V_{ID} Magnitude	ΔV_{ID}		-15		15	mV
Input Common Mode Voltage	V_{ICM}		500	600	700	mV
Change in V_{ICM} Magnitude	ΔV_{ICM}		-15		15	mV

LVDS DC Specifications (LVDS_25)

Table 8: LVDS DC Specifications

DC Parameter	Symbol	Conditions	Min	Typ	Max	Units
Supply Voltage	V_{CCO}		2.38	2.5	2.63	V
Output High Voltage for Q and $\overline{Q}$	V_{OH}	$R_T = 100$ Ω across Q and $\overline{Q}$ signals			1.475	V
Output Low Voltage for Q and $\overline{Q}$	V_{OL}	$R_T = 100$ Ω across Q and $\overline{Q}$ signals	0.925			V
Differential Output Voltage (Q - $\overline{Q}$), Q = High ($\overline{Q}$ - Q), $\overline{Q}$ = High	V_{ODIFF}	$R_T = 100$ Ω across Q and $\overline{Q}$ signals	250	350	400	mV
Output Common-Mode Voltage	V_{OCM}	$R_T = 100$ Ω across Q and $\overline{Q}$ signals	1.125	1.2	1.275	V
Differential Input Voltage (Q - $\overline{Q}$), Q = High ($\overline{Q}$ - Q), $\overline{Q}$ = High	V_{IDIFF}	Common-mode input voltage = 1.25V	100	350	600	mV
Input Common-Mode Voltage	V_{ICM}	Differential input voltage = ± 350 mV	0.3	1.2	2.2	V

Extended LVDS DC Specifications (LVDSEXT_25)

Table 9: Extended LVDS DC Specifications

DC Parameter	Symbol	Conditions	Min	Typ	Max	Units
Supply Voltage	V_{CCO}		2.38	2.5	2.63	V
Output High Voltage for Q and $\overline{Q}$	V_{OH}	$R_T = 100$ Ω across Q and $\overline{Q}$ signals			1.70	V
Output Low Voltage for Q and $\overline{Q}$	V_{OL}	$R_T = 100$ Ω across Q and $\overline{Q}$ signals	0.705			V
Differential Output Voltage (Q - $\overline{Q}$), Q = High ($\overline{Q}$ - Q), $\overline{Q}$ = High	V_{ODIFF}	$R_T = 100$ Ω across Q and $\overline{Q}$ signals	440		820	mV
Output Common-Mode Voltage	V_{OCM}	$R_T = 100$ Ω across Q and $\overline{Q}$ signals	1.125	1.200	1.275	V
Differential Input Voltage (Q - $\overline{Q}$), Q = High ($\overline{Q}$ - Q), $\overline{Q}$ = High	V_{IDIFF}	Common-mode input voltage = 1.25V	100		1000	mV
Input Common-Mode Voltage	V_{ICM}	Differential input voltage = ± 350 mV	0.3	1.2	2.2	V

Rocket I/O DC Input and Output Levels

Table 10: Rocket I/O DC Specifications

DC Parameter	Symbol	Conditions	Min	Typ	Max	Units
Peak-to-Peak Differential Input Voltage	DV _{IN}			175		mV
Peak-to-Peak Differential Output Voltage ^(1,2)	DV _{OUT}			800		mV
				1000		mV
				1200		mV
				1400		mV
				1600		mV

Notes:

- Output swing levels are selectable using TX_DIFF_CTRL attribute. See the **Rocket I/O Transceiver** section in Chapter 2, or refer to the *Rocket I/O User Manual* for details.
- Output preemphasis levels are selectable at 10% (default), 20%, 25%, and 33% using the TX_PREEMPHASIS attribute. See the **Rocket I/O Transceiver** section in Chapter 2 or the *Rocket I/O User Manual* for details.

Virtex-II Pro Performance Characteristics

This section provides the performance characteristics of some common functions and designs implemented in Virtex-II Pro devices. The numbers reported here are fully characterized worst-case values. Note that these values are subject to the same guidelines as **Virtex-II Pro Switching Characteristics**, page 135 (speed files).

Table 11 provides pin-to-pin values (in nanoseconds) including IOB delays; that is, delay through the device from input pin to output pin. In the case of multiple inputs and outputs, the worst delay is reported.

Table 11: Pin-to-Pin Performance

Description	Pin-to-Pin (w/ I/O delays)	Device Used & Speed Grade
Basic Functions:		
16-bit Address Decoder		
32-bit Address Decoder		
64-bit Address Decoder		
4:1 MUX		
8:1 MUX		
16:1 MUX		
32:1 MUX		
Combinatorial (pad to LUT to pad)		
Memory:		
Block RAM		
Pad to setup		
Clock to Pad		
Distributed RAM		
Pad to setup		
Clock to Pad		

Table 12 shows internal (register-to-register) performance. Values are reported in MHz.

Table 12: Register-to-Register Performance

Description	Register-to-Register Performance	Device Used & Speed Grade
Basic Functions:		
16-bit Address Decoder		
32-bit Address Decoder		
64-bit Address Decoder		
4:1 MUX		
8:1 MUX		
16:1 MUX		
32:1 MUX		
Register to LUT to Register		
8-bit Adder		
16-bit Adder		

Table 12: Register-to-Register Performance (Continued)

Description	Register-to-Register Performance	Device Used & Speed Grade
64-bit Adder		
64-bit Counter		
64-bit Accumulator		
Multiplier 18x18 (with Block RAM inputs)		
Multiplier 18x18 (with Register inputs)		
Memory:		
Block RAM		
Single-Port 4096 x 4 bits		
Single-Port 2048 x 9 bits		
Single-Port 1024 x 18 bits		
Single-Port 512 x 36 bits		
Dual-Port A:4096 x 4 bits & B:1024 x 18 bits		
Dual-Port A:1024 x 18 bits & B:1024 x 18 bits		
Dual-Port A:2048 x 9 bits & B: 512 x 36 bits		
Distributed RAM		
Single-Port 32 x 8-bit		
Single-Port 64 x 8-bit		
Single-Port 128 x 8-bit		
Dual-Port 16 x 8		
Dual-Port 32 x 8		
Dual-Port 64 x 8		
Dual-Port 128 x 8		
Shift Registers		
128-bit SRL		
256-bit SRL		
FIFOs (Async. in Block RAM)		
1024 x 18-bit		
1024 x 18-bit		
FIFOs (Sync. in SRL)		
128 x 8-bit		
128 x 16-bit		
CAMs in Block RAM		
32 x 9-bit		
64 x 9-bit		
128 x 9-bit		
256 x 9-bit		

Table 12: Register-to-Register Performance (Continued)

Description	Register-to-Register Performance	Device Used & Speed Grade
CAMs in SRL		
32 x 16-bit		
64 x 32-bit		
128 x 40-bit		
256 x 48-bit		
1024 x 16-bit		
1024 x 72-bit		

Virtex-II Pro Switching Characteristics

Switching characteristics are specified on a per-speed-grade basis and can be designated as Advance, Preliminary, or Production. Note that **Virtex-II Pro Performance Characteristics**, page 133 are subject to these guidelines, as well. Each designation is defined as follows:

Advance: These speed files are based on simulations only and are typically available soon after device design specifications are frozen. Although speed grades with this designation are considered relatively stable and conservative, some under-reporting might still occur.

Preliminary: These speed files are based on complete ES (engineering sample) silicon characterization. Devices and speed grades with this designation are intended to give a better indication of the expected performance of production silicon. The probability of under-reporting delays is greatly reduced as compared to Advance data.

Production: These speed files are released once enough production silicon of a particular device family member has been characterized to provide full correlation between speed files and devices over numerous production lots. There is no under-reporting of delays, and customers receive formal notification of any subsequent changes. Typically, the slowest speed grades transition to Production before faster speed grades.

Since individual family members are produced at different times, the migration from one category to another depends completely on the status of the fabrication process for each

device. Table 13 correlates the current status of each Virtex-II Pro device with a corresponding speed file designation.

All specifications are always representative of worst-case supply voltage and junction temperature conditions.

Table 13: Virtex-II Pro Device Speed Grade Designations

Device	Speed Grade Designations		
	Advance	Preliminary	Production
XC2VP2	-8, -7, -6		
XC2VP4	-8, -7, -6		
XC2VP7	-8, -7, -6		
XC2VP20	-8, -7, -6		
XC2VP50	-8, -7, -6		

Testing of Switching Characteristics

All devices are 100% functionally tested. Internal timing parameters are derived from measuring internal test patterns. Listed below are representative values. For more specific, more precise, and worst-case guaranteed data, use the values reported by the static timing analyzer (TRCE in the Xilinx Development System) and back-annotate to the simulation net list. Unless otherwise noted, values apply to all Virtex-II Pro devices.

PowerPC Switching Characteristics

Table 14: Processor Clocks Absolute AC Characteristics

	Speed Grade						
	-8		-7		-6		
Description	Min	Max	Min	Max	Min	Max	Units
CPMC405CLOCK frequency							MHz
JTAGC405TCK frequency ⁽¹⁾							MHz

Notes:

1. The theoretical maximum frequency of this clock is one-half the CPMC405CLOCK. However, the achievable maximum is dependent on the system, and will be much less

Table 15: Processor Block Switching Characteristics

		Speed Grade			Units
Description	Symbol	-8	-7	-6	
Setup and Hold Relative to Clock (CPMC405CLOCK)					
Device Control Register Bus control inputs	T_{PCK_DCR}/T_{PCKC_DCR}				ns, min
Device Control Register Bus data inputs	$T_{PDCK\_DCR}/T_{PCKD\_DCR}$				ns, min
Clock and Power Management control inputs	T_{PCK_CPM}/T_{PCKC_CPM}				ns, min
Reset control inputs	T_{PCK_RST}/T_{PCKC_RST}				ns, min
Debug control inputs	T_{PCK_DBG}/T_{PCKC_DBG}				ns, min
Trace control inputs	T_{PCK_TRC}/T_{PCKC_TRC}				ns, min
External Interrupt Controller control inputs	T_{PCK_EIC}/T_{PCKC_EIC}				ns, min
Clock to Out					
Device Control Register Bus control outputs	T_{PCKCO_DCR}				ns, max
Device Control Register Bus address outputs	T_{PCKAO_DCR}				ns, max
Device Control Register Bus data outputs	T_{PCKDO_DCR}				ns, max
Clock and Power Management control outputs	T_{PCKCO_CPM}				ns, max
Reset control outputs	T_{PCKCO_RST}				ns, max
Debug control outputs	T_{PCKCO_DBG}				ns, max
Trace control outputs	T_{PCKCO_TRC}				ns, max

Table 15: Processor Block Switching Characteristics (Continued)

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Clock					
CPMC405CLOCK minimum pulse width, high	T _{CPWH}				ns, min
CPMC405CLOCK minimum pulse width, low	T _{CPWL}				ns, min

Table 16: Processor Block PLB Switching Characteristics

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Setup and Hold Relative to Clock (PLBCLK)					
Processor Local Bus(ICU/DCU) control inputs	T _{PCKC} _PLB/T _{PCKC} _PLB				ns, min
Processor Local Bus (ICU/DCU) data inputs	T _{PDCK} _PLB/T _{PCKD} _PLB				ns, min
Clock to Out					
Processor Local Bus(ICU/DCU) control outputs	T _{PCKCO} _PLB				ns, max
Processor Local Bus(ICU/DCU) address bus outputs	T _{PCKAO} _PLB				ns, max
Processor Local Bus(ICU/DCU) data bus outputs	T _{PCKDO} _PLB				ns, max
Clock					
PLBCLK minimum pulse width, high	T _{PPWH}				ns, min
PLBCLK minimum pulse width, low	T _{PPWL}				ns, min

Table 17: Processor Block JTAG Switching Characteristics

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Setup and Hold Relative to Clock (JTAGC405TCK)					
JTAG control inputs	T _{PCKC_JTAG} /T _{PCKC_JTAG}				ns, min
JTAG reset input	T _{PCKC_JTAGRST} / T _{PCKC_JTAGRST}				ns, min
Clock to Out					
JTAG control outputs	T _{PCKCO_JTAG}				ns, max

Table 17: Processor Block JTAG Switching Characteristics (Continued)

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Clock					
JTAGC405TCK minimum pulse width, high	T _{JPWH}				ns, min
JTAGC405TCK minimum pulse width, low	T _{JPWL}				ns, min

Table 18: PowerPC 405 Data-Side On-Chip Memory Switching Characteristics

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Setup and Hold Relative to Clock (BRAMDSOCCLK)					
Data-Side On-Chip Memory data bus inputs	T _{PDCK_DSOCM} /T _{PCKD_DSOCM}				ns, min
Clock to Out					
Data-Side On-Chip Memory control outputs	T _{PCKCO_DSOCM}				ns, max
Data-Side On-Chip Memory address bus outputs	T _{PCKAO_DSOCM}				ns, max
Data-Side On-Chip Memory data bus outputs	T _{PCKDO_DSOCM}				ns, max
Clock					
BRAMDSOCCLK minimum pulse width, high	T _{DPWH}				ns, min
BRAMDSOCCLK minimum pulse width, low	T _{DPWL}				ns, min

Table 19: PowerPC 405 Instruction-Side On-Chip Memory Switching Characteristics

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Setup and Hold Relative to Clock (BRAMISOCCLK)					
Instruction-Side On-Chip Memory data bus inputs	T _{PDCK_ISOCM} /T _{PCKD_ISOCM}				ns, min
Clock to Out					
Instruction-Side On-Chip Memory control outputs	T _{PCKCO_ISOCM}				ns, max
Instruction-Side On-Chip Memory address bus outputs	T _{PCKAO_ISOCM}				ns, max
Instruction-Side On-Chip Memory data bus outputs	T _{PCKDO_ISOCM}				ns, max

Table 19: PowerPC 405 Instruction-Side On-Chip Memory Switching Characteristics (Continued)

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Clock					
BRAMISOCMCLK minimum pulse width, high	T _{IPWH}				ns, min
BRAMISOCMCLK minimum pulse width, low	T _{IPWL}				ns, min

Rocket I/O Switching Characteristics

Table 20: Rocket I/O Reference Clock Switching Characteristics

Description	Symbol	Conditions	All Speed Grades			Units
			Min	Typ	Max	
REFCLK frequency range ⁽¹⁾	F_{GCLK}		40	Note (1)	156.25	MHz
REFCLK frequency tolerance	F_{GTOL}				±100	ppm
REFCLK rise time	T_{RCLK}	20% – 80%				ns
REFCLK fall time	T_{FCLK}	20% – 80%				ns
REFCLK duty cycle	T_{DCREF}		45	50	55	%
REFCLK total jitter	T_{GJTT}	peak-to-peak			40	ps
Clock recovery frequency acquisition time	T_{LOCK}			10		μs
Clock recovery phase acquisition time	T_{PHASE}			960		bits
Bit error rate	BER				10 ⁻¹²	

Notes:

- REFCLK frequency is typically 1/20 of serial data rate.

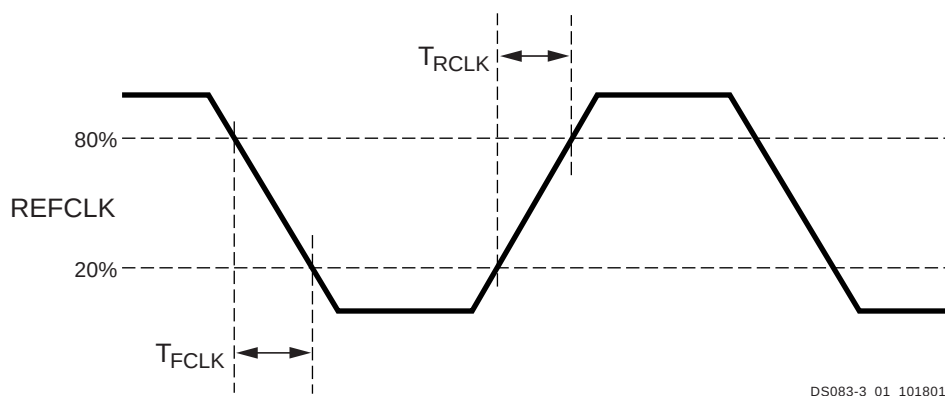


Figure 1: Reference Clock (REFCLK) Timing Parameters

Table 21: Rocket I/O Receiver Switching Characteristics

Description	Symbol	Conditions	Min	Typ	Max	Units
Receive total jitter tolerance	T_{JTOL}				0.65	UI ⁽¹⁾
Receive deterministic jitter tolerance	T_{DJTOL}				0.41	UI
Receive latency ⁽²⁾	T_{RXLAT}			25	42	RXUSR CLK cycles
RXUSRCLK duty cycle	T_{RXDC}		45	50	55	%
RXUSRCLK2 duty cycle	T_{RX2DC}		45	50	55	%
Bit error rate	BER				10 ⁻¹²	

Notes:

1. UI = Unit Interval
2. Receive latency delay from RXP/RXN to RXDATA

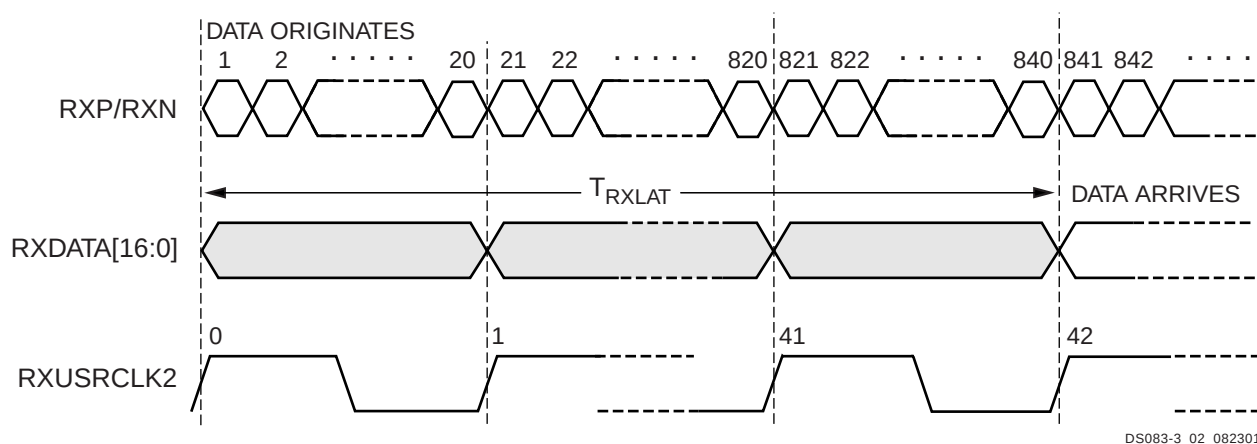


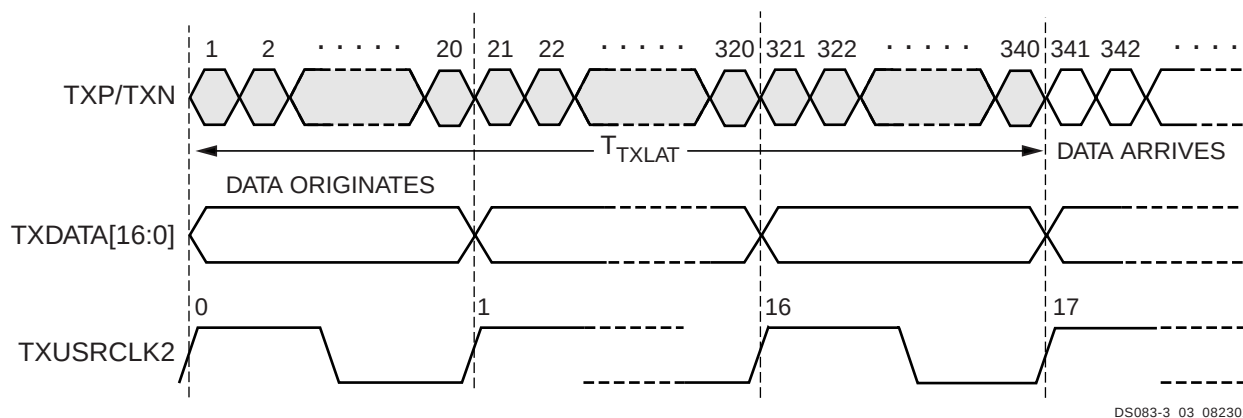
Figure 2: Receive Latency (Maximum)

Table 22: Rocket I/O Transmitter Switching Characteristics

Description	Symbol	Conditions	Min	Typ	Max	Units
Serial data rate, full-speed clock	F _{GTX}	Flipchip packages	0.800		3.125	Gb/s
		Wirebond packages	0.800		2.5	Gb/s
Serial data rate, half-speed clock		Flipchip packages	0.600		1.0	Gb/s
		Wirebond packages	0.600		1.0	Gb/s
Serial data output deterministic jitter	T _{DJ}				0.18	UI ⁽¹⁾
Serial data output random jitter	T _{RJ}				0.17	UI
TX rise time	T _{RTX}	20% – 80%		120		ps
TX fall time	T _{FTX}			120		ps
Transmit latency ⁽²⁾	T _{TXLAT}	Including CRC		14	17	TXUSR CLK cycles
		Excluding CRC		8	11	
TXUSRCLK duty cycle	T _{TXDC}		45	50	55	%
TXUSRCLK2 duty cycle	T _{TX2DC}		45	50	55	%

Notes:

1. UI = Unit Interval
2. Transmit latency delay from TXDATA to TXP/TXN



DS083-3_03_082301

Figure 3: Transmit Latency (Maximum, Including CRC)

Table 23: Rocket I/O RXUSRCLK Switching Characteristics

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Setup and Hold Relative to Clock (RXUSRCLK)					
CHBONDI control inputs	T _{GCCK_CHBI} /T _{GCKC_CHBI}				ns, min
Clock to Out					
CHBONDO control outputs	T _{GCKCO_CHBO}				ns, max
Clock					
RXUSRCLK minimum pulse width, High	T _{GPWH_RX}				ns, min
RXUSRCLK minimum pulse width, Low	T _{GPWL_RX}				ns, min

Table 24: Rocket I/O RXUSRCLK2 Switching Characteristics

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Setup and Hold Relative to Clock (RXUSRCLK2)					
RXRESET control input	T _{GCKC_RRST} /T _{GCKC_RRST}				ns, min
RXPOLARITY control input	T _{GCKC_RPOL} /T _{GCKC_RPOL}				ns, min
ENCHANSYNC control input	T _{GCKC_ECSY} /T _{GCKC_ECSY}				ns, min
Clock to Out					
RXNOTINTABLE status outputs	T _{GCKST_RNIT}				ns, max
RXDISPERR status outputs	T _{GCKST_RDERR}				ns, max
RXCHARISCOMMA status outputs	T _{GCKST_RCMCH}				ns, max
RXREALIGN status output	T _{GCKST_ALIGN}				ns, max
RXCOMMADET status output	T _{GCKST_CMDT}				ns, max
RXLOSSOFSYNC status outputs	T _{GCKST_RLOS}				ns, max
RXCLKCORCNT status outputs	T _{GCKST_RCCCNT}				ns, max
RXBUFSTATUS status outputs	T _{GCKST_RBSTA}				ns, max
RXCHECKINGCRC status output	T _{GCKST_RCCRC}				ns, max
RXRCRCERR status output	T _{GCKST_RCRCE}				ns, max
CHBONDDONE status output	T _{GCKST_CHBD}				ns, max
RXCHARISK status outputs	T _{GCKST_RKCH}				ns, max
RXRUNDISP status outputs	T _{GCKST_RRDIS}				ns, max
RXDATA data outputs	T _{GCKDO_RDAT}				ns, max
Clock					
RXUSRCLK2 minimum pulse width, High	T _{GPWH_RX2}				ns, min
RXUSRCLK2 minimum pulse width, Low	T _{GPWL_RX2}				ns, min

Table 25: Rocket I/O TXUSRCLK Switching Characteristics

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Setup and Hold Relative to Clock (TXUSRCLK2)					
CONFIGENABLE control input	T _{GCKK_CFGEN} /T _{GCKC_CFGEN}				ns, min
TXBYPASS8B10B control inputs	T _{GCKK_TBYP} /T _{GCKC_TBYP}				ns, min
TXFORCECRCERR control input	T _{GCKK_TCRCE} /T _{GCKC_TCRCE}				ns, min
TXPOLARITY control input	T _{GCKK_TPOL} /T _{GCKC_TPOL}				ns, min
TXINHIBIT control inputs	T _{GCKK_TINH} /T _{GCKC_TINH}				ns, min
LOOPBACK control inputs	T _{GCKK_LBK} /T _{GCKC_LBK}				ns, min
TXRESET control input	T _{GCKK_TRST} /T _{GCKC_TRST}				ns, min
TXCHARISK control inputs	T _{GCKK_TKCH} /T _{GCKC_TKCH}				ns, min
TXCHARDISPMODE control inputs	T _{GCKK_TCDM} /T _{GCKC_TCDM}				ns, min
TXCHARDISPVAL control inputs	T _{GCKK_TCDV} /T _{GCKC_TCDV}				ns, min
CONFIGIN data input	T _{GDCK_CFGIN} /T _{GCKD_CFGIN}				ns, min
TXDATA data inputs	T _{GDCK_TDAT} /T _{GCKD_TDAT}				ns, min
Clock to Out					
TXBUFERR status output	T _{GCKST_TBERR}				ns, max
TXKERR status outputs	T _{GCKST_TKERR}				ns, max
TXRUNDISP status outputs	T _{GCKST_TRDIS}				ns, max
CONFIGOUT data output	T _{GCKDO_CFGOUT}				ns, max
Clock					
TXUSRCLK minimum pulse width, High	T _{GPWH_TX}				ns, min
TXUSRCLK minimum pulse width, Low	T _{GPWL_TX}				ns, min
TXUSRCLK2 minimum pulse width, High	T _{GPWH_TX2}				ns, min
TXUSRCLK2 minimum pulse width, Low	T _{GPWL_TX2}				ns, min

IOB Input Switching Characteristics

Input delays associated with the pad are specified for LVCMOS 2.5V levels. For other standards, adjust the delays with the values shown in **IOB Input Switching Characteristics Standard Adjustments**, page 145.

Table 26: IOB Input Switching Characteristics

			Speed Grade			
Description	Symbol	Device	–8	–7	–6	Units
Propagation Delays						
Pad to I output, no delay	T _{IOPI}	All				ns, max
Pad to I output, with delay	T _{IOPID}	XC2VP2				ns, max
		XC2VP4				ns, max
		XC2VP7				ns, max
		XC2VP20				ns, max
		XC2VP50				ns, max
Propagation Delays						
Pad to output IQ via transparent latch, no delay	T _{IOPLI}	All				ns, max
Pad to output IQ via transparent latch, with delay	T _{IOPLID}	XC2VP2				ns, max
		XC2VP4				ns, max
		XC2VP7				ns, max
		XC2VP20				ns, max
		XC2VP50				ns, max
Clock CLK to output IQ	T _{IOCKIQ}	All				ns, max
Setup and Hold Times With Respect to Clock at IOB Input Register						
Pad, no delay	T _{IO PICK} /T _{IOICKP}	All				ns, min
Pad, with delay	T _{IO PICKD} /T _{IOICKPD}	XC2VP2				ns, max
		XC2VP4				ns, max
		XC2VP7				ns, max
		XC2VP20				ns, max
		XC2VP50				ns, max
ICE input	T _{IOICECK} /T _{IOICKICE}	All				ns, min
SR input (IFF, synchronous)	T _{IOSRCKI}	All				ns, min
Set/Reset Delays						
SR input to IQ (asynchronous)	T _{IOSRIQ}	All				ns, max
GSR to output IQ	T _{GSRQ}	All				ns, max

Notes:

1. Input timing for LVCMOS25 is measured at 1.25V. For other I/O standards, see Table 30.

IOB Input Switching Characteristics Standard Adjustments

Table 27: IOB Input Switching Characteristics Standard Adjustments

			Speed Grade			
Description	Symbol	Standard	−8	−7	−6	Units
Data Input Delay Adjustments						
Standard-specific data input delay adjustments	T _{ILVTTL}	LVTTL				ns
	T _{ILVCMOS33}	LVC MOS33				ns
	T _{ILVCMOS25}	LVC MOS25				ns
	T _{ILVCMOS18}	LVC MOS18				ns
	T _{ILVCMOS15}	LVC MOS15				ns
	T _{ILVDS_25}	LVDS_25				ns
	T _{ILVDS_25_EXT}	LVDS_25_EXT				ns
	T _{IPCI33_3}	PCI, 33 MHz, 3.3V				ns
	T _{IPCI66_3}	PCI, 66 MHz, 3.3V				ns
	T _{IGTL}	GTL				ns
	T _{IGTLPLUS}	GTLP				ns
	T _{IHSTL_I}	HSTL I				ns
	T _{IHSTL_II}	HSTL II				ns
	T _{IHSTL_III}	HSTL III				ns
	T _{IHSTL_IV}	HSTL IV				ns
	T _{IHSTL_I_18}	HSTL_I_18				ns
	T _{IHSTL_II_18}	HSTL_II_18				ns
	T _{IHSTL_III_18}	HSTL_III_18				ns
	T _{IHSTL_IV_18}	HSTL_IV_18				ns
	T _{ISSTL2_I}	SSTL2 I				ns
	T _{ISSTL2_II}	SSTL2 II				ns
	T _{ISSTL3_I}	SSTL3 I				ns
	T _{ISSTL3_II}	SSTL3 II				ns
	T _{ILVDCI33}	LVDCI_33				ns
	T _{ILVDCI25}	LVDCI_25				ns
	T _{ILVDCI18}	LVDCI_18				ns
	T _{ILVDCI15}	LVDCI_15				ns
	T _{ILVDCI_DV2_25}	LVDCI_DV2_25				ns
	T _{ILVDCI_DV2_18}	LVDCI_DV2_18				ns
	T _{ILVDCI_DV2_15}	LVDCI_DV2_15				ns
	T _{IGTL_DCI}	GTL_DCI				ns
	T _{IGTLP_DCI}	GTLP_DCI				ns
	T _{IHSTL_I_DCI}	HSTL_I_DCI				ns

Table 27: IOB Input Switching Characteristics Standard Adjustments (Continued)

Description	Symbol	Standard	Speed Grade			Units
			-8	-7	-6	
Standard-specific data input delay adjustments (continued)	$T_{IHSTL\_II\_DCI}$	HSTL_II_DCI				ns
	$T_{IHSTL\_III\_DCI}$	HSTL_III_DCI				ns
	$T_{IHSTL\_IV\_DCI}$	HSTL_IV_DCI				ns
	$T_{IHSTL\_I\_DCI\_18}$	HSTL_I_DCI_18				ns
	$T_{IHSTL\_II\_DCI\_18}$	HSTL_II_DCI_18				ns
	$T_{IHSTL\_III\_DCI\_18}$	HSTL_III_DCI_18				ns
	$T_{IHSTL\_IV\_DCI\_18}$	HSTL_IV_DCI_18				ns
	$T_{ISSTL2\_I\_DCI}$	SSTL2_I_DCI				ns
	$T_{ISSTL2\_II\_DCI}$	SSTL2_II_DCI				ns
	$T_{ISSTL3\_I\_DCI}$	SSTL3_I_DCI				ns
	$T_{ISSTL3\_II\_DCI}$	SSTL3_II_DCI				ns
	T_{ILDT_25}	LDT_25				ns
	T_{IULVDS_25}	ULVDS_25				ns

Notes:

1. Input timing for LVTTTL is measured at 1.4V. For other I/O standards, see [Table 30](#).

IOB Output Switching Characteristics

Output delays terminating at a pad are specified for LVCMOS25 with 12 mA drive and fast slew rate. For other standards, adjust the delays with the values shown in **IOB Output Switching Characteristics Standard Adjustments**, [page 147](#).

Table 28: IOB Output Switching Characteristics

		Speed Grade			
Description	Symbol	–8	–7	–6	Units
Propagation Delays					
O input to Pad	T _{IOOP}				ns, max
O input to Pad via transparent latch	T _{IOOLP}				ns, max
3-State Delays					
T input to Pad high-impedance ⁽²⁾	T _{IOTHZ}				ns, max
T input to valid data on Pad	T _{IOTON}				ns, max
T input to Pad high-impedance via transparent latch ⁽²⁾	T _{IOTLPHZ}				ns, max
T input to valid data on Pad via transparent latch	T _{IOTLPON}				ns, max
GTS to Pad high-impedance ⁽²⁾	T _{GTS}				ns, max
Sequential Delays					
Clock CLK to Pad	T _{IOCKP}				ns, max
Clock CLK to Pad high-impedance (synchronous) ⁽²⁾	T _{IOCKHZ}				ns, max
Clock CLK to valid data on Pad (synchronous)	T _{IOCKON}				ns, max

Table 28: IOB Output Switching Characteristics (Continued)

		Speed Grade			
Description	Symbol	–8	–7	–6	Units
Setup and Hold Times Before/After Clock CLK					
O input	T _{IOOCK} /T _{IOCKO}				ns, min
OCE input	T _{IOOCECK} /T _{IOCKOCE}				ns, min
SR input (OFF)	T _{IOSRCKO} /T _{IOCKOSR}				ns, min
3-State Setup Times, T input	T _{IOTCK} /T _{IOCKT}				ns, min
3-State Setup Times, TCE input	T _{IOTCECK} /T _{IOCKTCE}				ns, min
3-State Setup Times, SR input (TFF)	T _{IOSRCKT} /T _{IOCKTSR}				ns, min
Set/Reset Delays					
SR input to Pad (asynchronous)	T _{IOSRP}				ns, max
SR input to Pad high-impedance (asynchronous) ⁽²⁾	T _{IOSRHZ}				ns, max
SR input to valid data on Pad (asynchronous)	T _{IOSRON}				ns, max
GSR to Pad	T _{IOGSRQ}				ns, max

Notes:

1. A Zero “0” Hold Time listing indicates no hold time or a negative hold time. Negative values can not be guaranteed “best-case”, but if a “0” is listed, there is no positive hold time.
2. The 3-state turn-off delays should not be adjusted.

IOB Output Switching Characteristics Standard Adjustments

Output delays terminating at a pad are specified for LVCMOS25 with 12 mA drive and fast slew rate. For other standards, adjust the delays by the values shown.

Table 29: IOB Output Switching Characteristics Standard Adjustments

			Speed Grade			
Description	Symbol	Standard	–8	–7	–6	Units
Output Delay Adjustments						
Standard-specific adjustments for output delays terminating at pads (based on standard capacitive load, Csl)	T _{OLVTTL_S2}	LVTTL, Slow, 2 mA				ns
	T _{OLVTTL_S4}	4 mA				ns
	T _{OLVTTL_S6}	6 mA				ns
	T _{OLVTTL_S8}	8 mA				ns
	T _{OLVTTL_S12}	12 mA				ns
	T _{OLVTTL_S16}	16 mA				ns
	T _{OLVTTL_S24}	24 mA				ns
	T _{OLVTTL_F2}	LVTTL, Fast, 2 mA				ns
	T _{OLVTTL_F4}	4 mA				ns
	T _{OLVTTL_F6}	6 mA				ns
	T _{OLVTTL_F8}	8 mA				ns
	T _{OLVTTL_F12}	12 mA				ns

Table 29: IOB Output Switching Characteristics Standard Adjustments (Continued)

Description	Symbol	Standard	Speed Grade			Units
			–8	–7	–6	
Standard-specific adjustments for output delays terminating at pads (based on standard capacitive load, Csl) (continued)	T _{OLVTTL_F16}	16 mA				ns
	T _{OLVTTL_F24}	24 mA				ns
	T _{OLVDS_25}	LVDS				ns
	T _{OLVDSEXT_25}	LVDS				ns
	T _{OLDT_25}	LDT				ns
	T _{OBLVDS_25}	BLVDS				ns
	T _{OULVDS_25}	ULVDS				ns
	T _{OPCI33_3}	PCI, 33 MHz, 3.3V				ns
	T _{OPCI66_3}	PCI, 66 MHz, 3.3V				ns
	T _{OGTL}	GTL				ns
	T _{OGTLP}	GTLP				ns
	T _{OHSTL_I}	HSTL I				ns
	T _{OHSTL_II}	HSTL II				ns
	T _{OHSTL_III}	HSTL III				ns
	T _{OHSTL_IV}	HSTL IV				ns
	T _{OHSTL_I_18}	HSTL_I_18				ns
	T _{OHSTL_II_18}	HSTL_II_18				ns
	T _{OHSTL_III_18}	HSTL_III_18				ns
	T _{OHSTL_IV_18}	HSTL_IV_18				ns
	T _{OSSTL2_I}	SSTL2 I				ns
	T _{OSSTL2_II}	SSTL2 II				ns
	T _{OSSTL3_I}	SSTL3 I				ns
	T _{OSSTL3_II}	SSTL3 II				ns
	T _{OLVCMOS33_S2}	LVC MOS33, Slow, 2 mA				ns
	T _{OLVCMOS33_S4}	4 mA				ns
	T _{OLVCMOS33_S6}	6 mA				ns
	T _{OLVCMOS33_S8}	8 mA				ns
	T _{OLVCMOS33_S12}	12 mA				ns
	T _{OLVCMOS33_S16}	16 mA				ns
	T _{OLVCMOS33_S24}	24 mA				ns
	T _{OLVCMOS33_F2}	LVC MOS33, Fast, 2 mA				ns
	T _{OLVCMOS33_F4}	4 mA				ns
	T _{OLVCMOS33_F6}	6 mA				ns
	T _{OLVCMOS33_F8}	8 mA				ns
	T _{OLVCMOS33_F12}	12 mA				ns

Table 29: IOB Output Switching Characteristics Standard Adjustments (Continued)

Description	Symbol	Standard	Speed Grade			Units
			–8	–7	–6	
Standard-specific adjustments for output delays terminating at pads (based on standard capacitive load, Csl) (continued)	T _{OLVCMOS33_F16}	16 mA				ns
	T _{OLVCMOS33_F24}	24 mA				ns
	T _{OLVCMOS25_S2}	LVC MOS25, Slow, 2 mA				ns
	T _{OLVCMOS25_S4}	4 mA				ns
	T _{OLVCMOS25_S6}	6 mA				ns
	T _{OLVCMOS25_S8}	8 mA				ns
	T _{OLVCMOS25_S12}	12 mA				ns
	T _{OLVCMOS25_S16}	16 mA				ns
	T _{OLVCMOS25_S24}	24 mA				ns
	T _{OLVCMOS25_F2}	LVC MOS25, Fast, 2 mA				ns
	T _{OLVCMOS25_F4}	4 mA				ns
	T _{OLVCMOS25_F6}	6 mA				ns
	T _{OLVCMOS25_F8}	8 mA				ns
	T _{OLVCMOS25_F12}	12 mA				ns
	T _{OLVCMOS25_F16}	16 mA				ns
	T _{OLVCMOS25_F24}	24 mA				ns
	T _{OLVCMOS18_S2}	LVC MOS18, Slow, 2 mA				ns
	T _{OLVCMOS18_S4}	4 mA				ns
	T _{OLVCMOS18_S6}	6 mA				ns
	T _{OLVCMOS18_S8}	8 mA				ns
	T _{OLVCMOS18_S12}	12 mA				ns
	T _{OLVCMOS18_S16}	16 mA				ns
	T _{OLVCMOS18_F2}	LVC MOS18, Fast, 2 mA				ns
	T _{OLVCMOS18_F4}	4 mA				ns
	T _{OLVCMOS18_F6}	6 mA				ns
	T _{OLVCMOS18_F8}	8 mA				ns
	T _{OLVCMOS18_F12}	12 mA				ns
	T _{OLVCMOS18_F16}	16 mA				ns
	T _{OLVCMOS15_S2}	LVC MOS15, Slow, 2 mA				ns
	T _{OLVCMOS15_S4}	4 mA				ns
	T _{OLVCMOS15_S6}	6 mA				ns
	T _{OLVCMOS15_S8}	8 mA				ns
	T _{OLVCMOS15_S12}	12 mA				ns
	T _{OLVCMOS15_S16}	16 mA				ns

Table 29: IOB Output Switching Characteristics Standard Adjustments (Continued)

Description	Symbol	Standard	Speed Grade			Units
			-8	-7	-6	
Standard-specific adjustments for output delays terminating at pads (based on standard capacitive load, Csl) (continued)	T _{OLVCMOS15_F2}	LVC MOS15, Fast, 2 mA				ns
	T _{OLVCMOS15_F4}	4 mA				ns
	T _{OLVCMOS15_F6}	6 mA				ns
	T _{OLVCMOS15_F8}	8 mA				ns
	T _{OLVCMOS15_F12}	12 mA				ns
	T _{OLVCMOS15_F16}	16 mA				ns
	T _{OLVDCI33}	LVDCI_33				ns
	T _{OLVDCI25}	LVDCI_25				ns
	T _{OLVDCI18}	LVDCI_18				ns
	T _{OLVDCI15}	LVDCI_15				ns
	T _{OLVDCI_DV2_25}	LVDCI_DV2_25				ns
	T _{OLVDCI_DV2_18}	LVDCI_DV2_18				ns
	T _{OLVDCI_DV2_15}	LVDCI_DV2_15				ns
	T _{OGTL_DCI}	GTL_DCI				ns
	T _{OGTLP_DCI}	GTL_P_DCI				ns
	T _{OHSTL_I_DCI}	HSTL_I_DCI				ns
	T _{OHSTL_II_DCI}	HSTL_II_DCI				ns
	T _{OHSTL_III_DCI}	HSTL_III_DCI				ns
	T _{OHSTL_IV_DCI}	HSTL_IV_DCI				ns
	T _{OHSTL_I_DCI_18}	HSTL_I_DCI_18				ns
	T _{OHSTL_II_DCI_18}	HSTL_II_DCI_18				ns
	T _{OHSTL_III_DCI_18}	HSTL_III_DCI_18				ns
	T _{OHSTL_IV_DCI_18}	HSTL_IV_DCI_18				ns
	T _{OSSTL2_I_DCI}	SSTL2_I_DCI				ns
	T _{OSSTL2_II_DCI}	SSTL2_II_DCI				ns
	T _{OSSTL3_I_DCI}	SSTL3_I_DCI				ns
	T _{OSSTL3_II_DCI}	SSTL3_II_DCI				ns

Table 30: Delay Measurement Methodology

Standard	$V_L^{(1)}$	$V_H^{(1)}$	Meas. Point	$V_{REF} \text{ (Typ)}^{(2)}$
LVTTL	0	3	1.4	–
LVC MOS33	0	3.3	1.65	–
LVC MOS25	0	2.5	1.25	–
LVC MOS18	0	1.8	0.9	–
LVC MOS15	0	1.5	0.75	–
PCI33_3	Per PCI Specification			–
PCI66_3	Per PCI Specification			–
GTL	$V_{REF} - 0.2$	$V_{REF} + 0.2$	V_{REF}	0.80
GTLP	$V_{REF} - 0.2$	$V_{REF} + 0.2$	V_{REF}	1.0
HSTL Class I	$V_{REF} - 0.5$	$V_{REF} + 0.5$	V_{REF}	0.75
HSTL Class II	$V_{REF} - 0.5$	$V_{REF} + 0.5$	V_{REF}	0.75
HSTL Class III	$V_{REF} - 0.5$	$V_{REF} + 0.5$	V_{REF}	0.90
HSTL Class IV	$V_{REF} - 0.5$	$V_{REF} + 0.5$	V_{REF}	0.90
HSTL Class I (1.8V)	$V_{REF} - 0.5$	$V_{REF} + 0.5$	V_{REF}	1.08
HSTL Class II (1.8V)	$V_{REF} - 0.5$	$V_{REF} + 0.5$	V_{REF}	1.08
HSTL Class III (1.8V)	$V_{REF} - 0.5$	$V_{REF} + 0.5$	V_{REF}	1.08
HSTL Class IV (1.8V)	$V_{REF} - 0.5$	$V_{REF} + 0.5$	V_{REF}	1.08
SSTL3 I & II	$V_{REF} - 1.0$	$V_{REF} + 1.0$	V_{REF}	1.5
SSTL2 I & II	$V_{REF} - 0.75$	$V_{REF} + 0.75$	V_{REF}	1.25
LVDS_25	$1.2 - 0.125$	$1.2 + 0.125$	1.2	
LVDSEXT_25	$1.2 - 0.125$	$1.2 + 0.125$	1.2	
ULVDS_25	$0.6 - 0.125$	$0.6 + 0.125$	0.6	
LDT_25	$0.6 - 0.125$	$0.6 + 0.125$	0.6	

Notes:

1. Input waveform switches between V_L and V_H .
2. Measurements are made at $V_{REF} \text{ (Typ)}$, Maximum, and Minimum. Worst-case values are reported.

Table 31: Standard Capacitive Loads

Standard	Csl (pF)
LVTTL Fast Slew Rate, 2mA drive	35
LVTTL Fast Slew Rate, 4mA drive	35
LVTTL Fast Slew Rate, 6mA drive	35
LVTTL Fast Slew Rate, 8mA drive	35
LVTTL Fast Slew Rate, 12mA drive	35
LVTTL Fast Slew Rate, 16mA drive	35
LVTTL Fast Slew Rate, 24mA drive	35
LVTTL Slow Slew Rate, 2mA drive	35
LVTTL Slow Slew Rate, 4mA drive	35
LVTTL Slow Slew Rate, 6mA drive	35
LVTTL Slow Slew Rate, 8mA drive	35
LVTTL Slow Slew Rate, 12mA drive	35
LVTTL Slow Slew Rate, 16mA drive	35
LVTTL Slow Slew Rate, 24mA drive	35
LVC MOS33	35
LVC MOS25	35
LVC MOS18	35
LVC MOS15	35
PCI 33MHZ 3.3V	10
PCI 66 MHz 3.3V	10
GTL	0
GTLP	0
HSTL Class I (1.5V and 1.8V)	20
HSTL Class II (1.5V and 1.8V)	20
HSTL Class III (1.5V and 1.8V)	20
HSTL Class IV 1.5V and 1.8V	20
SSTL2 Class I	30
SSTL2 Class II	30
SSTL3 Class I	30
SSTL3 Class II	30

Notes:

1. I/O parameter measurements are made with the capacitance values shown above.
2. I/O standard measurements are reflected in the IBIS model information except where the IBIS format precludes it.
3. Use of IBIS models results in a more accurate prediction of the propagation delay:
 - a. Model the output in an IBIS simulation into the standard capacitive load.
 - b. Record the relative time to the V_{OH} or V_{OL} transition of interest.
 - c. Remove the capacitance, and model the actual PCB traces (transmission lines) and actual loads from the appropriate IBIS models for driven devices.
 - d. Record the results from the new simulation.
 - e. Compare with the capacitance simulation. The increase or decrease in delay from the capacitive load delay simulation should be added or subtracted from the value above to predict the actual delay.

Clock Distribution Switching Characteristics

Table 32: Clock Distribution Switching Characteristics

Description	Symbol	Speed Grade			Units
		-8	-7	-6	
Global Clock Buffer I input to O output	T _{GIO}				ns, max

CLB Switching Characteristics

Delays originating at F/G inputs vary slightly according to the input used (see Figure 22 in Data Sheet Module 1). The values listed below are worst-case. Precise values are provided by the timing analyzer.

Table 33: CLB Switching Characteristics

		Speed Grade			
Description	Symbol	–8	–7	–6	Units
Combinatorial Delays					
4-input function: F/G inputs to X/Y outputs	T _{ILO}				ns, max
5-input function: F/G inputs to F5 output	T _{IF5}				ns, max
5-input function: F/G inputs to X output	T _{IF5X}				ns, max
FXINA or FXINB inputs to Y output via MUXFX	T _{IFXY}				ns, max
FXINA input to FX output via MUXFX	T _{INAFX}				ns, max
FXINB input to FX output via MUXFX	T _{INBFX}				ns, max
SOPIN input to SOPOUT output via ORCY	T _{SOPSOP}				ns, max
Incremental delay routing through transparent latch to XQ/YQ outputs	T _{IFNCTL}				ns, max
Sequential Delays					
FF Clock CLK to XQ/YQ outputs	T _{CKO}				ns, max
Latch Clock CLK to XQ/YQ outputs	T _{CKLO}				ns, max
Setup and Hold Times Before/After Clock CLK					
BX/BY inputs	T _{DICK} /T _{CKDI}				ns, min
DY inputs	T _{DYCK} /T _{CKDY}				ns, min
DX inputs	T _{DXCK} /T _{CKDX}				ns, min
CE input	T _{CECK} /T _{CKCE}				ns, min
SR/BY inputs (synchronous)	T _{RCK} /T _{CKR}				ns, min
Clock CLK					
Minimum Pulse Width, High	T _{CH}				ns, min
Minimum Pulse Width, Low	T _{CL}				ns, min
Set/Reset					
Minimum Pulse Width, SR/BY inputs	T _{RPW}				ns, min
Delay from SR/BY inputs to XQ/YQ outputs (asynchronous)	T _{RQ}				ns, max
Toggle Frequency (MHz) (for export control)	F _{TOG}				MHz

Notes:

1. A Zero "0" Hold Time listing indicates no hold time or a negative hold time. Negative values can not be guaranteed "best-case", but if a "0" is listed, there is no positive hold time.

CLB Distributed RAM Switching Characteristics

Table 34: CLB Distributed RAM Switching Characteristics

		Speed Grade			
Description	Symbol	–8	–7	–6	Units
Sequential Delays					
Clock CLK to X/Y outputs (WE active) in 16 x 1 mode	T _{SHCKO16}				ns, max
Clock CLK to X/Y outputs (WE active) in 32 x 1 mode	T _{SHCKO32}				ns, max
Clock CLK to F5 output	T _{SHCKOF5}				ns, max
Setup and Hold Times Before/After Clock CLK					
BX/BY data inputs (DIN)	T _{DS} /T _{DH}				ns, min
F/G address inputs	T _{AS} /T _{AH}				ns, min
CE input (WE)	T _{WES} /T _{WEH}				ns, min
Clock CLK					
Minimum Pulse Width, High	T _{WPH}				ns, min
Minimum Pulse Width, Low	T _{WPL}				ns, min
Minimum clock period to meet address write cycle time	T _{WC}				ns, min

Notes:

1. A Zero "0" Hold Time listing indicates no hold time or a negative hold time. Negative values can not be guaranteed "best-case", but if a "0" is listed, there is no positive hold time.

CLB Shift Register Switching Characteristics

Table 35: CLB Shift Register Switching Characteristics

		Speed Grade			
Description	Symbol	–8	–7	–6	Units
Sequential Delays					
Clock CLK to X/Y outputs	T _{REG}				ns, max
Clock CLK to X/Y outputs	T _{REG32}				ns, max
Clock CLK to XB output via MC15 LUT output	T _{REGXB}				ns, max
Clock CLK to YB output via MC15 LUT output	T _{REGYB}				ns, max
Clock CLK to Shiftout	T _{CKSH}				ns, max
Clock CLK to F5 output	T _{REGF5}				ns, max
Setup and Hold Times Before/After Clock CLK					
BX/BY data inputs (DIN)	T _{SRLDS} /T _{SRLDH}				ns, min
CE input (WS)	T _{WSS} /T _{WSH}				ns, min
Clock CLK					
Minimum Pulse Width, High	T _{SRPH}				ns, min
Minimum Pulse Width, Low	T _{SRPL}				ns, min

Notes:

1. A Zero "0" Hold Time listing indicates no hold time or a negative hold time. Negative values can not be guaranteed "best-case", but if a "0" is listed, there is no positive hold time.

Multiplier Switching Characteristics

Table 36: Multiplier Switching Characteristics

		Speed Grade			
Description	Symbol	–8	–7	–6	Units
Propagation Delay to Output Pin					
Input to Pin35	T _{MULT_P35}				ns, max
Input to Pin34	T _{MULT_P34}				ns, max
Input to Pin33	T _{MULT_P33}				ns, max
Input to Pin32	T _{MULT_P32}				ns, max
Input to Pin31	T _{MULT_P31}				ns, max
Input to Pin30	T _{MULT_P30}				ns, max
Input to Pin29	T _{MULT_P29}				ns, max
Input to Pin28	T _{MULT_P28}				ns, max
Input to Pin27	T _{MULT_P27}				ns, max
Input to Pin26	T _{MULT_P26}				ns, max
Input to Pin25	T _{MULT_P25}				ns, max
Input to Pin24	T _{MULT_P24}				ns, max
Input to Pin23	T _{MULT_P23}				ns, max
Input to Pin22	T _{MULT_P22}				ns, max
Input to Pin21	T _{MULT_P21}				ns, max
Input to Pin20	T _{MULT_P20}				ns, max
Input to Pin19	T _{MULT_P19}				ns, max
Input to Pin18	T _{MULT_P18}				ns, max
Input to Pin17	T _{MULT_P17}				ns, max
Input to Pin16	T _{MULT_P16}				ns, max
Input to Pin15	T _{MULT_P15}				ns, max
Input to Pin14	T _{MULT_P14}				ns, max
Input to Pin13	T _{MULT_P13}				ns, max
Input to Pin12	T _{MULT_P12}				ns, max
Input to Pin11	T _{MULT_P11}				ns, max
Input to Pin10	T _{MULT_P10}				ns, max
Input to Pin9	T _{MULT_P9}				ns, max
Input to Pin8	T _{MULT_P8}				ns, max
Input to Pin7	T _{MULT_P7}				ns, max
Input to Pin6	T _{MULT_P6}				ns, max
Input to Pin5	T _{MULT_P5}				ns, max
Input to Pin4	T _{MULT_P4}				ns, max
Input to Pin3	T _{MULT_P3}				ns, max
Input to Pin2	T _{MULT_P2}				ns, max
Input to Pin1	T _{MULT_P1}				ns, max
Input to Pin0	T _{MULT_P0}				ns, max

Block SelectRAM Switching Characteristics

Table 37: Block SelectRAM Switching Characteristics

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Sequential Delays					
Clock CLK to DOUT output	T_{BCKO}				ns, max
Setup and Hold Times Before Clock CLK					
ADDR inputs	T_{BACK}/T_{BCKA}				ns, min
DIN inputs	T_{BDCK}/T_{BCKD}				ns, min
EN input	T_{BECK}/T_{BCKE}				ns, min
RST input	T_{BRCK}/T_{BCKR}				ns, min
WEN input	T_{BWCK}/T_{BCKW}				ns, min
Clock CLK					
Minimum Pulse Width, High	T_{BPWH}				ns, min
Minimum Pulse Width, Low	T_{BPWL}				ns, min

Notes:

1. A Zero "0" Hold Time listing indicates no hold time or a negative hold time. Negative values can not be guaranteed "best-case", but if a "0" is listed, there is no positive hold time.

TBUF Switching Characteristics

Table 38: TBUF Switching Characteristics

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
Combinatorial Delays					
IN input to OUT output	T_{IO}				ns, max
TRI input to OUT output high-impedance	T_{OFF}				ns, max
TRI input to valid data on OUT output	T_{ON}				ns, max

JTAG Test Access Port Switching Characteristics

Table 39: JTAG Test Access Port Switching Characteristics

		Speed Grade			
Description	Symbol	-8	-7	-6	Units
TMS and TDI Setup times before TCK	T_{TAPTK}				ns, min
TMS and TDI Hold times after TCK	T_{TCKTAP}				ns, min
Output delay from clock TCK to output TDO	T_{TCKTDO}				ns, max
Maximum TCK clock frequency	F_{TCK}				MHz, max

Virtex-II Pro Pin-to-Pin Output Parameter Guidelines

All devices are 100% functionally tested. Listed below are representative values for typical pin locations and normal clock loading. Values are expressed in nanoseconds unless otherwise noted.

Global Clock Input to Output Delay for LVCMOS25, 12 mA, Fast Slew Rate, With DCM

Table 40: Global Clock Input to Output Delay for LVCMOS25, 12 mA, Fast Slew Rate, With DCM

			Speed Grade			
Description	Symbol	Device	–8	–7	–6	Units
LVC MOS25 Global Clock Input to Output Delay using Output Flip-flop, 12 mA, Fast Slew Rate, <i>with</i> DCM. For data <i>output</i> with different standards, adjust the delays with the values shown in IOB Output Switching Characteristics Standard Adjustments , page 147 .						
Global Clock and OFF with DCM	T _{ICKOFFDCM}	XC2VP2				ns
		XC2VP4				ns
		XC2VP7				ns
		XC2VP20				ns
		XC2VP50				ns

Notes:

1. Listed above are representative values where one global clock input drives one vertical clock line in each accessible column, and where all accessible IOB and CLB flip-flops are clocked by the global clock net.
2. Output timing is measured at 50% V_{CC} threshold with 35 pF external capacitive load. For other I/O standards and different loads, see [Table 30](#).
3. DCM output jitter is already included in the timing calculation.

Global Clock Input to Output Delay for LVCMOS25, 12 mA, Fast Slew Rate, Without DCM

Table 41: Global Clock Input to Output Delay for LVCMOS25, 12 mA, Fast Slew Rate, Without DCM

			Speed Grade			
Description	Symbol	Device	–8	–7	–6	Units
LVC MOS25 Global Clock Input to Output Delay using Output Flip-flop, 12 mA, Fast Slew Rate, <i>without</i> DCM. For data <i>output</i> with different standards, adjust the delays with the values shown in IOB Output Switching Characteristics Standard Adjustments , page 147 .						
Global Clock and OFF without DCM	T _{ICKOF}	XC2VP2				ns
		XC2VP4				ns
		XC2VP7				ns
		XC2VP20				ns
		XC2VP50				ns

Notes:

1. Listed above are representative values where one global clock input drives one vertical clock line in each accessible column, and where all accessible IOB and CLB flip-flops are clocked by the global clock net.
2. Output timing is measured at 50% V_{CC} threshold with 35 pF external capacitive load. For other I/O standards and different loads, see [Table 30](#).
3. DCM output jitter is already included in the timing calculation.

Virtex-II Pro Pin-to-Pin Input Parameter Guidelines

All devices are 100% functionally tested. Listed below are representative values for typical pin locations and normal clock loading. Values are expressed in nanoseconds unless otherwise noted

Global Clock Set-Up and Hold for LVCMOS25 Standard, With DCM

Table 42: Global Clock Set-Up and Hold for LVCMOS25 Standard, With DCM

			Speed Grade			
Description	Symbol	Device	–8	–7	–6	Units
Input Setup and Hold Time Relative to Global Clock Input Signal for LVCMOS25 Standard. For data input with different standards, adjust the setup time delay by the values shown in IOB Input Switching Characteristics Standard Adjustments , page 145 .						
No Delay Global Clock and IFF with DCM	T_{PSDCM}/T_{PHDCM}	XC2VP2				ns
		XC2VP4				ns
		XC2VP7				ns
		XC2VP20				ns
		XC2VP50				ns

Notes:

1. IFF = Input Flip-Flop or Latch
2. Setup time is measured relative to the Global Clock input signal with the fastest route and the lightest load. Hold time is measured relative to the Global Clock input signal with the slowest route and heaviest load.
3. DCM output jitter is already included in the timing calculation.

Global Clock Set-Up and Hold for LVCMOS25 Standard, Without DCM

Table 43: Global Clock Set-Up and Hold for LVCMOS25 Standard, Without DCM

			Speed Grade			
Description	Symbol	Device	–8	–7	–6	Units
Input Setup and Hold Time Relative to Global Clock Input Signal for LVCMOS25 Standard. For data input with different standards, adjust the setup time delay by the values shown in IOB Input Switching Characteristics Standard Adjustments , page 145 .						
Full Delay Global Clock and IFF without DCM	T_{PSFD}/T_{PHFD}	XC2VP2				ns
		XC2VP4				ns
		XC2VP7				ns
		XC2VP20				ns
		XC2VP50				ns

Notes:

1. IFF = Input Flip-Flop or Latch
2. Setup time is measured relative to the Global Clock input signal with the fastest route and the lightest load. Hold time is measured relative to the Global Clock input signal with the slowest route and heaviest load.
3. A Zero "0" Hold Time listing indicates no hold time or a negative hold time. Negative values can not be guaranteed "best-case", but if a "0" is listed, there is no positive hold time.

DCM Timing Parameters

Testing of switching parameters is modeled after testing methods specified by MIL-M-38510/605; all devices are 100% functionally tested. Because of the difficulty in directly measuring many internal timing parameters, those parameters are derived from benchmark timing patterns. The fol-

lowing guidelines reflect worst-case values across the recommended operating conditions. All output jitter and phase specifications are determined through statistical measurement at the package pins.

Operating Frequency Ranges

Table 44: Operating Frequency Ranges

			Speed Grade			
Description	Symbol	Constraints	-8	-7	-6	Units
Output Clocks (Low Frequency Mode)						
CLK0, CLK90, CLK180, CLK270	CLKOUT_FREQ_1X_LF_MIN					MHz
	CLKOUT_FREQ_1X_LF_MAX					MHz
CLK2X, CLK2X180	CLKOUT_FREQ_2X_LF_MIN					MHz
	CLKOUT_FREQ_2X_LF_MAX					MHz
CLKDV	CLKOUT_FREQ_DV_LF_MIN					MHz
	CLKOUT_FREQ_DV_LF_MAX					MHz
CLKFX, CLKFX180	CLKOUT_FREQ_FX_LF_MIN					MHz
	CLKOUT_FREQ_FX_LF_MAX					MHz

Table 44: Operating Frequency Ranges (Continued)

			Speed Grade			Units
Description	Symbol	Constraints	-8	-7	-6	
Input Clocks (Low Frequency Mode)						
CLKIN (using DLL outputs) ⁽¹⁾	CLKIN_FREQ_DLL_LF_MIN					MHz
	CLKIN_FREQ_DLL_LF_MAX					MHz
CLKIN (using CLKFX outputs) ⁽²⁾	CLKIN_FREQ_FX_LF_MIN					MHz
	CLKIN_FREQ_FX_LF_MAX					MHz
PSCLK	PSCLK_FREQ_LF_MIN					MHz
	PSCLK_FREQ_LF_MAX					MHz
Output Clocks (High Frequency Mode)						
CLK0, CLK180	CLKOUT_FREQ_1X_HF_MIN					MHz
	CLKOUT_FREQ_1X_HF_MAX					MHz
CLKDV	CLKOUT_FREQ_DV_HF_MIN					MHz
	CLKOUT_FREQ_DV_HF_MAX					MHz
CLKFX, CLKFX180	CLKOUT_FREQ_FX_HF_MIN					MHz
	CLKOUT_FREQ_FX_HF_MAX					MHz
Input Clocks (High Frequency Mode)						
CLKIN (using DLL outputs) ⁽¹⁾	CLKIN_FREQ_DLL_HF_MIN					MHz
	CLKIN_FREQ_DLL_HF_MAX					MHz
CLKIN (using CLKFX outputs) ⁽²⁾	CLKIN_FREQ_FX_HF_MIN					MHz
	CLKIN_FREQ_FX_HF_MAX					MHz
PSCLK	PSCLK_FREQ_HF_MIN					MHz
	PSCLK_FREQ_HF_MAX					MHz

Notes:

1. "DLL outputs" is used here to describe the outputs: CLK0, CLK90, CLK180, CLK270, CLK2X, CLK2X180, and CLKDV.
2. If both DLL and CLKFX outputs are used, follow the more restrictive specification.

Input Clock Tolerances

Table 45: Input Clock Tolerances

			Speed Grade						
			–8		–7		–6		
Description	Symbol	Constraints	Min	Max	Min	Max	Min	Max	Units
Input Clock Low/high Pulse Width									
PSCLK CLKIN ⁽³⁾	PSCLK_PULSE CLKIN_PULSE	< 1MHz							ns
		1 - 10 MHz							ns
		10 - 25 MHz							ns
		25 - 50 MHz							ns
		50 - 100 MHz							ns
		100 - 150 MHz							ns
		150 - 200 MHz							ns
		200 - 250 MHz							ns
		250 - 300 MHz							ns
		300 - 350 MHz							ns
		350 - 400 MHz							ns
		> 400 MHz							ns
Input Clock Cycle-Cycle Jitter (Low Frequency Mode)									
CLKIN (using DLL outputs) ⁽¹⁾	CLKIN_CYC_JITT_DLL_LF								ps
CLKIN (using CLKFX outputs) ⁽²⁾	CLKIN_CYC_JITT_FX_LF								ps
Input Clock Cycle-Cycle Jitter (High Frequency Mode)									
CLKIN (using DLL outputs) ⁽¹⁾	CLKIN_CYC_JITT_DLL_HF								ps
CLKIN (using CLKFX outputs) ⁽²⁾	CLKIN_CYC_JITT_FX_HF								ps
Input Clock Period Jitter (Low Frequency Mode)									
CLKIN (using DLL outputs) ⁽¹⁾	CLKIN_PER_JITT_DLL_LF								ns
CLKIN (using CLKFX outputs) ⁽²⁾	CLKIN_PER_JITT_FX_LF								ns
Input Clock Period Jitter (High Frequency Mode)									
CLKIN (using DLL outputs) ⁽¹⁾	CLKIN_PER_JITT_DLL_HF								ns
CLKIN (using CLKFX outputs) ⁽²⁾	CLKIN_PER_JITT_FX_HF								ns
Feedback Clock Path Delay Variation									
CLKFB off-chip feedback	CLKFB_DELAY_VAR_EXT								ns

Notes:

1. "DLL outputs" is used here to describe the outputs: CLK0, CLK90, CLK180, CLK270, CLK2X, CLK2X180, and CLKDV.
2. If both DLL and CLKFX outputs are used, follow the more restrictive specification.
3. Specification also applies to PSCLK.

Output Clock Jitter

Table 46: Output Clock Jitter

			Speed Grade						
			–8		–7		–6		
Description	Symbol	Constraints	Min	Max	Min	Max	Min	Max	Units
Clock Synthesis Period Jitter									
CLK0	CLKOUT_PER_JITT_0								ps
CLK90	CLKOUT_PER_JITT_90								ps
CLK180	CLKOUT_PER_JITT_180								ps
CLK270	CLKOUT_PER_JITT_270								ps
CLK2X, CLK2X180	CLKOUT_PER_JITT_2X								ps
CLKDV (integer division)	CLKOUT_PER_JITT_DV1								ps
CLKDV (non-integer division)	CLKOUT_PER_JITT_DV2								ps
CLKFX, CLKFX180	CLKOUT_PER_JITT_FX								ps

Output Clock Phase Alignment

Table 47: Output Clock Phase Alignment

			Speed Grade						
			–8		–7		–6		
Description	Symbol	Constraints	Min	Max	Min	Max	Min	Max	Units
Phase Offset Between CLKIN and CLKFB									
CLKIN/CLKFB	CLKIN_CLKFB_PHASE								ps
Phase Offset Between Any DCM Outputs									
All CLK outputs	CLKOUT_PHASE								ps
Duty Cycle Precision									
DLL outputs ⁽¹⁾	CLKOUT_DUTY_CYCLE_DLL								ps
CLKFX outputs	CLKOUT_DUTY_CYCLE_FX								ps

Notes:

- “DLL outputs” is used here to describe the outputs: CLK0, CLK90, CLK180, CLK270, CLK2X, CLK2X180, and CLKDV.

Miscellaneous Timing Parameters

Table 48: Miscellaneous Timing Parameters

			Speed Grade			
Description	Symbol	Constraints F_{CLKIN}	-8	-7	-6	Units
Time Required to Achieve LOCK						
Using DLL outputs ⁽¹⁾	LOCK_DLL:					
	LOCK_DLL_60	> 60MHz				us
	LOCK_DLL_50_60	50 - 60 MHz				us
	LOCK_DLL_40_50	40 - 50 MHz				us
	LOCK_DLL_30_40	30 - 40 MHz				us
	LOCK_DLL_24_30	24 - 30 MHz				us
Using CLKFX outputs	LOCK_FX_MIN					ms
	LOCK_FX_MAX					ms
Additional lock time with fine phase shifting	LOCK_DLL_FINE_SHIFT					us
Fine Phase Shifting						
Absolute shifting range	FINE_SHIFT_RANGE					ns
Delay Lines						
Tap delay resolution	DCM_TAP_MIN					ps
	DCM_TAP_MAX					ps

Notes:

1. "DLL outputs" is used here to describe the outputs: CLK0, CLK90, CLK180, CLK270, CLK2X, CLK2X180, and CLKDV.

Frequency Synthesis

Table 49: Frequency Synthesis

Attribute	Min	Max
CLKFX_MULTIPLY	2	32
CLKFX_DIVIDE	1	32

Parameter Cross-Reference

Table 50: Parameter Cross-Reference

Libraries Guide	Data Sheet
DLL_CLKOUT_{MIN MAX}_LF	CLKOUT_FREQ_{1X 2X DV}_LF
DFS_CLKOUT_{MIN MAX}_LF	CLKOUT_FREQ_FX_LF
DLL_CLKIN_{MIN MAX}_LF	CLKIN_FREQ_DLL_LF
DFS_CLKIN_{MIN MAX}_LF	CLKIN_FREQ_FX_LF
DLL_CLKOUT_{MIN MAX}_HF	CLKOUT_FREQ_{1X DV}_HF
DFS_CLKOUT_{MIN MAX}_HF	CLKOUT_FREQ_FX_HF
DLL_CLKIN_{MIN MAX}_HF	CLKIN_FREQ_DLL_HF
DFS_CLKIN_{MIN MAX}_HF	CLKIN_FREQ_FX_HF

Revision History

This section records the change history for this module of the data sheet.

Date	Version	Revision
01/31/02	1.0	Initial Xilinx release.

Virtex-II Pro Data Sheet Modules

The Virtex-II Pro Data Sheet contains the following modules:

- **Virtex-II Pro™ Platform FPGAs: Introduction and Overview (Module 1)**
- **Virtex-II Pro™ Platform FPGAs: Functional Description (Module 2)**
- **Virtex-II Pro Platform FPGAs: DC and Switching Characteristics (Module 3)**
- **Virtex-II Pro Platform FPGAs: Pinout Information (Module 4)**

This document provides **Virtex-II Pro Device/Package Combinations and Maximum I/Os** and **Virtex-II Pro Pin Definitions**, followed by pinout tables for the following packages:

- **FG256 Fine-Pitch BGA Package**
- **FG456 Fine-Pitch BGA Package**

- **FF672 Flip-Chip Fine-Pitch BGA Package**
- **FF896 Flip-Chip Fine-Pitch BGA Package**
- **FF1152 Flip-Chip Fine-Pitch BGA Package**
- **FF1517 Flip-Chip Fine-Pitch BGA Package**
- **BF957 Flip-Chip BGA Package**

Virtex-II Pro Device/Package Combinations and Maximum I/Os

Wire-bond and flip-chip packages are available. **Table 1** and **Table 2** show the maximum number of user I/Os possible in wire-bond and flip-chip packages, respectively.

- FG denotes wire-bond fine-pitch BGA (1.00 mm pitch).
- FF denotes flip-chip fine-pitch BGA (1.00 mm pitch).
- BF denotes flip-chip BGA (1.27 mm pitch).

Table 1: Wire-Bond Packages Information

Package	FG256	FG456
Pitch (mm)	1.00	1.00
Size (mm)	17 x 17	23 x 23
I/Os	140	248

Table 2: Flip-Chip Packages Information

Package	FF672	FF896	FF1152	FF1517	BF957
Pitch (mm)	1.00	1.00	1.00	1.00	1.27
Size (mm)	27 x 27	31 x 31	35 x 35	40 x 40	40 x 40
I/Os	396	556	692	852	584

Table 3 shows the number of available I/Os and the number of Rocket I/O™ multi-gigabit transceiver (MGT) pins for each Virtex-II Pro device/package combination. The number of I/Os per package includes all user I/Os *except* the fifteen control pins (CCLK, DONE, M0, M1, M2, PROG_B, PWRDWN_B, TCK, TDI, TDO, TMS, HSWAP_EN, DXN, DXP, AND RSVD) and the nine (per transceiver) Rocket I/O MGT pins (TXP, TXN, RXP, RXN, AVCCAUXTX, AVCCAUXRX, VTTX, VTRX, and GNDA). The number of transceivers in the device is the number of Rocket I/O MGT pins in **Table 3** divided by nine.

Table 3: Virtex-II Pro Available I/Os and Rocket I/O MGT Pins per Device/Package Combination

Device⇒ Package⇓	XC2VP2		XC2VP4		XC2VP7		XC2VP20		XC2VP50	
	Available User I/Os	Rocket I/O MGT Pins	Available User I/Os	Rocket I/O MGT Pins	Available User I/Os	Rocket I/O MGT Pins	Available User I/Os	Rocket I/O MGT Pins	Available User I/Os	Rocket I/O MGT Pins
FG256	140	36	140	36						
FG456	156	72	248	72	248	72				
FF672	204	72	348	72	396	72				
FF896					396	72	556	72		
FF1152							564	144	692	144
FF1517									852	144
BF957							564	108	584	108

Table 4 shows the number of 3.3V SelectI/Os in each bank and the total for each device/package combination.

Table 4: 3.3V SelectI/O Banks

Virtex-II Pro Device	Package	Bank0	Bank1	Bank2	Bank3	Bank4	Bank5	Bank6	Bank7	Total 3.3V I/Os
2VP2	FG256		17	18	18					53
	FG456		21	18	18					57
	FF672		27	24	24					75
2VP4	FG256		17	17	17					51
	FG456		21	40	42					103
	FF672		27	60	60					147
2VP7	FG456		21	40	42					103
	FF672		39	60	60					159
	FF896		39	60	60					159
2VP20	BF957	57	57							114
	FF896		55	84						139
	FF1152	57	57							114
2VP50	BF957	59	59							118
	FF1152	69	69							138
	FF1517	81	81							162

Virtex-II Pro Pin Definitions

This section describes the pinouts for Virtex-II Pro devices in the following packages:

- FG256 and FG456: wire-bond fine-pitch BGA of 1.00 mm pitch
- FF672, FF896, FF1152, and FF1517: flip-chip fine-pitch BGA of 1.00 mm pitch
- BF957: flip-chip BGA of 1.27 mm pitch

All of the devices supported in a particular package are pinout compatible and are listed in the same table (one

table per package). Pins that are not available for the smallest devices are listed in right-hand columns.

Each device is split into eight I/O banks to allow for flexibility in the choice of I/O standards (see the Virtex-II Pro *Data Sheet*). Global pins, including JTAG, configuration, and power/ground pins, are listed at the end of each table. Table 5 provides definitions for all pin types.

All Virtex-II Pro pinout tables are available on the distribution CD-ROM, or on the web (at <http://www.xilinx.com>).

Pin Definitions

Table 5 provides a description of each pin type listed in Virtex-II Pro pinout tables.

Table 5: Virtex-II Pro Pin Definitions

Pin Name	Direction	Description
User I/O Pins		
IO_LXXY_#	Input/Output	All user I/O pins are capable of differential signalling and can implement LVDS, ULVDS, BLVDS, or LDT pairs. Each user I/O is labeled "IO_LXXY_#", where: IO indicates a user I/O pin. LXXY indicates a differential pair, with XX a unique pair in the bank and Y = P/N for the positive and negative sides of the differential pair. # indicates the bank number (0 through 7)
Dual-Function Pins		
IO_LXXY_#/ZZZ		The dual-function pins are labelled "IO_LXXY_#/ZZZ", where ZZZ can be one of the following pins: Per Bank - VRP , VRN , or VREF Globally - GCLKX(S/P) , BUSY/DOUT , INIT_B , DIN/D0 – D7 , RDWR_B , or CS_B
With /ZZZ:		
DIN / D0, D1, D2, D3, D4, D5, D6, D7	Input/Output	In SelectMAP mode, D0 through D7 are configuration data pins. These pins become user I/Os after configuration, unless the SelectMAP port is retained. In bit-serial modes, DIN (D0) is the single-data input. This pin becomes a user I/O after configuration.
CS_B	Input	In SelectMAP mode, this is the active-low Chip Select signal. The pin becomes a user I/O after configuration, unless the SelectMAP port is retained.
RDWR_B	Input	In SelectMAP mode, this is the active-low Write Enable signal. The pin becomes a user I/O after configuration, unless the SelectMAP port is retained.
BUSY/DOUT	Output	In SelectMAP mode, BUSY controls the rate at which configuration data is loaded. The pin becomes a user I/O after configuration, unless the SelectMAP port is retained. In bit-serial modes, DOUT provides preamble and configuration data to downstream devices in a daisy-chain. The pin becomes a user I/O after configuration.
INIT_B	Bidirectional (open-drain)	When Low, this pin indicates that the configuration memory is being cleared. When held Low, the start of configuration is delayed. During configuration, a Low on this output indicates that a configuration data error has occurred. The pin becomes a user I/O after configuration.
GCLKx (S/P)	Input	These are clock input pins that connect to Global Clock Buffers. These pins become regular user I/Os when not needed for clocks.
VRP	Input	This pin is for the DCI voltage reference resistor of P transistor (per bank).
VRN	Input	This pin is for the DCI voltage reference resistor of N transistor (per bank).
ALT_VRP	Input	This is the alternative pin for the DCI voltage reference resistor of P transistor.
ALT_VRN	Input	This is the alternative pin for the DCI voltage reference resistor of N transistor.
V _{REF}	Input	These are input threshold voltage pins. They become user I/Os when an external threshold voltage is not needed (per bank).
Dedicated Pins⁽¹⁾		
CCLK	Input/Output	Configuration clock. Output in Master mode or Input in Slave mode.

Table 5: Virtex-II Pro Pin Definitions (Continued)

Pin Name	Direction	Description
PROG_B	Input	Active Low asynchronous reset to configuration logic. This pin has a permanent weak pull-up resistor.
DONE	Input/Output	DONE is a bidirectional signal with an optional internal pull-up resistor. As an output, this pin indicates completion of the configuration process. As an input, a Low level on DONE can be configured to delay the start-up sequence.
M2, M1, M0	Input	Configuration mode selection.
HSWAP_EN	Input	Enable I/O pullups during configuration.
TCK	Input	Boundary Scan Clock.
TDI	Input	Boundary Scan Data Input.
TDO	Output	Boundary Scan Data Output.
TMS	Input	Boundary Scan Mode Select.
PWRDWN_B	Input	Power down pin.
Other Pins		
DXN, DXP	N/A	Temperature-sensing diode pins (Anode: DXP, Cathode: DXN).
V _{BATT}	Input	Decryptor key memory backup supply. (Do not connect if battery is not used.)
RSVD	N/A	Reserved pin - do not connect.
V _{CCO}	Input	Power-supply pins for the output drivers (per bank).
V _{CCAUX}	Input	Power-supply pins for auxiliary circuits.
V _{CCINT}	Input	Power-supply pins for the internal core logic.
GND	Input	Ground.
AVCCAUXRX#	Input	Analog power supply for receive circuitry of the Rocket I/O multi-gigabit transceiver (2.5V).
AVCCAUTX#	Input	Analog power supply for transmit circuitry of the Rocket I/O multi-gigabit transceiver (2.5V).
VTRXPAD#	Input	Receive termination supply for the Rocket I/O multi-gigabit transceiver (1.8V - 2.8V).
VTTXPAD#	Input	Transmit termination supply for the Rocket I/O multi-gigabit transceiver (1.8V - 2.8V).
GNDA# ⁽²⁾	Input	Ground for the analog circuitry of the Rocket I/O multi-gigabit transceiver.
RXPPAD#	Output	Positive differential receive port of the Rocket I/O multi-gigabit transceiver.
RXNPAD#	Output	Negative differential receive port of the Rocket I/O multi-gigabit transceiver.
TXPPAD#	Input	Positive differential transmit port of the Rocket I/O multi-gigabit transceiver.
TXNPAD#	Input	Negative differential transmit port of the Rocket I/O multi-gigabit transceiver.

Notes:

1. All dedicated pins (JTAG and configuration) are powered by V_{CCAUX} (independent of the bank V_{CCO} voltage).
2. Two pads on the die are tied to the same package pin (GNDA) in order to lower the resistance on these connections. Thus, duplicate entries exist for GNDA pins.

FG256 Fine-Pitch BGA Package

As shown in Table 6, XC2VP2 and XC2VP4 Virtex-II Pro devices are available in the FG256 fine-pitch BGA package. The pins in each of these devices are identical. Following this table are the **FG256 Fine-Pitch BGA Package Specifications (1.00mm pitch)**.

Table 6: FG256 — XC2VP2 and XC2VP4

Bank	Pin Description	Pin Number
0	IO_L01N_0/VRP_0	C2
0	IO_L01P_0/VRN_0	C3
0	IO_L02N_0	B3
0	IO_L02P_0	C4
0	IO_L03N_0	A2
0	IO_L03P_0/VREF_0	A3
0	IO_L06N_0	D5
0	IO_L06P_0	C5
0	IO_L07P_0	D6
0	IO_L09N_0	E6
0	IO_L09P_0/VREF_0	E7
0	IO_L69N_0	D7
0	IO_L69P_0/VREF_0	C7
0	IO_L74N_0/GCLK7P	D8
0	IO_L74P_0/GCLK6S	C8
0	IO_L75N_0/GCLK5P	B8
0	IO_L75P_0/GCLK4S	A8
1	IO_L75N_1/GCLK3P	A9
1	IO_L75P_1/GCLK2S	B9
1	IO_L74N_1/GCLK1P	C9
1	IO_L74P_1/GCLK0S	D9
1	IO_L69N_1/VREF_1	C10
1	IO_L69P_1	D10
1	IO_L09N_1/VREF_1	E10
1	IO_L09P_1	E11
1	IO_L07N_1	D11
1	IO_L06N_1	C12
1	IO_L06P_1	D12
1	IO_L03N_1/VREF_1	A14
1	IO_L03P_1	A15

Table 6: FG256 — XC2VP2 and XC2VP4

Bank	Pin Description	Pin Number
1	IO_L02N_1	C13
1	IO_L02P_1	B14
1	IO_L01N_1/VRP_1	C14
1	IO_L01P_1/VRN_1	C15
2	IO_L01N_2/VRP_2	E14
2	IO_L01P_2/VRN_2	E15
2	IO_L02N_2	E13
2	IO_L02P_2	F12
2	IO_L03N_2	F13
2	IO_L03P_2	F14
2	IO_L04N_2/VREF_2	F15
2	IO_L04P_2	F16
2	IO_L06N_2	G13
2	IO_L06P_2	G14
2	IO_L85N_2	G15
2	IO_L85P_2	G16
2	IO_L86N_2	G12
2	IO_L86P_2	H13
2	IO_L88N_2/VREF_2	H14
2	IO_L88P_2	H15
2	IO_L90N_2	H16
2	IO_L90P_2	J16
3	IO_L90N_3	J15
3	IO_L90P_3	J14
3	IO_L89N_3	J13
3	IO_L89P_3	K12
3	IO_L87N_3/VREF_3	K16
3	IO_L87P_3	K15
3	IO_L85N_3	K14
3	IO_L85P_3	K13
3	IO_L06N_3	L16
3	IO_L06P_3	L15
3	IO_L05N_3	L14

Table 6: FG256 — XC2VP2 and XC2VP4

Bank	Pin Description	Pin Number
3	IO_L05P_3	L13
3	IO_L03N_3/VREF_3	L12
3	IO_L03P_3	M13
3	IO_L02N_3	M16
3	IO_L02P_3	N16
3	IO_L01N_3/VRP_3	M15
3	IO_L01P_3/VRN_3	M14
4	IO_L01N_4/DOUT	P15
4	IO_L01P_4/INIT_B	P14
4	IO_L02N_4/D0	R14
4	IO_L02P_4/D1	P13
4	IO_L03N_4/D2	T15
4	IO_L03P_4/D3	T14
4	IO_L06N_4/VRP_4	N12
4	IO_L06P_4/VRN_4	P12
4	IO_L07P_4/VREF_4	N11
4	IO_L09N_4	M11
4	IO_L09P_4/VREF_4	M10
4	IO_L69N_4	N10
4	IO_L69P_4/VREF_4	P10
4	IO_L74N_4/GCLK3S	N9
4	IO_L74P_4/GCLK2P	P9
4	IO_L75N_4/GCLK1S	R9
4	IO_L75P_4/GCLK0P	T9
5	IO_L75N_5/GCLK7S	T8
5	IO_L75P_5/GCLK6P	R8
5	IO_L74N_5/GCLK5S	P8
5	IO_L74P_5/GCLK4P	N8
5	IO_L69N_5/VREF_5	P7
5	IO_L69P_5	N7
5	IO_L09N_5/VREF_5	M7
5	IO_L09P_5	M6
5	IO_L07N_5/VREF_5	N6

Table 6: FG256 — XC2VP2 and XC2VP4

Bank	Pin Description	Pin Number
5	IO_L06N_5/VRP_5	P5
5	IO_L06P_5/VRN_5	N5
5	IO_L03N_5/D4	T3
5	IO_L03P_5/D5	T2
5	IO_L02N_5/D6	P4
5	IO_L02P_5/D7	R3
5	IO_L01N_5/RDWR_B	P3
5	IO_L01P_5/CS_B	P2
6	IO_L01P_6/VRN_6	M3
6	IO_L01N_6/VRP_6	M2
6	IO_L02P_6	N1
6	IO_L02N_6	M1
6	IO_L03P_6	M4
6	IO_L03N_6/VREF_6	L5
6	IO_L05P_6	L4
6	IO_L05N_6	L3
6	IO_L06P_6	L2
6	IO_L06N_6	L1
6	IO_L85P_6	K4
6	IO_L85N_6	K3
6	IO_L87P_6	K2
6	IO_L87N_6/VREF_6	K1
6	IO_L89P_6	K5
6	IO_L89N_6	J4
6	IO_L90P_6	J3
6	IO_L90N_6	J2
7	IO_L90P_7	J1
7	IO_L90N_7	H1
7	IO_L88P_7	H2
7	IO_L88N_7/VREF_7	H3
7	IO_L86P_7	H4
7	IO_L86N_7	G5
7	IO_L85P_7	G1

Table 6: FG256 — XC2VP2 and XC2VP4

Bank	Pin Description	Pin Number
7	IO_L85N_7	G2
7	IO_L06P_7	G3
7	IO_L06N_7	G4
7	IO_L04P_7	F1
7	IO_L04N_7/VREF_7	F2
7	IO_L03P_7	F3
7	IO_L03N_7	F4
7	IO_L02P_7	F5
7	IO_L02N_7	E4
7	IO_L01P_7/VRN_7	E2
7	IO_L01N_7/VRP_7	E3
0	VCCO_0	F8
0	VCCO_0	F7
0	VCCO_0	E8
1	VCCO_1	F9
1	VCCO_1	F10
1	VCCO_1	E9
2	VCCO_2	H12
2	VCCO_2	H11
2	VCCO_2	G11
3	VCCO_3	K11
3	VCCO_3	J12
3	VCCO_3	J11
4	VCCO_4	M9
4	VCCO_4	L9
4	VCCO_4	L10
5	VCCO_5	M8
5	VCCO_5	L8
5	VCCO_5	L7
6	VCCO_6	K6
6	VCCO_6	J6
6	VCCO_6	J5
7	VCCO_7	H6
7	VCCO_7	H5

Table 6: FG256 — XC2VP2 and XC2VP4

Bank	Pin Description	Pin Number
7	VCCO_7	G6
N/A	CCLK	N15
N/A	PROG_B	D1
N/A	DONE	P16
N/A	M0	N3
N/A	M1	N2
N/A	M2	P1
N/A	TCK	D16
N/A	TDI	E1
N/A	TDO	E16
N/A	TMS	C16
N/A	PWRDWN_B	N14
N/A	HSWAP_EN	C1
N/A	RSVD	D14
N/A	VBATT	D15
N/A	DXP	D2
N/A	DXN	D3
N/A	AVCCAUXTX6	B5
N/A	VTTXPAD6	B4
N/A	TXNPAD6	A4
N/A	TXPPAD6	A5
N/A	GND A6	C6
N/A	GND A6	C6
N/A	RXPPAD6	A6
N/A	RXNPAD6	A7
N/A	VTRXPAD6	B6
N/A	AVCCAUXRX6	B7
N/A	AVCCAUXTX7	B11
N/A	VTTXPAD7	B10
N/A	TXNPAD7	A10
N/A	TXPPAD7	A11
N/A	GND A7	C11
N/A	GND A7	C11
N/A	RXPPAD7	A12

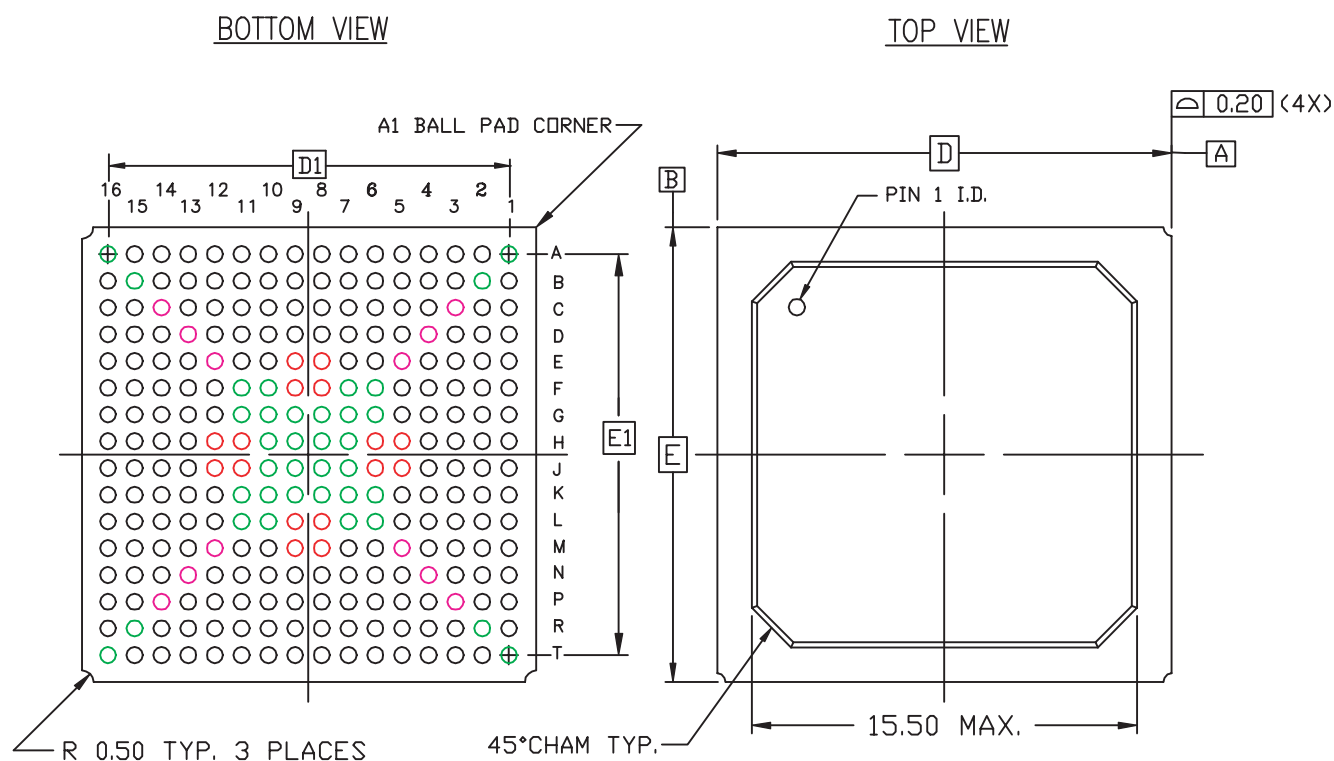
Table 6: FG256 — XC2VP2 and XC2VP4

Bank	Pin Description	Pin Number
N/A	RXNPAD7	A13
N/A	VTRXPAD7	B12
N/A	AVCCAUXRX7	B13
N/A	AVCCAUXRX18	R13
N/A	VTRXPAD18	R12
N/A	RXNPAD18	T13
N/A	RXPPAD18	T12
N/A	GND A18	P11
N/A	GND A18	P11
N/A	TXPPAD18	T11
N/A	TXNPAD18	T10
N/A	VTTXPAD18	R10
N/A	AVCCAUXTX18	R11
N/A	AVCCAUXRX19	R7
N/A	VTRXPAD19	R6
N/A	RXNPAD19	T7
N/A	RXPPAD19	T6
N/A	GND A19	P6
N/A	GND A19	P6
N/A	TXPPAD19	T5
N/A	TXNPAD19	T4
N/A	VTTXPAD19	R4
N/A	AVCCAUXTX19	R5
N/A	VCCINT	N4
N/A	VCCINT	N13
N/A	VCCINT	M5
N/A	VCCINT	M12
N/A	VCCINT	E5
N/A	VCCINT	E12
N/A	VCCINT	D4
N/A	VCCINT	D13
N/A	VCCAUX	R16
N/A	VCCAUX	R1
N/A	VCCAUX	B16

Table 6: FG256 — XC2VP2 and XC2VP4

Bank	Pin Description	Pin Number
N/A	VCCAUX	B1
N/A	GND	T16
N/A	GND	T1
N/A	GND	R2
N/A	GND	R15
N/A	GND	L6
N/A	GND	L11
N/A	GND	K9
N/A	GND	K8
N/A	GND	K7
N/A	GND	K10
N/A	GND	J9
N/A	GND	J8
N/A	GND	J7
N/A	GND	J10
N/A	GND	H9
N/A	GND	H8
N/A	GND	H7
N/A	GND	H10
N/A	GND	G9
N/A	GND	G8
N/A	GND	G7
N/A	GND	G10
N/A	GND	F6
N/A	GND	F11
N/A	GND	B2
N/A	GND	B15
N/A	GND	A16
N/A	GND	A1

FG256 Fine-Pitch BGA Package Specifications (1.00mm pitch)



SYMBOL	MILLIMETERS		
	MIN.	NOM.	MAX.
A	$\sim$	1.73	2.00
A ₁	0.40	0.50	0.60
D/E	17.00 BSC		
D ₁ /E ₁	15.00 REF		
e	1.00 BSC		
øb	0.50	0.60	0.70
aaa	$\sim$	$\sim$	0.20
ccc	$\sim$	$\sim$	0.35
ddd	$\sim$	$\sim$	0.30
eee	$\sim$	$\sim$	0.10
M	16		

NOTES:

1. ALL DIMENSIONS AND TOLERANCES CONFORM TO ANSI Y14.5M-1994
2. SYMBOL 'M' IS THE BALL MATRIX SIZE.
3. CONFORMS TO JEDEC MO-151 AAF-1.

Figure 1: FG256 Fine-Pitch BGA Package Specifications

FG456 Fine-Pitch BGA Package

As shown in [Table 7](#), XC2VP2, XC2VP4, and XC2VP7 Virtex-II Pro devices are available in the FG456 fine-pitch BGA package. The pins in these devices are same, except for the differences shown in the "No Connects" column. Following this table are the **FG456 Fine-Pitch BGA Package Specifications (1.00mm pitch)**.

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
0	IO_L01N_0/VRP_0	D5			
0	IO_L01P_0/VRN_0	D6			
0	IO_L02N_0	E6			
0	IO_L02P_0	E7			
0	IO_L03N_0	D7			
0	IO_L03P_0/VREF_0	C7			
0	IO_L05_0/No_Pair	E8			
0	IO_L06N_0	D8			
0	IO_L06P_0	C8			
0	IO_L07N_0	F9			
0	IO_L07P_0	E9			
0	IO_L09N_0	D9			
0	IO_L09P_0/VREF_0	D10			
0	IO_L67N_0	F10			
0	IO_L67P_0	E10			
0	IO_L69N_0	C10			
0	IO_L69P_0/VREF_0	B11			
0	IO_L74N_0/GCLK7P	F11			
0	IO_L74P_0/GCLK6S	E11			
0	IO_L75N_0/GCLK5P	D11			
0	IO_L75P_0/GCLK4S	C11			
1	IO_L75N_1/GCLK3P	C12			
1	IO_L75P_1/GCLK2S	D12			
1	IO_L74N_1/GCLK1P	E12			
1	IO_L74P_1/GCLK0S	F12			
1	IO_L69N_1/VREF_1	B12			
1	IO_L69P_1	C13			
1	IO_L67N_1	E13			
1	IO_L67P_1	F13			
1	IO_L09N_1/VREF_1	D13			

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
1	IO_L09P_1	D14			
1	IO_L07N_1	E14			
1	IO_L07P_1	F14			
1	IO_L06N_1	C15			
1	IO_L06P_1	D15			
1	IO_L05_1/No_Pair	E15			
1	IO_L03N_1/VREF_1	C16			
1	IO_L03P_1	D16			
1	IO_L02N_1	E16			
1	IO_L02P_1	E17			
1	IO_L01N_1/VRP_1	D17			
1	IO_L01P_1/VRN_1	D18			
2	IO_L01N_2/VRP_2	C21			
2	IO_L01P_2/VRN_2	C22			
2	IO_L02N_2	D21			
2	IO_L02P_2	D22			
2	IO_L03N_2	E19			
2	IO_L03P_2	E20			
2	IO_L04N_2/VREF_2	E21			
2	IO_L04P_2	E22			
2	IO_L06N_2	F19			
2	IO_L06P_2	F20			
2	IO_L43N_2	F21	NC		
2	IO_L43P_2	F22	NC		
2	IO_L46N_2/VREF_2	F18	NC		
2	IO_L46P_2	G18	NC		
2	IO_L48N_2	G19	NC		
2	IO_L48P_2	G20	NC		
2	IO_L49N_2	G21	NC		
2	IO_L49P_2	G22	NC		
2	IO_L50N_2	H19	NC		
2	IO_L50P_2	H20	NC		
2	IO_L52N_2/VREF_2	H21	NC		
2	IO_L52P_2	H22	NC		

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
2	IO_L54N_2	H18	NC		
2	IO_L54P_2	J17	NC		
2	IO_L55N_2	J19	NC		
2	IO_L55P_2	J20	NC		
2	IO_L56N_2	J21	NC		
2	IO_L56P_2	J22	NC		
2	IO_L58N_2/VREF_2	J18	NC		
2	IO_L58P_2	K18	NC		
2	IO_L60N_2	K19	NC		
2	IO_L60P_2	K20	NC		
2	IO_L85N_2	K21			
2	IO_L85P_2	K22			
2	IO_L86N_2	K17			
2	IO_L86P_2	L17			
2	IO_L88N_2/VREF_2	L18			
2	IO_L88P_2	L19			
2	IO_L90N_2	L20			
2	IO_L90P_2	L21			
3	IO_L90N_3	M21			
3	IO_L90P_3	M20			
3	IO_L89N_3	M19			
3	IO_L89P_3	M18			
3	IO_L87N_3/VREF_3	M17			
3	IO_L87P_3	N17			
3	IO_L85N_3	N22			
3	IO_L85P_3	N21			
3	IO_L60N_3	N20	NC		
3	IO_L60P_3	N19	NC		
3	IO_L59N_3	N18	NC		
3	IO_L59P_3	P18	NC		
3	IO_L57N_3/VREF_3	P22	NC		
3	IO_L57P_3	P21	NC		
3	IO_L55N_3	P20	NC		
3	IO_L55P_3	P19	NC		

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
3	IO_L54N_3	P17	NC		
3	IO_L54P_3	R18	NC		
3	IO_L53N_3	R22	NC		
3	IO_L53P_3	R21	NC		
3	IO_L51N_3/VREF_3	R20	NC		
3	IO_L51P_3	R19	NC		
3	IO_L49N_3	T22	NC		
3	IO_L49P_3	T21	NC		
3	IO_L48N_3	T20	NC		
3	IO_L48P_3	T19	NC		
3	IO_L47N_3	T18	NC		
3	IO_L47P_3	U18	NC		
3	IO_L45N_3/VREF_3	U22	NC		
3	IO_L45P_3	U21	NC		
3	IO_L43N_3	U20	NC		
3	IO_L43P_3	U19	NC		
3	IO_L06N_3	V22			
3	IO_L06P_3	V21			
3	IO_L05N_3	V20			
3	IO_L05P_3	V19			
3	IO_L03N_3/VREF_3	W22			
3	IO_L03P_3	W21			
3	IO_L02N_3	Y22			
3	IO_L02P_3	Y21			
3	IO_L01N_3/VRP_3	AA22			
3	IO_L01P_3/VRN_3	AB21			
4	IO_L01N_4/DOUT	W18			
4	IO_L01P_4/INIT_B	W17			
4	IO_L02N_4/D0	V17			
4	IO_L02P_4/D1	V16			
4	IO_L03N_4/D2	W16			
4	IO_L03P_4/D3	Y16			
4	IO_L05_4/No_Pair	V15			
4	IO_L06N_4/VRP_4	W15			

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
4	IO_L06P_4/VRN_4	Y15			
4	IO_L07N_4	U14			
4	IO_L07P_4/VREF_4	V14			
4	IO_L09N_4	W14			
4	IO_L09P_4/VREF_4	W13			
4	IO_L67N_4	U13			
4	IO_L67P_4	V13			
4	IO_L69N_4	Y13			
4	IO_L69P_4/VREF_4	AA12			
4	IO_L74N_4/GCLK3S	U12			
4	IO_L74P_4/GCLK2P	V12			
4	IO_L75N_4/GCLK1S	W12			
4	IO_L75P_4/GCLK0P	Y12			
5	IO_L75N_5/GCLK7S	Y11			
5	IO_L75P_5/GCLK6P	W11			
5	IO_L74N_5/GCLK5S	V11			
5	IO_L74P_5/GCLK4P	U11			
5	IO_L69N_5/VREF_5	AA11			
5	IO_L69P_5	Y10			
5	IO_L67N_5	V10			
5	IO_L67P_5	U10			
5	IO_L09N_5/VREF_5	W10			
5	IO_L09P_5	W9			
5	IO_L07N_5/VREF_5	V9			
5	IO_L07P_5	U9			
5	IO_L06N_5/VRP_5	Y8			
5	IO_L06P_5/VRN_5	W8			
5	IO_L05_5/No_Pair	V8			
5	IO_L03N_5/D4	Y7			
5	IO_L03P_5/D5	W7			
5	IO_L02N_5/D6	V7			
5	IO_L02P_5/D7	V6			
5	IO_L01N_5/RDWR_B	W6			
5	IO_L01P_5/CS_B	W5			

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
6	IO_L01P_6/VRN_6	AB2			
6	IO_L01N_6/VRP_6	AA1			
6	IO_L02P_6	Y2			
6	IO_L02N_6	Y1			
6	IO_L03P_6	W2			
6	IO_L03N_6/VREF_6	W1			
6	IO_L05P_6	V4			
6	IO_L05N_6	V3			
6	IO_L06P_6	V2			
6	IO_L06N_6	V1			
6	IO_L43P_6	U4	NC		
6	IO_L43N_6	U3	NC		
6	IO_L45P_6	U2	NC		
6	IO_L45N_6/VREF_6	U1	NC		
6	IO_L47P_6	U5	NC		
6	IO_L47N_6	T5	NC		
6	IO_L48P_6	T4	NC		
6	IO_L48N_6	T3	NC		
6	IO_L49P_6	T2	NC		
6	IO_L49N_6	T1	NC		
6	IO_L51P_6	R4	NC		
6	IO_L51N_6/VREF_6	R3	NC		
6	IO_L53P_6	R2	NC		
6	IO_L53N_6	R1	NC		
6	IO_L54P_6	R5	NC		
6	IO_L54N_6	P6	NC		
6	IO_L55P_6	P4	NC		
6	IO_L55N_6	P3	NC		
6	IO_L57P_6	P2	NC		
6	IO_L57N_6/VREF_6	P1	NC		
6	IO_L59P_6	P5	NC		
6	IO_L59N_6	N5	NC		
6	IO_L60P_6	N4	NC		
6	IO_L60N_6	N3	NC		

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
6	IO_L85P_6	N2			
6	IO_L85N_6	N1			
6	IO_L87P_6	N6			
6	IO_L87N_6/VREF_6	M6			
6	IO_L89P_6	M5			
6	IO_L89N_6	M4			
6	IO_L90P_6	M3			
6	IO_L90N_6	M2			
7	IO_L90P_7	L2			
7	IO_L90N_7	L3			
7	IO_L88P_7	L4			
7	IO_L88N_7/VREF_7	L5			
7	IO_L86P_7	L6			
7	IO_L86N_7	K6			
7	IO_L85P_7	K1			
7	IO_L85N_7	K2			
7	IO_L60P_7	K3	NC		
7	IO_L60N_7	K4	NC		
7	IO_L58P_7	K5	NC		
7	IO_L58N_7/VREF_7	J5	NC		
7	IO_L56P_7	J1	NC		
7	IO_L56N_7	J2	NC		
7	IO_L55P_7	J3	NC		
7	IO_L55N_7	J4	NC		
7	IO_L54P_7	J6	NC		
7	IO_L54N_7	H5	NC		
7	IO_L52P_7	H1	NC		
7	IO_L52N_7/VREF_7	H2	NC		
7	IO_L50P_7	H3	NC		
7	IO_L50N_7	H4	NC		
7	IO_L49P_7	G1	NC		
7	IO_L49N_7	G2	NC		
7	IO_L48P_7	G3	NC		
7	IO_L48N_7	G4	NC		

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
7	IO_L46P_7	G5	NC		
7	IO_L46N_7/VREF_7	F5	NC		
7	IO_L43P_7	F1	NC		
7	IO_L43N_7	F2	NC		
7	IO_L06P_7	F3			
7	IO_L06N_7	F4			
7	IO_L04P_7	E1			
7	IO_L04N_7/VREF_7	E2			
7	IO_L03P_7	E3			
7	IO_L03N_7	E4			
7	IO_L02P_7	D1			
7	IO_L02N_7	D2			
7	IO_L01P_7/VRN_7	C1			
7	IO_L01N_7/VRP_7	C2			
0	VCCO_0	G9			
0	VCCO_0	G11			
0	VCCO_0	G10			
0	VCCO_0	F8			
0	VCCO_0	F7			
1	VCCO_1	G14			
1	VCCO_1	G13			
1	VCCO_1	G12			
1	VCCO_1	F16			
1	VCCO_1	F15			
2	VCCO_2	L16			
2	VCCO_2	K16			
2	VCCO_2	J16			
2	VCCO_2	H17			
2	VCCO_2	G17			
3	VCCO_3	T17			
3	VCCO_3	R17			
3	VCCO_3	P16			
3	VCCO_3	N16			
3	VCCO_3	M16			

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
4	VCCO_4	U16			
4	VCCO_4	U15			
4	VCCO_4	T14			
4	VCCO_4	T13			
4	VCCO_4	T12			
5	VCCO_5	U8			
5	VCCO_5	U7			
5	VCCO_5	T9			
5	VCCO_5	T11			
5	VCCO_5	T10			
6	VCCO_6	T6			
6	VCCO_6	R6			
6	VCCO_6	P7			
6	VCCO_6	N7			
6	VCCO_6	M7			
7	VCCO_7	L7			
7	VCCO_7	K7			
7	VCCO_7	J7			
7	VCCO_7	H6			
7	VCCO_7	G6			
N/A	CCLK	W20			
N/A	PROG_B	B1			
N/A	DONE	Y18			
N/A	M0	Y4			
N/A	M1	W3			
N/A	M2	Y5			
N/A	TCK	B22			
N/A	TDI	D3			
N/A	TDO	D20			
N/A	TMS	A21			
N/A	PWRDWN_B	Y19			
N/A	HSWAP_EN	A2			
N/A	RSVD	C18			
N/A	VBATT	C19			

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
N/A	DXP	C4			
N/A	DXN	C5			
N/A	AVCCAUXTX4	B4	NC	NC	
N/A	VTTXPAD4	B3	NC	NC	
N/A	TXNPAD4	A3	NC	NC	
N/A	TXPPAD4	A4	NC	NC	
N/A	GND4	C6	NC	NC	
N/A	GND4	C6	NC	NC	
N/A	RXPPAD4	A5	NC	NC	
N/A	RXNPAD4	A6	NC	NC	
N/A	VTRXPAD4	B5	NC	NC	
N/A	AVCCAUXRX4	B6	NC	NC	
N/A	AVCCAUXTX6	B8			
N/A	VTTXPAD6	B7			
N/A	TXNPAD6	A7			
N/A	TXPPAD6	A8			
N/A	GND6	C9			
N/A	GND6	C9			
N/A	RXPPAD6	A9			
N/A	RXNPAD6	A10			
N/A	VTRXPAD6	B9			
N/A	AVCCAUXRX6	B10			
N/A	AVCCAUXTX7	B14			
N/A	VTTXPAD7	B13			
N/A	TXNPAD7	A13			
N/A	TXPPAD7	A14			
N/A	GND7	C14			
N/A	GND7	C14			
N/A	RXPPAD7	A15			
N/A	RXNPAD7	A16			
N/A	VTRXPAD7	B15			
N/A	AVCCAUXRX7	B16			
N/A	AVCCAUXTX9	B18	NC	NC	
N/A	VTTXPAD9	B17	NC	NC	
N/A	TXNPAD9	A17	NC	NC	

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
N/A	TXPPAD9	A18	NC	NC	
N/A	GND A9	C17	NC	NC	
N/A	GND A9	C17	NC	NC	
N/A	RXPPAD9	A19	NC	NC	
N/A	RXNPAD9	A20	NC	NC	
N/A	VTRXPAD9	B19	NC	NC	
N/A	AVCCAUXRX9	B20	NC	NC	
N/A	AVCCAUXRX16	AA20	NC	NC	
N/A	VTRXPAD16	AA19	NC	NC	
N/A	RXNPAD16	AB20	NC	NC	
N/A	RXPPAD16	AB19	NC	NC	
N/A	GND A16	Y17	NC	NC	
N/A	GND A16	Y17	NC	NC	
N/A	TXPPAD16	AB18	NC	NC	
N/A	TXNPAD16	AB17	NC	NC	
N/A	VTTXPAD16	AA17	NC	NC	
N/A	AVCCAUXTX16	AA18	NC	NC	
N/A	AVCCAUXRX18	AA16			
N/A	VTRXPAD18	AA15			
N/A	RXNPAD18	AB16			
N/A	RXPPAD18	AB15			
N/A	GND A18	Y14			
N/A	GND A18	Y14			
N/A	TXPPAD18	AB14			
N/A	TXNPAD18	AB13			
N/A	VTTXPAD18	AA13			
N/A	AVCCAUXTX18	AA14			
N/A	AVCCAUXRX19	AA10			
N/A	VTRXPAD19	AA9			
N/A	RXNPAD19	AB10			
N/A	RXPPAD19	AB9			
N/A	GND A19	Y9			
N/A	GND A19	Y9			
N/A	TXPPAD19	AB8			
N/A	TXNPAD19	AB7			

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
N/A	VTTXPAD19	AA7			
N/A	AVCCAUXTX19	AA8			
N/A	AVCCAUXRX21	AA6	NC	NC	
N/A	VTRXPAD21	AA5	NC	NC	
N/A	RXNPAD21	AB6	NC	NC	
N/A	RXPPAD21	AB5	NC	NC	
N/A	GND A21	Y6	NC	NC	
N/A	GND A21	Y6	NC	NC	
N/A	TXPPAD21	AB4	NC	NC	
N/A	TXNPAD21	AB3	NC	NC	
N/A	VTTXPAD21	AA3	NC	NC	
N/A	AVCCAUXTX21	AA4	NC	NC	
N/A	VCCINT	U6			
N/A	VCCINT	U17			
N/A	VCCINT	T8			
N/A	VCCINT	T7			
N/A	VCCINT	T16			
N/A	VCCINT	T15			
N/A	VCCINT	R7			
N/A	VCCINT	R16			
N/A	VCCINT	H7			
N/A	VCCINT	H16			
N/A	VCCINT	G8			
N/A	VCCINT	G7			
N/A	VCCINT	G16			
N/A	VCCINT	G15			
N/A	VCCINT	F6			
N/A	VCCINT	F17			
N/A	VCCAUX	M22			
N/A	VCCAUX	L1			
N/A	VCCAUX	B21			
N/A	VCCAUX	B2			
N/A	VCCAUX	AB11			
N/A	VCCAUX	AA21			

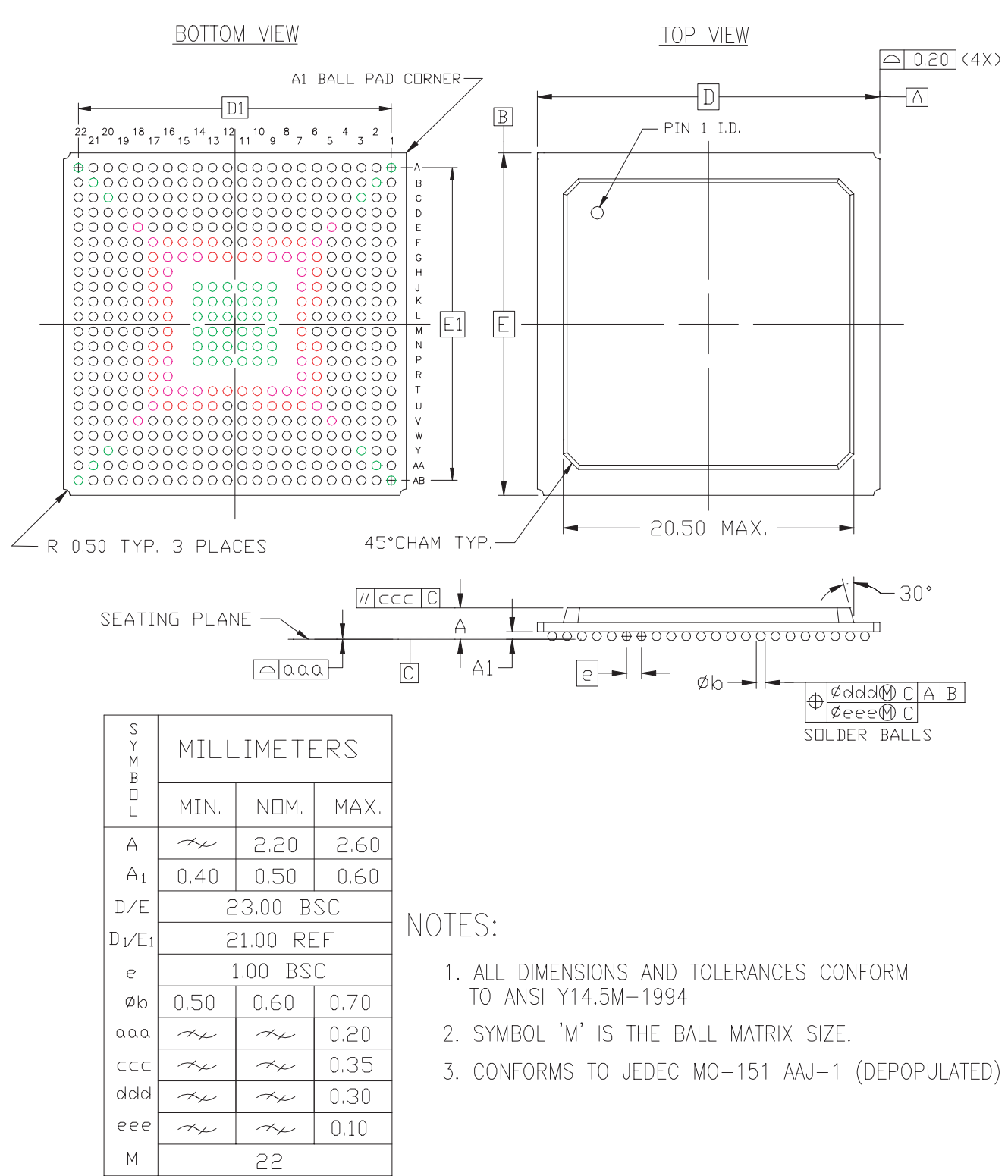
Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
N/A	VCCAUX	AA2			
N/A	VCCAUX	A12			
N/A	GND	Y3			
N/A	GND	Y20			
N/A	GND	W4			
N/A	GND	W19			
N/A	GND	V5			
N/A	GND	V18			
N/A	GND	P9			
N/A	GND	P14			
N/A	GND	P13			
N/A	GND	P12			
N/A	GND	P11			
N/A	GND	P10			
N/A	GND	N9			
N/A	GND	N14			
N/A	GND	N13			
N/A	GND	N12			
N/A	GND	N11			
N/A	GND	N10			
N/A	GND	M9			
N/A	GND	M14			
N/A	GND	M13			
N/A	GND	M12			
N/A	GND	M11			
N/A	GND	M10			
N/A	GND	M1			
N/A	GND	L9			
N/A	GND	L22			
N/A	GND	L14			
N/A	GND	L13			
N/A	GND	L12			
N/A	GND	L11			
N/A	GND	L10			
N/A	GND	K9			

Table 7: FG456 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
N/A	GND	K14			
N/A	GND	K13			
N/A	GND	K12			
N/A	GND	K11			
N/A	GND	K10			
N/A	GND	J9			
N/A	GND	J14			
N/A	GND	J13			
N/A	GND	J12			
N/A	GND	J11			
N/A	GND	J10			
N/A	GND	E5			
N/A	GND	E18			
N/A	GND	D4			
N/A	GND	D19			
N/A	GND	C3			
N/A	GND	C20			
N/A	GND	AB22			
N/A	GND	AB12			
N/A	GND	AB1			
N/A	GND	A22			
N/A	GND	A11			
N/A	GND	A1			

FG456 Fine-Pitch BGA Package Specifications (1.00mm pitch)



456-BALL FINE PITCH BGA (FG456)

Figure 2: FG456 Fine-Pitch BGA Package Specifications

FF672 Flip-Chip Fine-Pitch BGA Package

As shown in Table 8, XC2VP2, XC2VP4, and XC2VP7 Virtex-II Pro devices are available in the FF672 flip-chip fine-pitch BGA package. Pins in each of these devices are the same, except for differences shown in the "No Connects" column. Following this table are the **FF672 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)**.

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
0	IO_L01N_0/VRP_0	B24			
0	IO_L01P_0/VRN_0	A24			
0	IO_L02N_0	D21			
0	IO_L02P_0	C21			
0	IO_L03N_0	E20			
0	IO_L03P_0/VREF_0	D20			
0	IO_L05_0/No_Pair	F19			
0	IO_L06N_0	E19			
0	IO_L06P_0	E18			
0	IO_L07N_0	D19			
0	IO_L07P_0	C19			
0	IO_L08N_0	B19			
0	IO_L08P_0	A19			
0	IO_L09N_0	G18			
0	IO_L09P_0/VREF_0	F18			
0	IO_L37N_0	D18	NC	NC	
0	IO_L37P_0	C18	NC	NC	
0	IO_L38N_0	G17	NC	NC	
0	IO_L38P_0	H16	NC	NC	
0	IO_L39N_0	F17	NC	NC	
0	IO_L39P_0	F16	NC	NC	
0	IO_L43N_0	E17	NC	NC	
0	IO_L43P_0	D17	NC	NC	
0	IO_L44N_0	G16	NC	NC	
0	IO_L44P_0	G15	NC	NC	
0	IO_L45N_0	E16	NC	NC	
0	IO_L45P_0/VREF_0	D16	NC	NC	
0	IO_L67N_0	F15			
0	IO_L67P_0	E15			
0	IO_L68N_0	D15			
0	IO_L68P_0	C15			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
0	IO_L69N_0	H15			
0	IO_L69P_0/VREF_0	H14			
0	IO_L73N_0	G14			
0	IO_L73P_0	F14			
0	IO_L74N_0/GCLK7P	E14			
0	IO_L74P_0/GCLK6S	D14			
0	IO_L75N_0/GCLK5P	C14			
0	IO_L75P_0/GCLK4S	B14			
1	IO_L75N_1/GCLK3P	B13			
1	IO_L75P_1/GCLK2S	C13			
1	IO_L74N_1/GCLK1P	D13			
1	IO_L74P_1/GCLK0S	E13			
1	IO_L73N_1	F13			
1	IO_L73P_1	G13			
1	IO_L69N_1/VREF_1	H13			
1	IO_L69P_1	H12			
1	IO_L68N_1	C12			
1	IO_L68P_1	D12			
1	IO_L67N_1	E12			
1	IO_L67P_1	F12			
1	IO_L45N_1/VREF_1	D11	NC	NC	
1	IO_L45P_1	E11	NC	NC	
1	IO_L44N_1	G12	NC	NC	
1	IO_L44P_1	G11	NC	NC	
1	IO_L43N_1	D10	NC	NC	
1	IO_L43P_1	E10	NC	NC	
1	IO_L39N_1	F11	NC	NC	
1	IO_L39P_1	F10	NC	NC	
1	IO_L38N_1	H11	NC	NC	
1	IO_L38P_1	G10	NC	NC	
1	IO_L37N_1	C9	NC	NC	
1	IO_L37P_1	D9	NC	NC	
1	IO_L09N_1/VREF_1	F9			
1	IO_L09P_1	G9			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
1	IO_L08N_1	A8			
1	IO_L08P_1	B8			
1	IO_L07N_1	C8			
1	IO_L07P_1	D8			
1	IO_L06N_1	E9			
1	IO_L06P_1	E8			
1	IO_L05_1/No_Pair	F8			
1	IO_L03N_1/VREF_1	D7			
1	IO_L03P_1	E7			
1	IO_L02N_1	C6			
1	IO_L02P_1	D6			
1	IO_L01N_1/VRP_1	A3			
1	IO_L01P_1/VRN_1	B3			
2	IO_L01N_2/VRP_2	C4			
2	IO_L01P_2/VRN_2	D3			
2	IO_L02N_2	A2			
2	IO_L02P_2	B1			
2	IO_L03N_2	C2			
2	IO_L03P_2	C1			
2	IO_L04N_2/VREF_2	D2			
2	IO_L04P_2	D1			
2	IO_L05N_2	E4			
2	IO_L05P_2	E3			
2	IO_L06N_2	E2			
2	IO_L06P_2	E1			
2	IO_L40N_2/VREF_2	F5	NC	NC	NC
2	IO_L40P_2	F4	NC	NC	NC
2	IO_L42N_2	F3	NC	NC	NC
2	IO_L42P_2	F2	NC	NC	NC
2	IO_L43N_2	G6	NC		
2	IO_L43P_2	G5	NC		
2	IO_L44N_2	G4	NC		
2	IO_L44P_2	G3	NC		
2	IO_L45N_2	F1	NC		

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
2	IO_L45P_2	G1	NC		
2	IO_L46N_2/VREF_2	H6	NC		
2	IO_L46P_2	H5	NC		
2	IO_L47N_2	H4	NC		
2	IO_L47P_2	H3	NC		
2	IO_L48N_2	H2	NC		
2	IO_L48P_2	H1	NC		
2	IO_L49N_2	J7	NC		
2	IO_L49P_2	J6	NC		
2	IO_L50N_2	J5	NC		
2	IO_L50P_2	J4	NC		
2	IO_L51N_2	J3	NC		
2	IO_L51P_2	J2	NC		
2	IO_L52N_2/VREF_2	K6	NC		
2	IO_L52P_2	K5	NC		
2	IO_L53N_2	K4	NC		
2	IO_L53P_2	K3	NC		
2	IO_L54N_2	J1	NC		
2	IO_L54P_2	K1	NC		
2	IO_L55N_2	K7	NC		
2	IO_L55P_2	L8	NC		
2	IO_L56N_2	L7	NC		
2	IO_L56P_2	M7	NC		
2	IO_L57N_2	L6	NC		
2	IO_L57P_2	L5	NC		
2	IO_L58N_2/VREF_2	L4	NC		
2	IO_L58P_2	L3	NC		
2	IO_L59N_2	L2	NC		
2	IO_L59P_2	L1	NC		
2	IO_L60N_2	M8	NC		
2	IO_L60P_2	N8	NC		
2	IO_L85N_2	M6			
2	IO_L85P_2	M5			
2	IO_L86N_2	M4			
2	IO_L86P_2	M3			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
2	IO_L87N_2	M2			
2	IO_L87P_2	M1			
2	IO_L88N_2/VREF_2	N7			
2	IO_L88P_2	N6			
2	IO_L89N_2	N5			
2	IO_L89P_2	N4			
2	IO_L90N_2	N3			
2	IO_L90P_2	N2			
3	IO_L90N_3	P2			
3	IO_L90P_3	P3			
3	IO_L89N_3	P4			
3	IO_L89P_3	P5			
3	IO_L88N_3	P6			
3	IO_L88P_3	P7			
3	IO_L87N_3/VREF_3	R1			
3	IO_L87P_3	R2			
3	IO_L86N_3	R3			
3	IO_L86P_3	R4			
3	IO_L85N_3	R5			
3	IO_L85P_3	R6			
3	IO_L60N_3	P8	NC		
3	IO_L60P_3	R8	NC		
3	IO_L59N_3	T1	NC		
3	IO_L59P_3	T2	NC		
3	IO_L58N_3	T3	NC		
3	IO_L58P_3	T4	NC		
3	IO_L57N_3/VREF_3	T5	NC		
3	IO_L57P_3	T6	NC		
3	IO_L56N_3	R7	NC		
3	IO_L56P_3	T7	NC		
3	IO_L55N_3	T8	NC		
3	IO_L55P_3	U7	NC		
3	IO_L54N_3	U1	NC		
3	IO_L54P_3	V1	NC		

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
3	IO_L53N_3	U3	NC		
3	IO_L53P_3	U4	NC		
3	IO_L52N_3	U5	NC		
3	IO_L52P_3	U6	NC		
3	IO_L51N_3/VREF_3	V2	NC		
3	IO_L51P_3	V3	NC		
3	IO_L50N_3	V4	NC		
3	IO_L50P_3	V5	NC		
3	IO_L49N_3	V6	NC		
3	IO_L49P_3	V7	NC		
3	IO_L48N_3	W1	NC		
3	IO_L48P_3	W2	NC		
3	IO_L47N_3	W3	NC		
3	IO_L47P_3	W4	NC		
3	IO_L46N_3	W5	NC		
3	IO_L46P_3	W6	NC		
3	IO_L45N_3/VREF_3	Y1	NC		
3	IO_L45P_3	AA1	NC		
3	IO_L44N_3	Y3	NC		
3	IO_L44P_3	Y4	NC		
3	IO_L43N_3	Y5	NC		
3	IO_L43P_3	Y6	NC		
3	IO_L42N_3	AA2	NC	NC	NC
3	IO_L42P_3	AA3	NC	NC	NC
3	IO_L41N_3	AA4	NC	NC	NC
3	IO_L41P_3	AA5	NC	NC	NC
3	IO_L39N_3/VREF_3	AB1	NC	NC	NC
3	IO_L39P_3	AB2	NC	NC	NC
3	IO_L06N_3	AB3			
3	IO_L06P_3	AB4			
3	IO_L05N_3	AC1			
3	IO_L05P_3	AC2			
3	IO_L04N_3	AD1			
3	IO_L04P_3	AD2			
3	IO_L03N_3/VREF_3	AE1			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
3	IO_L03P_3	AF2			
3	IO_L02N_3	AC3			
3	IO_L02P_3	AD4			
3	IO_L01N_3/VRP_3	AE3			
3	IO_L01P_3/VRN_3	AF3			
4	IO_L01N_4/DOUT	AC6			
4	IO_L01P_4/INIT_B	AD6			
4	IO_L02N_4/D0	AB7			
4	IO_L02P_4/D1	AC7			
4	IO_L03N_4/D2	AA7			
4	IO_L03P_4/D3	AA8			
4	IO_L05_4/No_Pair	Y8			
4	IO_L06N_4/VRP_4	AB8			
4	IO_L06P_4/VRN_4	AB9			
4	IO_L07N_4	AC8			
4	IO_L07P_4/VREF_4	AD8			
4	IO_L08N_4	AE8			
4	IO_L08P_4	AF8			
4	IO_L09N_4	Y9			
4	IO_L09P_4/VREF_4	AA9			
4	IO_L37N_4	AC9	NC	NC	
4	IO_L37P_4	AD9	NC	NC	
4	IO_L38N_4	Y10	NC	NC	
4	IO_L38P_4	W11	NC	NC	
4	IO_L39N_4	AA10	NC	NC	
4	IO_L39P_4	AA11	NC	NC	
4	IO_L43N_4	AB10	NC	NC	
4	IO_L43P_4	AC10	NC	NC	
4	IO_L44N_4	Y11	NC	NC	
4	IO_L44P_4	Y12	NC	NC	
4	IO_L45N_4	AB11	NC	NC	
4	IO_L45P_4/VREF_4	AC11	NC	NC	
4	IO_L67N_4	AA12			
4	IO_L67P_4	AB12			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
4	IO_L68N_4	AC12			
4	IO_L68P_4	AD12			
4	IO_L69N_4	W12			
4	IO_L69P_4/VREF_4	W13			
4	IO_L73N_4	Y13			
4	IO_L73P_4	AA13			
4	IO_L74N_4/GCLK3S	AB13			
4	IO_L74P_4/GCLK2P	AC13			
4	IO_L75N_4/GCLK1S	AD13			
4	IO_L75P_4/GCLK0P	AE13			
5	IO_L75N_5/GCLK7S	AE14			
5	IO_L75P_5/GCLK6P	AD14			
5	IO_L74N_5/GCLK5S	AC14			
5	IO_L74P_5/GCLK4P	AB14			
5	IO_L73N_5	AA14			
5	IO_L73P_5	Y14			
5	IO_L69N_5/VREF_5	W14			
5	IO_L69P_5	W15			
5	IO_L68N_5	AD15			
5	IO_L68P_5	AC15			
5	IO_L67N_5	AB15			
5	IO_L67P_5	AA15			
5	IO_L45N_5/VREF_5	AC16	NC	NC	
5	IO_L45P_5	AB16	NC	NC	
5	IO_L44N_5	Y15	NC	NC	
5	IO_L44P_5	Y16	NC	NC	
5	IO_L43N_5	AC17	NC	NC	
5	IO_L43P_5	AB17	NC	NC	
5	IO_L39N_5	AA16	NC	NC	
5	IO_L39P_5	AA17	NC	NC	
5	IO_L38N_5	W16	NC	NC	
5	IO_L38P_5	Y17	NC	NC	
5	IO_L37N_5	AD18	NC	NC	
5	IO_L37P_5	AC18	NC	NC	

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
5	IO_L09N_5/VREF_5	AA18			
5	IO_L09P_5	Y18			
5	IO_L08N_5	AF19			
5	IO_L08P_5	AE19			
5	IO_L07N_5/VREF_5	AD19			
5	IO_L07P_5	AC19			
5	IO_L06N_5/VRP_5	AB18			
5	IO_L06P_5/VRN_5	AB19			
5	IO_L05_5/No_Pair	Y19			
5	IO_L03N_5/D4	AA19			
5	IO_L03P_5/D5	AA20			
5	IO_L02N_5/D6	AC20			
5	IO_L02P_5/D7	AB20			
5	IO_L01N_5/RDWR_B	AD21			
5	IO_L01P_5/CS_B	AC21			
6	IO_L01P_6/VRN_6	AF24			
6	IO_L01N_6/VRP_6	AE24			
6	IO_L02P_6	AD23			
6	IO_L02N_6	AC24			
6	IO_L03P_6	AE26			
6	IO_L03N_6/VREF_6	AF25			
6	IO_L04P_6	AD25			
6	IO_L04N_6	AD26			
6	IO_L05P_6	AC25			
6	IO_L05N_6	AC26			
6	IO_L06P_6	AB23			
6	IO_L06N_6	AB24			
6	IO_L39P_6	AB25	NC	NC	NC
6	IO_L39N_6/VREF_6	AB26	NC	NC	NC
6	IO_L41P_6	AA22	NC	NC	NC
6	IO_L41N_6	AA23	NC	NC	NC
6	IO_L42P_6	AA24	NC	NC	NC
6	IO_L42N_6	AA25	NC	NC	NC
6	IO_L43P_6	Y21	NC		

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
6	IO_L43N_6	Y22	NC		
6	IO_L44P_6	Y23	NC		
6	IO_L44N_6	Y24	NC		
6	IO_L45P_6	AA26	NC		
6	IO_L45N_6/VREF_6	Y26	NC		
6	IO_L46P_6	W21	NC		
6	IO_L46N_6	W22	NC		
6	IO_L47P_6	W23	NC		
6	IO_L47N_6	W24	NC		
6	IO_L48P_6	W25	NC		
6	IO_L48N_6	W26	NC		
6	IO_L49P_6	V20	NC		
6	IO_L49N_6	V21	NC		
6	IO_L50P_6	V22	NC		
6	IO_L50N_6	V23	NC		
6	IO_L51P_6	V24	NC		
6	IO_L51N_6/VREF_6	V25	NC		
6	IO_L52P_6	U21	NC		
6	IO_L52N_6	U22	NC		
6	IO_L53P_6	U23	NC		
6	IO_L53N_6	U24	NC		
6	IO_L54P_6	V26	NC		
6	IO_L54N_6	U26	NC		
6	IO_L55P_6	U20	NC		
6	IO_L55N_6	T19	NC		
6	IO_L56P_6	T20	NC		
6	IO_L56N_6	R20	NC		
6	IO_L57P_6	T21	NC		
6	IO_L57N_6/VREF_6	T22	NC		
6	IO_L58P_6	T23	NC		
6	IO_L58N_6	T24	NC		
6	IO_L59P_6	T25	NC		
6	IO_L59N_6	T26	NC		
6	IO_L60P_6	R19	NC		
6	IO_L60N_6	P19	NC		

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
6	IO_L85P_6	R21			
6	IO_L85N_6	R22			
6	IO_L86P_6	R23			
6	IO_L86N_6	R24			
6	IO_L87P_6	R25			
6	IO_L87N_6/VREF_6	R26			
6	IO_L88P_6	P20			
6	IO_L88N_6	P21			
6	IO_L89P_6	P22			
6	IO_L89N_6	P23			
6	IO_L90P_6	P24			
6	IO_L90N_6	P25			
7	IO_L90P_7	N25			
7	IO_L90N_7	N24			
7	IO_L89P_7	N23			
7	IO_L89N_7	N22			
7	IO_L88P_7	N21			
7	IO_L88N_7/VREF_7	N20			
7	IO_L87P_7	M26			
7	IO_L87N_7	M25			
7	IO_L86P_7	M24			
7	IO_L86N_7	M23			
7	IO_L85P_7	M22			
7	IO_L85N_7	M21			
7	IO_L60P_7	N19	NC		
7	IO_L60N_7	M19	NC		
7	IO_L59P_7	L26	NC		
7	IO_L59N_7	L25	NC		
7	IO_L58P_7	L24	NC		
7	IO_L58N_7/VREF_7	L23	NC		
7	IO_L57P_7	L22	NC		
7	IO_L57N_7	L21	NC		
7	IO_L56P_7	M20	NC		
7	IO_L56N_7	L20	NC		

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
7	IO_L55P_7	L19	NC		
7	IO_L55N_7	K20	NC		
7	IO_L54P_7	K26	NC		
7	IO_L54N_7	J26	NC		
7	IO_L53P_7	K24	NC		
7	IO_L53N_7	K23	NC		
7	IO_L52P_7	K22	NC		
7	IO_L52N_7/VREF_7	K21	NC		
7	IO_L51P_7	J25	NC		
7	IO_L51N_7	J24	NC		
7	IO_L50P_7	J23	NC		
7	IO_L50N_7	J22	NC		
7	IO_L49P_7	J21	NC		
7	IO_L49N_7	J20	NC		
7	IO_L48P_7	H26	NC		
7	IO_L48N_7	H25	NC		
7	IO_L47P_7	H24	NC		
7	IO_L47N_7	H23	NC		
7	IO_L46P_7	H22	NC		
7	IO_L46N_7/VREF_7	H21	NC		
7	IO_L45P_7	G26	NC		
7	IO_L45N_7	F26	NC		
7	IO_L44P_7	G24	NC		
7	IO_L44N_7	G23	NC		
7	IO_L43P_7	G22	NC		
7	IO_L43N_7	G21	NC		
7	IO_L42P_7	F25	NC	NC	NC
7	IO_L42N_7	F24	NC	NC	NC
7	IO_L40P_7	F23	NC	NC	NC
7	IO_L40N_7/VREF_7	F22	NC	NC	NC
7	IO_L06P_7	E26			
7	IO_L06N_7	E25			
7	IO_L05P_7	E24			
7	IO_L05N_7	E23			
7	IO_L04P_7	D26			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
7	IO_L04N_7/VREF_7	D25			
7	IO_L03P_7	C26			
7	IO_L03N_7	C25			
7	IO_L02P_7	B26			
7	IO_L02N_7	A25			
7	IO_L01P_7/VRN_7	D24			
7	IO_L01N_7/VRP_7	C23			
0	VCCO_0	C17			
0	VCCO_0	C20			
0	VCCO_0	H17			
0	VCCO_0	H18			
0	VCCO_0	J14			
0	VCCO_0	J15			
0	VCCO_0	J16			
1	VCCO_1	C7			
1	VCCO_1	H9			
1	VCCO_1	C10			
1	VCCO_1	H10			
1	VCCO_1	J11			
1	VCCO_1	J12			
1	VCCO_1	J13			
2	VCCO_2	G2			
2	VCCO_2	J8			
2	VCCO_2	K2			
2	VCCO_2	K8			
2	VCCO_2	L9			
2	VCCO_2	M9			
2	VCCO_2	N9			
3	VCCO_3	P9			
3	VCCO_3	R9			
3	VCCO_3	T9			
3	VCCO_3	U2			
3	VCCO_3	U8			
3	VCCO_3	V8			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
3	VCCO_3	Y2			
4	VCCO_4	W9			
4	VCCO_4	AD7			
4	VCCO_4	V11			
4	VCCO_4	V12			
4	VCCO_4	V13			
4	VCCO_4	W10			
4	VCCO_4	AD10			
5	VCCO_5	V14			
5	VCCO_5	V15			
5	VCCO_5	V16			
5	VCCO_5	W17			
5	VCCO_5	W18			
5	VCCO_5	AD17			
5	VCCO_5	AD20			
6	VCCO_6	P18			
6	VCCO_6	R18			
6	VCCO_6	T18			
6	VCCO_6	U19			
6	VCCO_6	U25			
6	VCCO_6	V19			
6	VCCO_6	Y25			
7	VCCO_7	G25			
7	VCCO_7	J19			
7	VCCO_7	K19			
7	VCCO_7	K25			
7	VCCO_7	L18			
7	VCCO_7	M18			
7	VCCO_7	N18			
N/A	CCLK	W7			
N/A	PROG_B	D22			
N/A	DONE	AB6			
N/A	M0	AC22			
N/A	M1	W20			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
N/A	M2	AB21			
N/A	TCK	G8			
N/A	TDI	H20			
N/A	TDO	H7			
N/A	TMS	F7			
N/A	PWRDWN_B	AC5			
N/A	HSWAP_EN	E21			
N/A	RSVD	D5			
N/A	VBATT	E6			
N/A	DXP	F20			
N/A	DXN	G19			
N/A	AVCCAUXTX7	B11			
N/A	VTTXPAD7	B12			
N/A	TXNPAD7	A12			
N/A	TXPPAD7	A11			
N/A	GND A7	C11			
N/A	GND A7	C11			
N/A	RXPPAD7	A10			
N/A	RXNPAD7	A9			
N/A	VTRXPAD7	B10			
N/A	AVCCAUXRX7	B9			
N/A	AVCCAUXTX9	B6	NC	NC	
N/A	VTTXPAD9	B7	NC	NC	
N/A	TXNPAD9	A7	NC	NC	
N/A	TXPPAD9	A6	NC	NC	
N/A	GND A9	C5	NC	NC	
N/A	GND A9	C5	NC	NC	
N/A	RXPPAD9	A5	NC	NC	
N/A	RXNPAD9	A4	NC	NC	
N/A	VTRXPAD9	B5	NC	NC	
N/A	AVCCAUXRX9	B4	NC	NC	
N/A	AVCCAUXRX16	AE4	NC	NC	
N/A	VTRXPAD16	AE5	NC	NC	
N/A	RXNPAD16	AF4	NC	NC	
N/A	RXPPAD16	AF5	NC	NC	

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
N/A	GND A16	AD5	NC	NC	
N/A	GND A16	AD5	NC	NC	
N/A	TXPPAD16	AF6	NC	NC	
N/A	TXNPAD16	AF7	NC	NC	
N/A	VTTXPAD16	AE7	NC	NC	
N/A	AVCCAUXTX16	AE6	NC	NC	
N/A	AVCCAUXRX18	AE9			
N/A	VTRXPAD18	AE10			
N/A	RXNPAD18	AF9			
N/A	RXPPAD18	AF10			
N/A	GND A18	AD11			
N/A	GND A18	AD11			
N/A	TXPPAD18	AF11			
N/A	TXNPAD18	AF12			
N/A	VTTXPAD18	AE12			
N/A	AVCCAUXTX18	AE11			
N/A	AVCCAUXTX4	B22	NC	NC	
N/A	VTTXPAD4	B23	NC	NC	
N/A	TXNPAD4	A23	NC	NC	
N/A	TXPPAD4	A22	NC	NC	
N/A	GND A4	C22	NC	NC	
N/A	GND A4	C22	NC	NC	
N/A	RXPPAD4	A21	NC	NC	
N/A	RXNPAD4	A20	NC	NC	
N/A	VTRXPAD4	B21	NC	NC	
N/A	AVCCAUXRX4	B20	NC	NC	
N/A	AVCCAUXTX6	B17			
N/A	VTTXPAD6	B18			
N/A	TXNPAD6	A18			
N/A	TXPPAD6	A17			
N/A	GND A6	C16			
N/A	GND A6	C16			
N/A	RXPPAD6	A16			
N/A	RXNPAD6	A15			
N/A	VTRXPAD6	B16			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
N/A	AVCCAUXRX6	B15			
N/A	AVCCAUXRX19	AE15			
N/A	VTRXPAD19	AE16			
N/A	RXNPAD19	AF15			
N/A	RXPPAD19	AF16			
N/A	GND A19	AD16			
N/A	GND A19	AD16			
N/A	TXPPAD19	AF17			
N/A	TXNPAD19	AF18			
N/A	VTTXPAD19	AE18			
N/A	AVCCAUXTX19	AE17			
N/A	AVCCAUXRX21	AE20	NC	NC	
N/A	VTRXPAD21	AE21	NC	NC	
N/A	RXNPAD21	AF20	NC	NC	
N/A	RXPPAD21	AF21	NC	NC	
N/A	GND A21	AD22	NC	NC	
N/A	GND A21	AD22	NC	NC	
N/A	TXPPAD21	AF22	NC	NC	
N/A	TXNPAD21	AF23	NC	NC	
N/A	VTTXPAD21	AE23	NC	NC	
N/A	AVCCAUXTX21	AE22	NC	NC	
N/A	VCCINT	H8			
N/A	VCCINT	J9			
N/A	VCCINT	K9			
N/A	VCCINT	U9			
N/A	VCCINT	V9			
N/A	VCCINT	W8			
N/A	VCCINT	H19			
N/A	VCCINT	J10			
N/A	VCCINT	J17			
N/A	VCCINT	J18			
N/A	VCCINT	K11			
N/A	VCCINT	K16			
N/A	VCCINT	K18			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
N/A	VCCINT	L10			
N/A	VCCINT	L17			
N/A	VCCINT	T10			
N/A	VCCINT	T17			
N/A	VCCINT	U11			
N/A	VCCINT	U16			
N/A	VCCINT	U18			
N/A	VCCINT	V10			
N/A	VCCINT	V17			
N/A	VCCINT	V18			
N/A	VCCINT	W19			
N/A	VCCAUX	B2			
N/A	VCCAUX	N1			
N/A	VCCAUX	P1			
N/A	VCCAUX	A13			
N/A	VCCAUX	A14			
N/A	VCCAUX	AE2			
N/A	VCCAUX	B25			
N/A	VCCAUX	N26			
N/A	VCCAUX	P26			
N/A	VCCAUX	AE25			
N/A	VCCAUX	AF13			
N/A	VCCAUX	AF14			
N/A	GND	C3			
N/A	GND	D4			
N/A	GND	E5			
N/A	GND	F6			
N/A	GND	G7			
N/A	GND	Y7			
N/A	GND	AA6			
N/A	GND	AB5			
N/A	GND	AC4			
N/A	GND	AD3			
N/A	GND	C24			
N/A	GND	D23			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
N/A	GND	E22			
N/A	GND	F21			
N/A	GND	G20			
N/A	GND	K10			
N/A	GND	K12			
N/A	GND	K13			
N/A	GND	K14			
N/A	GND	K15			
N/A	GND	K17			
N/A	GND	L11			
N/A	GND	L12			
N/A	GND	L13			
N/A	GND	L14			
N/A	GND	L15			
N/A	GND	L16			
N/A	GND	M10			
N/A	GND	M11			
N/A	GND	M12			
N/A	GND	M13			
N/A	GND	M14			
N/A	GND	M15			
N/A	GND	M16			
N/A	GND	M17			
N/A	GND	N10			
N/A	GND	N11			
N/A	GND	N12			
N/A	GND	N13			
N/A	GND	N14			
N/A	GND	N15			
N/A	GND	N16			
N/A	GND	N17			
N/A	GND	P10			
N/A	GND	P11			
N/A	GND	P12			
N/A	GND	P13			

Table 8: FF672 — XC2VP2, XC2VP4, and XC2VP7

Bank	Pin Description	Pin Number	No Connects		
			XC2V P2	XC2V P4	XC2V P7
N/A	GND	P14			
N/A	GND	P15			
N/A	GND	P16			
N/A	GND	P17			
N/A	GND	R10			
N/A	GND	R11			
N/A	GND	R12			
N/A	GND	R13			
N/A	GND	R14			
N/A	GND	R15			
N/A	GND	R16			
N/A	GND	R17			
N/A	GND	T11			
N/A	GND	T12			
N/A	GND	T13			
N/A	GND	T14			
N/A	GND	T15			
N/A	GND	T16			
N/A	GND	U10			
N/A	GND	U12			
N/A	GND	U13			
N/A	GND	U14			
N/A	GND	U15			
N/A	GND	U17			
N/A	GND	Y20			
N/A	GND	AA21			
N/A	GND	AB22			
N/A	GND	AC23			
N/A	GND	AD24			

FF672 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)

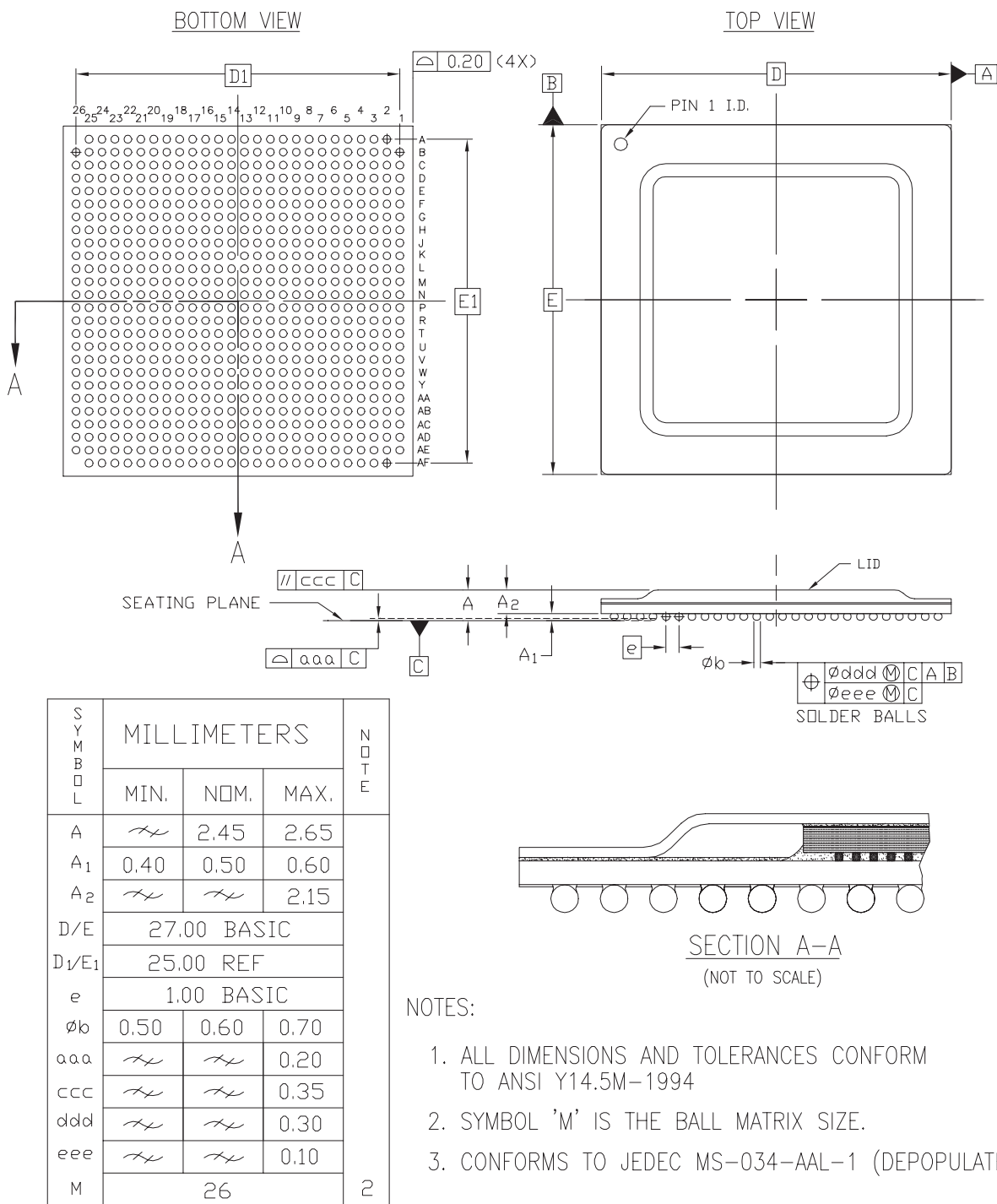


Figure 3: FF672 Flip-Chip Fine-Pitch BGA Package Specifications

FF896 Flip-Chip Fine-Pitch BGA Package

As shown in [Table 9](#), the XC2VP7 and XC2VP20 Virtex-II Pro devices are available in the FF896 flip-chip fine-pitch BGA package. Pins in each of these devices are the same, except for differences shown in the "No Connects" column. Following this table are the **FF896 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)**.

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
0	IO_L01N_0/VRP_0	E25		
0	IO_L01P_0/VRN_0	E24		
0	IO_L02N_0	F24		
0	IO_L02P_0	F23		
0	IO_L03N_0	E23		
0	IO_L03P_0/VREF_0	E22		
0	IO_L05_0/No_Pair	G23		
0	IO_L06N_0	H22		
0	IO_L06P_0	G22		
0	IO_L07N_0	F22		
0	IO_L07P_0	F21		
0	IO_L08N_0	D24		
0	IO_L08P_0	C24		
0	IO_L09N_0	H21		
0	IO_L09P_0/VREF_0	G21		
0	IO_L37N_0	E21		
0	IO_L37P_0	D21		
0	IO_L38N_0	D23		
0	IO_L38P_0	C23		
0	IO_L39N_0	H20		
0	IO_L39P_0	G20		
0	IO_L43N_0	E20		
0	IO_L43P_0	D20		
0	IO_L44N_0	B23		
0	IO_L44P_0	A23		
0	IO_L45N_0	H19		
0	IO_L45P_0/VREF_0	G19		
0	IO_L46N_0	E19	NC	
0	IO_L46P_0	E18	NC	
0	IO_L47N_0	C22	NC	
0	IO_L47P_0	B22	NC	
0	IO_L48N_0	F20	NC	
0	IO_L48P_0	F19	NC	

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
0	IO_L49N_0	G17	NC	
0	IO_L49P_0	F17	NC	
0	IO_L50_0/No_Pair	B21	NC	
0	IO_L53_0/No_Pair	A21	NC	
0	IO_L54N_0	H18	NC	
0	IO_L54P_0	G18	NC	
0	IO_L56N_0	C21	NC	
0	IO_L56P_0	C20	NC	
0	IO_L57N_0	J17	NC	
0	IO_L57P_0/VREF_0	H17	NC	
0	IO_L67N_0	E17		
0	IO_L67P_0	D17		
0	IO_L68N_0	D18		
0	IO_L68P_0	C18		
0	IO_L69N_0	J16		
0	IO_L69P_0/VREF_0	H16		
0	IO_L73N_0	E16		
0	IO_L73P_0	D16		
0	IO_L74N_0/GCLK7P	C16		
0	IO_L74P_0/GCLK6S	B16		
0	IO_L75N_0/GCLK5P	G16		
0	IO_L75P_0/GCLK4S	F16		
1	IO_L75N_1/GCLK3P	F15		
1	IO_L75P_1/GCLK2S	G15		
1	IO_L74N_1/GCLK1P	B15		
1	IO_L74P_1/GCLK0S	C15		
1	IO_L73N_1	D15		
1	IO_L73P_1	E15		
1	IO_L69N_1/VREF_1	H15		
1	IO_L69P_1	J15		
1	IO_L68N_1	C13		
1	IO_L68P_1	D13		
1	IO_L67N_1	D14		
1	IO_L67P_1	E14		
1	IO_L57N_1/VREF_1	H14	NC	
1	IO_L57P_1	J14	NC	

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
1	IO_L56N_1	C11	NC	
1	IO_L56P_1	C10	NC	
1	IO_L54N_1	G13	NC	
1	IO_L54P_1	H13	NC	
1	IO_L53_1/No_Pair	A10	NC	
1	IO_L50_1/No_Pair	B10	NC	
1	IO_L49N_1	F14	NC	
1	IO_L49P_1	G14	NC	
1	IO_L48N_1	F12	NC	
1	IO_L48P_1	F11	NC	
1	IO_L47N_1	B9	NC	
1	IO_L47P_1	C9	NC	
1	IO_L46N_1	E13	NC	
1	IO_L46P_1	E12	NC	
1	IO_L45N_1/VREF_1	G12		
1	IO_L45P_1	H12		
1	IO_L44N_1	A8		
1	IO_L44P_1	B8		
1	IO_L43N_1	D11		
1	IO_L43P_1	E11		
1	IO_L39N_1	G11		
1	IO_L39P_1	H11		
1	IO_L38N_1	C8		
1	IO_L38P_1	D8		
1	IO_L37N_1	D10		
1	IO_L37P_1	E10		
1	IO_L09N_1/VREF_1	G10		
1	IO_L09P_1	H10		
1	IO_L08N_1	C7		
1	IO_L08P_1	D7		
1	IO_L07N_1	F10		
1	IO_L07P_1	F9		
1	IO_L06N_1	G9		
1	IO_L06P_1	H9		
1	IO_L05_1/No_Pair	G8		
1	IO_L03N_1/VREF_1	E9		
1	IO_L03P_1	E8		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
1	IO_L02N_1	F8		
1	IO_L02P_1	F7		
1	IO_L01N_1/VRP_1	E7		
1	IO_L01P_1/VRN_1	E6		
2	IO_L01N_2/VRP_2	A3		
2	IO_L01P_2/VRN_2	B3		
2	IO_L02N_2	G6		
2	IO_L02P_2	G5		
2	IO_L03N_2	C5		
2	IO_L03P_2	D5		
2	IO_L04N_2/VREF_2	C2		
2	IO_L04P_2	C1		
2	IO_L05N_2	J8		
2	IO_L05P_2	J7		
2	IO_L06N_2	C4		
2	IO_L06P_2	D3		
2	IO_L31N_2	D2	NC	
2	IO_L31P_2	D1	NC	
2	IO_L32N_2	H6	NC	
2	IO_L32P_2	H5	NC	
2	IO_L33N_2	E4	NC	
2	IO_L33P_2	E3	NC	
2	IO_L34N_2/VREF_2	E2	NC	
2	IO_L34P_2	E1	NC	
2	IO_L35N_2	K8	NC	
2	IO_L35P_2	K7	NC	
2	IO_L36N_2	F4	NC	
2	IO_L36P_2	F3	NC	
2	IO_L37N_2	F2	NC	
2	IO_L37P_2	F1	NC	
2	IO_L38N_2	J6	NC	
2	IO_L38P_2	J5	NC	
2	IO_L39N_2	G4	NC	
2	IO_L39P_2	G3	NC	
2	IO_L40N_2/VREF_2	G2	NC	
2	IO_L40P_2	G1	NC	

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
2	IO_L41N_2	L8	NC	
2	IO_L41P_2	L7	NC	
2	IO_L42N_2	H4	NC	
2	IO_L42P_2	H3	NC	
2	IO_L43N_2	H2		
2	IO_L43P_2	J2		
2	IO_L44N_2	M8		
2	IO_L44P_2	M7		
2	IO_L45N_2	K6		
2	IO_L45P_2	K5		
2	IO_L46N_2/VREF_2	J1		
2	IO_L46P_2	K1		
2	IO_L47N_2	M6		
2	IO_L47P_2	M5		
2	IO_L48N_2	J4		
2	IO_L48P_2	J3		
2	IO_L49N_2	K2		
2	IO_L49P_2	L2		
2	IO_L50N_2	N8		
2	IO_L50P_2	N7		
2	IO_L51N_2	K4		
2	IO_L51P_2	K3		
2	IO_L52N_2/VREF_2	L1		
2	IO_L52P_2	M1		
2	IO_L53N_2	N6		
2	IO_L53P_2	N5		
2	IO_L54N_2	L5		
2	IO_L54P_2	L4		
2	IO_L55N_2	M2		
2	IO_L55P_2	N2		
2	IO_L56N_2	P9		
2	IO_L56P_2	R9		
2	IO_L57N_2	M4		
2	IO_L57P_2	M3		
2	IO_L58N_2/VREF_2	N1		
2	IO_L58P_2	P1		
2	IO_L59N_2	P8		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
2	IO_L59P_2	P7		
2	IO_L60N_2	N4		
2	IO_L60P_2	N3		
2	IO_L85N_2	P3		
2	IO_L85P_2	P2		
2	IO_L86N_2	R8		
2	IO_L86P_2	R7		
2	IO_L87N_2	P5		
2	IO_L87P_2	P4		
2	IO_L88N_2/VREF_2	R2		
2	IO_L88P_2	T2		
2	IO_L89N_2	R6		
2	IO_L89P_2	R5		
2	IO_L90N_2	R4		
2	IO_L90P_2	R3		
3	IO_L90N_3	U1		
3	IO_L90P_3	V1		
3	IO_L89N_3	T5		
3	IO_L89P_3	T6		
3	IO_L88N_3	T3		
3	IO_L88P_3	T4		
3	IO_L87N_3/VREF_3	U2		
3	IO_L87P_3	U3		
3	IO_L86N_3	T7		
3	IO_L86P_3	T8		
3	IO_L85N_3	U4		
3	IO_L85P_3	U5		
3	IO_L60N_3	V2		
3	IO_L60P_3	W2		
3	IO_L59N_3	T9		
3	IO_L59P_3	U9		
3	IO_L58N_3	V3		
3	IO_L58P_3	V4		
3	IO_L57N_3/VREF_3	W1		
3	IO_L57P_3	Y1		
3	IO_L56N_3	U7		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
3	IO_L56P_3	U8		
3	IO_L55N_3	V5		
3	IO_L55P_3	V6		
3	IO_L54N_3	Y2		
3	IO_L54P_3	AA2		
3	IO_L53N_3	V7		
3	IO_L53P_3	V8		
3	IO_L52N_3	W3		
3	IO_L52P_3	W4		
3	IO_L51N_3/VREF_3	AA1		
3	IO_L51P_3	AB1		
3	IO_L50N_3	W5		
3	IO_L50P_3	W6		
3	IO_L49N_3	Y4		
3	IO_L49P_3	Y5		
3	IO_L48N_3	AA3		
3	IO_L48P_3	AA4		
3	IO_L47N_3	W7		
3	IO_L47P_3	W8		
3	IO_L46N_3	AB3		
3	IO_L46P_3	AB4		
3	IO_L45N_3/VREF_3	AB2		
3	IO_L45P_3	AC2		
3	IO_L44N_3	AA5		
3	IO_L44P_3	AA6		
3	IO_L43N_3	AC3		
3	IO_L43P_3	AC4		
3	IO_L42N_3	AD1	NC	
3	IO_L42P_3	AD2	NC	
3	IO_L41N_3	Y7	NC	
3	IO_L41P_3	Y8	NC	
3	IO_L40N_3	AB5	NC	
3	IO_L40P_3	AB6	NC	
3	IO_L39N_3/VREF_3	AE1	NC	
3	IO_L39P_3	AE2	NC	
3	IO_L38N_3	AA7	NC	
3	IO_L38P_3	AA8	NC	

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
3	IO_L37N_3	AD3	NC	
3	IO_L37P_3	AD4	NC	
3	IO_L36N_3	AF1	NC	
3	IO_L36P_3	AF2	NC	
3	IO_L35N_3	AC5	NC	
3	IO_L35P_3	AC6	NC	
3	IO_L34N_3	AF3	NC	
3	IO_L34P_3	AF4	NC	
3	IO_L33N_3/VREF_3	AE3	NC	
3	IO_L33P_3	AE4	NC	
3	IO_L32N_3	AB7	NC	
3	IO_L32P_3	AB8	NC	
3	IO_L31N_3	AE5	NC	
3	IO_L31P_3	AF6	NC	
3	IO_L06N_3	AG1		
3	IO_L06P_3	AG2		
3	IO_L05N_3	AD5		
3	IO_L05P_3	AD6		
3	IO_L04N_3	AG3		
3	IO_L04P_3	AH4		
3	IO_L03N_3/VREF_3	AH1		
3	IO_L03P_3	AH2		
3	IO_L02N_3	AG5		
3	IO_L02P_3	AH5		
3	IO_L01N_3/VRP_3	AJ3		
3	IO_L01P_3/VRN_3	AK3		
4	IO_L01N_4/DOUT	AG6		
4	IO_L01P_4/INIT_B	AF7		
4	IO_L02N_4/D0	AC9		
4	IO_L02P_4/D1	AD9		
4	IO_L03N_4/D2	AG7		
4	IO_L03P_4/D3	AH7		
4	IO_L05_4/No_Pair	AD8		
4	IO_L06N_4/VRP_4	AG8		
4	IO_L06P_4/VRN_4	AH8		
4	IO_L07N_4	AC10		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
4	IO_L07P_4/VREF_4	AD10		
4	IO_L08N_4	AE7		
4	IO_L08P_4	AE8		
4	IO_L09N_4	AJ8		
4	IO_L09P_4/VREF_4	AK8		
4	IO_L37N_4	AC11		
4	IO_L37P_4	AD11		
4	IO_L38N_4	AF8		
4	IO_L38P_4	AF9		
4	IO_L39N_4	AF10		
4	IO_L39P_4	AG10		
4	IO_L43N_4	AC12		
4	IO_L43P_4	AD12		
4	IO_L44N_4	AE9		
4	IO_L44P_4	AE10		
4	IO_L45N_4	AH9		
4	IO_L45P_4/VREF_4	AJ9		
4	IO_L46N_4	AC13	NC	
4	IO_L46P_4	AD13	NC	
4	IO_L47N_4	AE11	NC	
4	IO_L47P_4	AE12	NC	
4	IO_L48N_4	AH10	NC	
4	IO_L48P_4	AH11	NC	
4	IO_L49N_4	AB14	NC	
4	IO_L49P_4	AC14	NC	
4	IO_L50_4/No_Pair	AF11	NC	
4	IO_L53_4/No_Pair	AG11	NC	
4	IO_L54N_4	AJ10	NC	
4	IO_L54P_4	AK10	NC	
4	IO_L56N_4	AF12	NC	
4	IO_L56P_4	AF13	NC	
4	IO_L57N_4	AG13	NC	
4	IO_L57P_4/VREF_4	AH13	NC	
4	IO_L67N_4	AB15		
4	IO_L67P_4	AC15		
4	IO_L68N_4	AD14		
4	IO_L68P_4	AE14		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
4	IO_L69N_4	AF14		
4	IO_L69P_4/VREF_4	AG14		
4	IO_L73N_4	AD15		
4	IO_L73P_4	AE15		
4	IO_L74N_4/GCLK3S	AF15		
4	IO_L74P_4/GCLK2P	AG15		
4	IO_L75N_4/GCLK1S	AH15		
4	IO_L75P_4/GCLK0P	AJ15		
5	IO_L75N_5/GCLK7S	AJ16		
5	IO_L75P_5/GCLK6P	AH16		
5	IO_L74N_5/GCLK5S	AG16		
5	IO_L74P_5/GCLK4P	AF16		
5	IO_L73N_5	AE16		
5	IO_L73P_5	AD16		
5	IO_L69N_5/VREF_5	AG17		
5	IO_L69P_5	AF17		
5	IO_L68N_5	AE17		
5	IO_L68P_5	AD17		
5	IO_L67N_5	AC16		
5	IO_L67P_5	AB16		
5	IO_L57N_5/VREF_5	AH18	NC	
5	IO_L57P_5	AG18	NC	
5	IO_L56N_5	AF18	NC	
5	IO_L56P_5	AF19	NC	
5	IO_L54N_5	AK21	NC	
5	IO_L54P_5	AJ21	NC	
5	IO_L53_5/No_Pair	AG20	NC	
5	IO_L50_5/No_Pair	AF20	NC	
5	IO_L49N_5	AC17	NC	
5	IO_L49P_5	AB17	NC	
5	IO_L48N_5	AH20	NC	
5	IO_L48P_5	AH21	NC	
5	IO_L47N_5	AE19	NC	
5	IO_L47P_5	AE20	NC	
5	IO_L46N_5	AD18	NC	
5	IO_L46P_5	AC18	NC	

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
5	IO_L45N_5/VREF_5	AJ22		
5	IO_L45P_5	AH22		
5	IO_L44N_5	AE21		
5	IO_L44P_5	AE22		
5	IO_L43N_5	AD19		
5	IO_L43P_5	AC19		
5	IO_L39N_5	AG21		
5	IO_L39P_5	AF21		
5	IO_L38N_5	AF22		
5	IO_L38P_5	AF23		
5	IO_L37N_5	AD20		
5	IO_L37P_5	AC20		
5	IO_L09N_5/VREF_5	AK23		
5	IO_L09P_5	AJ23		
5	IO_L08N_5	AE23		
5	IO_L08P_5	AE24		
5	IO_L07N_5/VREF_5	AD21		
5	IO_L07P_5	AC21		
5	IO_L06N_5/VRP_5	AH23		
5	IO_L06P_5/VRN_5	AG23		
5	IO_L05_5/No_Pair	AD23		
5	IO_L03N_5/D4	AH24		
5	IO_L03P_5/D5	AG24		
5	IO_L02N_5/D6	AD22		
5	IO_L02P_5/D7	AC22		
5	IO_L01N_5/RDWR_B	AF24		
5	IO_L01P_5/CS_B	AG25		
6	IO_L01P_6/VRN_6	AK28		
6	IO_L01N_6/VRP_6	AJ28		
6	IO_L02P_6	AH26		
6	IO_L02N_6	AG26		
6	IO_L03P_6	AH29		
6	IO_L03N_6/VREF_6	AH30		
6	IO_L04P_6	AH27		
6	IO_L04N_6	AG28		
6	IO_L05P_6	AD25		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
6	IO_L05N_6	AD26		
6	IO_L06P_6	AG29		
6	IO_L06N_6	AG30		
6	IO_L31P_6	AF25	NC	
6	IO_L31N_6	AE26	NC	
6	IO_L32P_6	AB23	NC	
6	IO_L32N_6	AB24	NC	
6	IO_L33P_6	AE27	NC	
6	IO_L33N_6/VREF_6	AE28	NC	
6	IO_L34P_6	AF27	NC	
6	IO_L34N_6	AF28	NC	
6	IO_L35P_6	AC25	NC	
6	IO_L35N_6	AC26	NC	
6	IO_L36P_6	AF29	NC	
6	IO_L36N_6	AF30	NC	
6	IO_L37P_6	AD27	NC	
6	IO_L37N_6	AD28	NC	
6	IO_L38P_6	AA23	NC	
6	IO_L38N_6	AA24	NC	
6	IO_L39P_6	AE29	NC	
6	IO_L39N_6/VREF_6	AE30	NC	
6	IO_L40P_6	AB25	NC	
6	IO_L40N_6	AB26	NC	
6	IO_L41P_6	Y23	NC	
6	IO_L41N_6	Y24	NC	
6	IO_L42P_6	AD29	NC	
6	IO_L42N_6	AD30	NC	
6	IO_L43P_6	AC27		
6	IO_L43N_6	AC28		
6	IO_L44P_6	AA25		
6	IO_L44N_6	AA26		
6	IO_L45P_6	AC29		
6	IO_L45N_6/VREF_6	AB29		
6	IO_L46P_6	AB27		
6	IO_L46N_6	AB28		
6	IO_L47P_6	W23		
6	IO_L47N_6	W24		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
6	IO_L48P_6	AA27		
6	IO_L48N_6	AA28		
6	IO_L49P_6	Y26		
6	IO_L49N_6	Y27		
6	IO_L50P_6	W25		
6	IO_L50N_6	W26		
6	IO_L51P_6	AB30		
6	IO_L51N_6/VREF_6	AA30		
6	IO_L52P_6	W27		
6	IO_L52N_6	W28		
6	IO_L53P_6	V23		
6	IO_L53N_6	V24		
6	IO_L54P_6	AA29		
6	IO_L54N_6	Y29		
6	IO_L55P_6	V25		
6	IO_L55N_6	V26		
6	IO_L56P_6	U23		
6	IO_L56N_6	U24		
6	IO_L57P_6	Y30		
6	IO_L57N_6/VREF_6	W30		
6	IO_L58P_6	V27		
6	IO_L58N_6	V28		
6	IO_L59P_6	U22		
6	IO_L59N_6	T22		
6	IO_L60P_6	W29		
6	IO_L60N_6	V29		
6	IO_L85P_6	U26		
6	IO_L85N_6	U27		
6	IO_L86P_6	T23		
6	IO_L86N_6	T24		
6	IO_L87P_6	U28		
6	IO_L87N_6/VREF_6	U29		
6	IO_L88P_6	T27		
6	IO_L88N_6	T28		
6	IO_L89P_6	T25		
6	IO_L89N_6	T26		
6	IO_L90P_6	V30		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
6	IO_L90N_6	U30		
7	IO_L90P_7	R28		
7	IO_L90N_7	R27		
7	IO_L89P_7	R26		
7	IO_L89N_7	R25		
7	IO_L88P_7	T29		
7	IO_L88N_7/VREF_7	R29		
7	IO_L87P_7	P27		
7	IO_L87N_7	P26		
7	IO_L86P_7	R24		
7	IO_L86N_7	R23		
7	IO_L85P_7	P29		
7	IO_L85N_7	P28		
7	IO_L60P_7	N28		
7	IO_L60N_7	N27		
7	IO_L59P_7	P24		
7	IO_L59N_7	P23		
7	IO_L58P_7	P30		
7	IO_L58N_7/VREF_7	N30		
7	IO_L57P_7	M28		
7	IO_L57N_7	M27		
7	IO_L56P_7	R22		
7	IO_L56N_7	P22		
7	IO_L55P_7	N29		
7	IO_L55N_7	M29		
7	IO_L54P_7	L27		
7	IO_L54N_7	L26		
7	IO_L53P_7	N26		
7	IO_L53N_7	N25		
7	IO_L52P_7	M30		
7	IO_L52N_7/VREF_7	L30		
7	IO_L51P_7	K28		
7	IO_L51N_7	K27		
7	IO_L50P_7	N24		
7	IO_L50N_7	N23		
7	IO_L49P_7	L29		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
7	IO_L49N_7	K29		
7	IO_L48P_7	J28		
7	IO_L48N_7	J27		
7	IO_L47P_7	M26		
7	IO_L47N_7	M25		
7	IO_L46P_7	K30		
7	IO_L46N_7/VREF_7	J30		
7	IO_L45P_7	K26		
7	IO_L45N_7	K25		
7	IO_L44P_7	M24		
7	IO_L44N_7	M23		
7	IO_L43P_7	J29		
7	IO_L43N_7	H29		
7	IO_L42P_7	H28	NC	
7	IO_L42N_7	H27	NC	
7	IO_L41P_7	L24	NC	
7	IO_L41N_7	L23	NC	
7	IO_L40P_7	G30	NC	
7	IO_L40N_7/VREF_7	G29	NC	
7	IO_L39P_7	G28	NC	
7	IO_L39N_7	G27	NC	
7	IO_L38P_7	J26	NC	
7	IO_L38N_7	J25	NC	
7	IO_L37P_7	F30	NC	
7	IO_L37N_7	F29	NC	
7	IO_L36P_7	F28	NC	
7	IO_L36N_7	F27	NC	
7	IO_L35P_7	K24	NC	
7	IO_L35N_7	K23	NC	
7	IO_L34P_7	E30	NC	
7	IO_L34N_7/VREF_7	E29	NC	
7	IO_L33P_7	E28	NC	
7	IO_L33N_7	E27	NC	
7	IO_L32P_7	H26	NC	
7	IO_L32N_7	H25	NC	
7	IO_L31P_7	D30	NC	
7	IO_L31N_7	D29	NC	

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
7	IO_L06P_7	D28		
7	IO_L06N_7	C27		
7	IO_L05P_7	J24		
7	IO_L05N_7	J23		
7	IO_L04P_7	C30		
7	IO_L04N_7/VREF_7	C29		
7	IO_L03P_7	D26		
7	IO_L03N_7	C26		
7	IO_L02P_7	G26		
7	IO_L02N_7	G25		
7	IO_L01P_7/VRN_7	B28		
7	IO_L01N_7/VRP_7	A28		
0	VCCO_0	K21		
0	VCCO_0	K20		
0	VCCO_0	K19		
0	VCCO_0	K18		
0	VCCO_0	K17		
0	VCCO_0	K16		
0	VCCO_0	J21		
0	VCCO_0	J20		
0	VCCO_0	J19		
0	VCCO_0	J18		
1	VCCO_1	K15		
1	VCCO_1	K14		
1	VCCO_1	K13		
1	VCCO_1	K12		
1	VCCO_1	K11		
1	VCCO_1	K10		
1	VCCO_1	J13		
1	VCCO_1	J12		
1	VCCO_1	J11		
1	VCCO_1	J10		
2	VCCO_2	R10		
2	VCCO_2	P10		
2	VCCO_2	N10		
2	VCCO_2	N9		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
2	VCCO_2	M10		
2	VCCO_2	M9		
2	VCCO_2	L10		
2	VCCO_2	L9		
2	VCCO_2	K9		
2	VCCO_2	J9		
3	VCCO_3	AB9		
3	VCCO_3	AA9		
3	VCCO_3	Y10		
3	VCCO_3	Y9		
3	VCCO_3	W10		
3	VCCO_3	W9		
3	VCCO_3	V10		
3	VCCO_3	V9		
3	VCCO_3	U10		
3	VCCO_3	T10		
4	VCCO_4	AB13		
4	VCCO_4	AB12		
4	VCCO_4	AB11		
4	VCCO_4	AB10		
4	VCCO_4	AA15		
4	VCCO_4	AA14		
4	VCCO_4	AA13		
4	VCCO_4	AA12		
4	VCCO_4	AA11		
4	VCCO_4	AA10		
5	VCCO_5	AB21		
5	VCCO_5	AB20		
5	VCCO_5	AB19		
5	VCCO_5	AB18		
5	VCCO_5	AA21		
5	VCCO_5	AA20		
5	VCCO_5	AA19		
5	VCCO_5	AA18		
5	VCCO_5	AA17		
5	VCCO_5	AA16		
6	VCCO_6	AB22		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
6	VCCO_6	AA22		
6	VCCO_6	Y22		
6	VCCO_6	Y21		
6	VCCO_6	W22		
6	VCCO_6	W21		
6	VCCO_6	V22		
6	VCCO_6	V21		
6	VCCO_6	U21		
6	VCCO_6	T21		
7	VCCO_7	R21		
7	VCCO_7	P21		
7	VCCO_7	N22		
7	VCCO_7	N21		
7	VCCO_7	M22		
7	VCCO_7	M21		
7	VCCO_7	L22		
7	VCCO_7	L21		
7	VCCO_7	K22		
7	VCCO_7	J22		
N/A	CCLK	AC7		
N/A	PROG_B	G24		
N/A	DONE	AC8		
N/A	M0	AD24		
N/A	M1	AC24		
N/A	M2	AC23		
N/A	TCK	G7		
N/A	TDI	F26		
N/A	TDO	F5		
N/A	TMS	H8		
N/A	PWRDWN_B	AD7		
N/A	HSWAP_EN	H23		
N/A	RSVD	D6		
N/A	VBATT	H7		
N/A	DXP	H24		
N/A	DXN	D25		
N/A	AVCCAUXTX4	B26		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
N/A	VTTXPAD4	B27		
N/A	TXNPAD4	A27		
N/A	TXPPAD4	A26		
N/A	GND4	C25		
N/A	GND4	C25		
N/A	RXPPAD4	A25		
N/A	RXNPAD4	A24		
N/A	VTRXPAD4	B25		
N/A	AVCCAUXRX4	B24		
N/A	AVCCAUXTX6	B19		
N/A	VTTXPAD6	B20		
N/A	TXNPAD6	A20		
N/A	TXPPAD6	A19		
N/A	GND6	C19		
N/A	GND6	C19		
N/A	RXPPAD6	A18		
N/A	RXNPAD6	A17		
N/A	VTRXPAD6	B18		
N/A	AVCCAUXRX6	B17		
N/A	AVCCAUXTX7	B13		
N/A	VTTXPAD7	B14		
N/A	TXNPAD7	A14		
N/A	TXPPAD7	A13		
N/A	GND7	C12		
N/A	GND7	C12		
N/A	RXPPAD7	A12		
N/A	RXNPAD7	A11		
N/A	VTRXPAD7	B12		
N/A	AVCCAUXRX7	B11		
N/A	AVCCAUXTX9	B6		
N/A	VTTXPAD9	B7		
N/A	TXNPAD9	A7		
N/A	TXPPAD9	A6		
N/A	GND9	C6		
N/A	GND9	C6		
N/A	RXPPAD9	A5		
N/A	RXNPAD9	A4		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
N/A	VTRXPAD9	B5		
N/A	AVCCAUXRX9	B4		
N/A	AVCCAUXRX16	AJ4		
N/A	VTRXPAD16	AJ5		
N/A	RXNPAD16	AK4		
N/A	RXPPAD16	AK5		
N/A	GNDA16	AH6		
N/A	GNDA16	AH6		
N/A	TXPPAD16	AK6		
N/A	TXNPAD16	AK7		
N/A	VTTXPAD16	AJ7		
N/A	AVCCAUXTX16	AJ6		
N/A	AVCCAUXRX18	AJ11		
N/A	VTRXPAD18	AJ12		
N/A	RXNPAD18	AK11		
N/A	RXPPAD18	AK12		
N/A	GNDA18	AH12		
N/A	GNDA18	AH12		
N/A	TXPPAD18	AK13		
N/A	TXNPAD18	AK14		
N/A	VTTXPAD18	AJ14		
N/A	AVCCAUXTX18	AJ13		
N/A	AVCCAUXRX19	AJ17		
N/A	VTRXPAD19	AJ18		
N/A	RXNPAD19	AK17		
N/A	RXPPAD19	AK18		
N/A	GNDA19	AH19		
N/A	GNDA19	AH19		
N/A	TXPPAD19	AK19		
N/A	TXNPAD19	AK20		
N/A	VTTXPAD19	AJ20		
N/A	AVCCAUXTX19	AJ19		
N/A	AVCCAUXRX21	AJ24		
N/A	VTRXPAD21	AJ25		
N/A	RXNPAD21	AK24		
N/A	RXPPAD21	AK25		
N/A	GNDA21	AH25		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
N/A	GNDA21	AH25		
N/A	TXPPAD21	AK26		
N/A	TXNPAD21	AK27		
N/A	VTTXPAD21	AJ27		
N/A	AVCCAUXTX21	AJ26		
N/A	VCCAUX	AK29		
N/A	VCCAUX	AK16		
N/A	VCCAUX	AK15		
N/A	VCCAUX	AK2		
N/A	VCCAUX	AJ30		
N/A	VCCAUX	AJ1		
N/A	VCCAUX	T30		
N/A	VCCAUX	T1		
N/A	VCCAUX	R30		
N/A	VCCAUX	R1		
N/A	VCCAUX	B30		
N/A	VCCAUX	B1		
N/A	VCCAUX	A29		
N/A	VCCAUX	A16		
N/A	VCCAUX	A15		
N/A	VCCAUX	A2		
N/A	VCCINT	Y19		
N/A	VCCINT	Y18		
N/A	VCCINT	Y17		
N/A	VCCINT	Y16		
N/A	VCCINT	Y15		
N/A	VCCINT	Y14		
N/A	VCCINT	Y13		
N/A	VCCINT	Y12		
N/A	VCCINT	W20		
N/A	VCCINT	W11		
N/A	VCCINT	V20		
N/A	VCCINT	V11		
N/A	VCCINT	U20		
N/A	VCCINT	U11		
N/A	VCCINT	T20		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
N/A	VCCINT	T11		
N/A	VCCINT	R20		
N/A	VCCINT	R11		
N/A	VCCINT	P20		
N/A	VCCINT	P11		
N/A	VCCINT	N20		
N/A	VCCINT	N11		
N/A	VCCINT	M20		
N/A	VCCINT	M11		
N/A	VCCINT	L19		
N/A	VCCINT	L18		
N/A	VCCINT	L17		
N/A	VCCINT	L16		
N/A	VCCINT	L15		
N/A	VCCINT	L14		
N/A	VCCINT	L13		
N/A	VCCINT	L12		
N/A	GND	AK22		
N/A	GND	AK9		
N/A	GND	AJ29		
N/A	GND	AJ2		
N/A	GND	AH28		
N/A	GND	AH17		
N/A	GND	AH14		
N/A	GND	AH3		
N/A	GND	AG27		
N/A	GND	AG22		
N/A	GND	AG19		
N/A	GND	AG12		
N/A	GND	AG9		
N/A	GND	AG4		
N/A	GND	AF26		
N/A	GND	AF5		
N/A	GND	AE25		
N/A	GND	AE18		
N/A	GND	AE13		
N/A	GND	AE6		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
N/A	GND	AC30		
N/A	GND	AC1		
N/A	GND	Y28		
N/A	GND	Y25		
N/A	GND	Y20		
N/A	GND	Y11		
N/A	GND	Y6		
N/A	GND	Y3		
N/A	GND	W19		
N/A	GND	W18		
N/A	GND	W17		
N/A	GND	W16		
N/A	GND	W15		
N/A	GND	W14		
N/A	GND	W13		
N/A	GND	W12		
N/A	GND	V19		
N/A	GND	V18		
N/A	GND	V17		
N/A	GND	V16		
N/A	GND	V15		
N/A	GND	V14		
N/A	GND	V13		
N/A	GND	V12		
N/A	GND	U25		
N/A	GND	U19		
N/A	GND	U18		
N/A	GND	U17		
N/A	GND	U16		
N/A	GND	U15		
N/A	GND	U14		
N/A	GND	U13		
N/A	GND	U12		
N/A	GND	U6		
N/A	GND	T19		
N/A	GND	T18		
N/A	GND	T17		

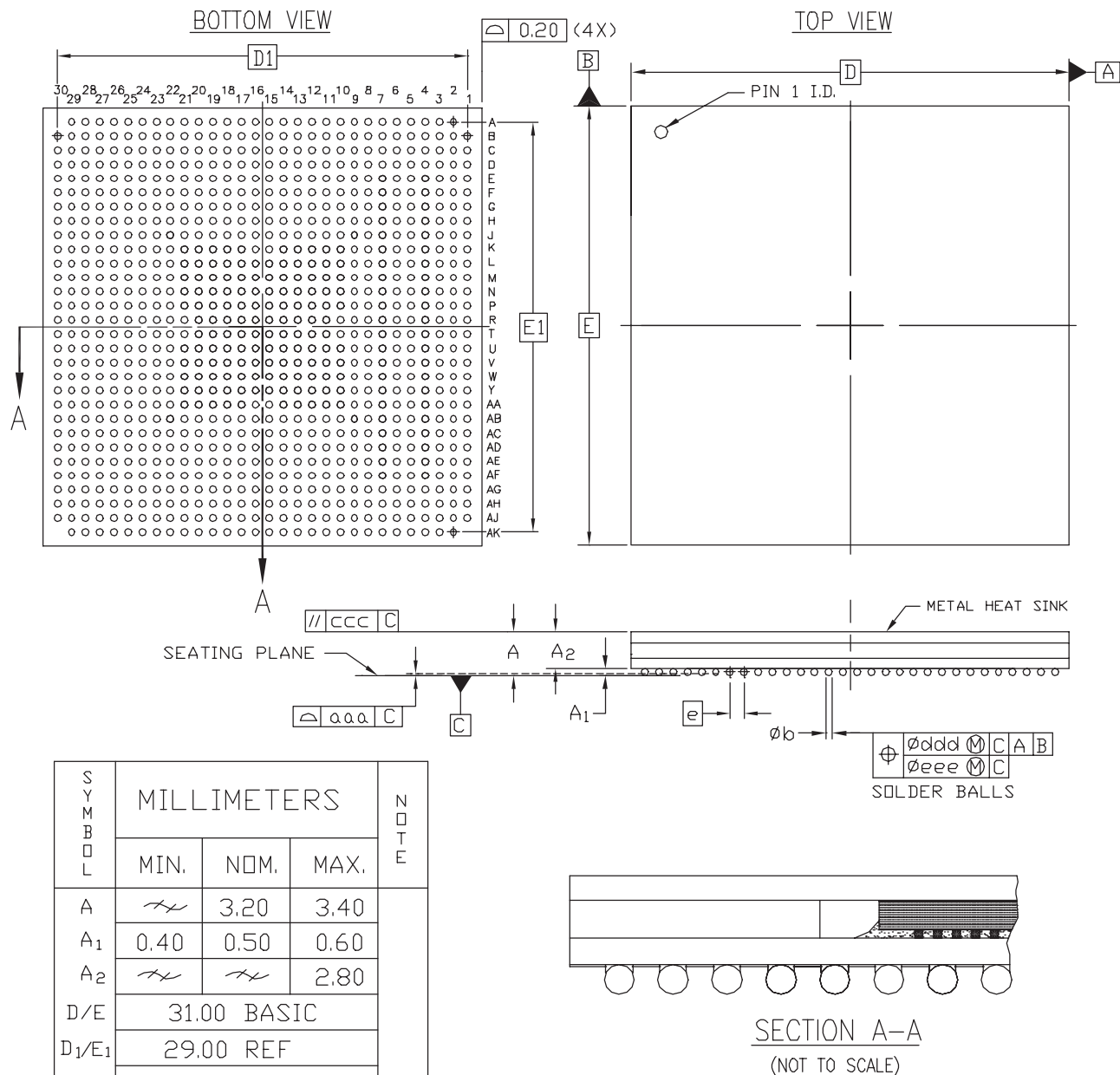
Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
N/A	GND	T16		
N/A	GND	T15		
N/A	GND	T14		
N/A	GND	T13		
N/A	GND	T12		
N/A	GND	R19		
N/A	GND	R18		
N/A	GND	R17		
N/A	GND	R16		
N/A	GND	R15		
N/A	GND	R14		
N/A	GND	R13		
N/A	GND	R12		
N/A	GND	P25		
N/A	GND	P19		
N/A	GND	P18		
N/A	GND	P17		
N/A	GND	P16		
N/A	GND	P15		
N/A	GND	P14		
N/A	GND	P13		
N/A	GND	P12		
N/A	GND	P6		
N/A	GND	N19		
N/A	GND	N18		
N/A	GND	N17		
N/A	GND	N16		
N/A	GND	N15		
N/A	GND	N14		
N/A	GND	N13		
N/A	GND	N12		
N/A	GND	M19		
N/A	GND	M18		
N/A	GND	M17		
N/A	GND	M16		
N/A	GND	M15		
N/A	GND	M14		

Table 9: FF896 — XC2VP7 and XC2VP20

Bank	Pin Description	Pin Number	No Connects	
			XC2V P7	XC2V P20
N/A	GND	M13		
N/A	GND	M12		
N/A	GND	L28		
N/A	GND	L25		
N/A	GND	L20		
N/A	GND	L11		
N/A	GND	L6		
N/A	GND	L3		
N/A	GND	H30		
N/A	GND	H1		
N/A	GND	F25		
N/A	GND	F18		
N/A	GND	F13		
N/A	GND	F6		
N/A	GND	E26		
N/A	GND	E5		
N/A	GND	D27		
N/A	GND	D22		
N/A	GND	D19		
N/A	GND	D12		
N/A	GND	D9		
N/A	GND	D4		
N/A	GND	C28		
N/A	GND	C17		
N/A	GND	C14		
N/A	GND	C3		
N/A	GND	B29		
N/A	GND	B2		
N/A	GND	A22		
N/A	GND	A9		

FF896 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)



NOTES:

1. ALL DIMENSIONS AND TOLERANCES CONFORM TO ANSI Y14.5M-1994
2. SYMBOL 'M' IS THE BALL MATRIX SIZE.
3. CONFORMS TO JEDEC MS-034-AAN-1 (DEPOPULATED)

Figure 4: FF896 Flip-Chip Fine-Pitch BGA Package Specifications

FF1152 Flip-Chip Fine-Pitch BGA Package

As shown in [Table 10](#), XC2VP20 and XC2VP50 Virtex-II Pro devices are available in the FF1152 flip-chip fine-pitch BGA package. Pins in each of these devices are the same, except for the differences shown in the No Connect column. Following this table are the **FF1152 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)**.

Table 10: **FF1152 — XC2VP20 and XC2VP50**

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
0	IO_L01N_0/VRP_0	E29		
0	IO_L01P_0/VRN_0	E28		
0	IO_L02N_0	H26		
0	IO_L02P_0	G26		
0	IO_L03N_0	H25		
0	IO_L03P_0/VREF_0	G25		
0	IO_L05_0/No_Pair	J25		
0	IO_L06N_0	K24		
0	IO_L06P_0	J24		
0	IO_L07N_0	F26		
0	IO_L07P_0	E26		
0	IO_L08N_0	D30		
0	IO_L08P_0	D29		
0	IO_L09N_0	K23		
0	IO_L09P_0/VREF_0	J23		
0	IO_L19N_0	F24	NC	
0	IO_L19P_0	E24	NC	
0	IO_L20N_0	D28	NC	
0	IO_L20P_0	C28	NC	
0	IO_L21N_0	H24	NC	
0	IO_L21P_0	G24	NC	
0	IO_L25N_0	G23	NC	
0	IO_L25P_0	F23	NC	
0	IO_L26N_0	E27	NC	
0	IO_L26P_0	D27	NC	
0	IO_L27N_0	K22	NC	
0	IO_L27P_0/VREF_0	J22	NC	
0	IO_L37N_0	H22		
0	IO_L37P_0	G22		
0	IO_L38N_0	D26		
0	IO_L38P_0	C26		
0	IO_L39N_0	K21		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
0	IO_L39P_0	J21		
0	IO_L43N_0	F22		
0	IO_L43P_0	E22		
0	IO_L44N_0	E25		
0	IO_L44P_0	D25		
0	IO_L45N_0	H21		
0	IO_L45P_0/VREF_0	G21		
0	IO_L46N_0	D22		
0	IO_L46P_0	D23		
0	IO_L47N_0	D24		
0	IO_L47P_0	C24		
0	IO_L48N_0	K20		
0	IO_L48P_0	J20		
0	IO_L49N_0	F21		
0	IO_L49P_0	E21		
0	IO_L50_0/No_Pair	C21		
0	IO_L53_0/No_Pair	C22		
0	IO_L54N_0	L19		
0	IO_L54P_0	K19		
0	IO_L55N_0	G20		
0	IO_L55P_0	F20		
0	IO_L56N_0	D21		
0	IO_L56P_0	D20		
0	IO_L57N_0	J19		
0	IO_L57P_0/VREF_0	H19		
0	IO_L67N_0	G19		
0	IO_L67P_0	F19		
0	IO_L68N_0	E19		
0	IO_L68P_0	D19		
0	IO_L69N_0	L18		
0	IO_L69P_0/VREF_0	K18		
0	IO_L73N_0	G18		
0	IO_L73P_0	F18		
0	IO_L74N_0/GCLK7P	E18		
0	IO_L74P_0/GCLK6S	D18		
0	IO_L75N_0/GCLK5P	J18		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
0	IO_L75P_0/GCLK4S	H18		
1	IO_L75N_1/GCLK3P	H17		
1	IO_L75P_1/GCLK2S	J17		
1	IO_L74N_1/GCLK1P	D17		
1	IO_L74P_1/GCLK0S	E17		
1	IO_L73N_1	F17		
1	IO_L73P_1	G17		
1	IO_L69N_1/VREF_1	K17		
1	IO_L69P_1	L17		
1	IO_L68N_1	D16		
1	IO_L68P_1	E16		
1	IO_L67N_1	F16		
1	IO_L67P_1	G16		
1	IO_L57N_1/VREF_1	H16		
1	IO_L57P_1	J16		
1	IO_L56N_1	D15		
1	IO_L56P_1	D14		
1	IO_L55N_1	F15		
1	IO_L55P_1	G15		
1	IO_L54N_1	K16		
1	IO_L54P_1	L16		
1	IO_L53_1/No_Pair	C13		
1	IO_L50_1/No_Pair	C14		
1	IO_L49N_1	E14		
1	IO_L49P_1	F14		
1	IO_L48N_1	J15		
1	IO_L48P_1	K15		
1	IO_L47N_1	C11		
1	IO_L47P_1	D11		
1	IO_L46N_1	D12		
1	IO_L46P_1	D13		
1	IO_L45N_1/VREF_1	G14		
1	IO_L45P_1	H14		
1	IO_L44N_1	D10		
1	IO_L44P_1	E10		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
1	IO_L43N_1	E13		
1	IO_L43P_1	F13		
1	IO_L39N_1	J14		
1	IO_L39P_1	K14		
1	IO_L38N_1	C9		
1	IO_L38P_1	D9		
1	IO_L37N_1	G13		
1	IO_L37P_1	H13		
1	IO_L27N_1/VREF_1	J13	NC	
1	IO_L27P_1	K13	NC	
1	IO_L26N_1	D8	NC	
1	IO_L26P_1	E8	NC	
1	IO_L25N_1	F12	NC	
1	IO_L25P_1	G12	NC	
1	IO_L21N_1	G11	NC	
1	IO_L21P_1	H11	NC	
1	IO_L20N_1	C7	NC	
1	IO_L20P_1	D7	NC	
1	IO_L19N_1	E11	NC	
1	IO_L19P_1	F11	NC	
1	IO_L09N_1/VREF_1	J12		
1	IO_L09P_1	K12		
1	IO_L08N_1	D6		
1	IO_L08P_1	D5		
1	IO_L07N_1	E9		
1	IO_L07P_1	F9		
1	IO_L06N_1	J11		
1	IO_L06P_1	K11		
1	IO_L05_1/No_Pair	J10		
1	IO_L03N_1/VREF_1	G10		
1	IO_L03P_1	H10		
1	IO_L02N_1	G9		
1	IO_L02P_1	H9		
1	IO_L01N_1/VRP_1	E7		
1	IO_L01P_1/VRN_1	E6		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
2	IO_L01N_2/VRP_2	D2		
2	IO_L01P_2/VRN_2	D1		
2	IO_L02N_2	F8		
2	IO_L02P_2	F7		
2	IO_L03N_2	E4		
2	IO_L03P_2	E3		
2	IO_L04N_2/VREF_2	E2		
2	IO_L04P_2	E1		
2	IO_L05N_2	J8		
2	IO_L05P_2	J7		
2	IO_L06N_2	F5		
2	IO_L06P_2	F4		
2	IO_L07N_2	G4	NC	
2	IO_L07P_2	G3	NC	
2	IO_L09N_2	G6	NC	
2	IO_L09P_2	G5	NC	
2	IO_L10N_2/VREF_2	F2	NC	
2	IO_L10P_2	F1	NC	
2	IO_L11N_2	L10	NC	
2	IO_L11P_2	L9	NC	
2	IO_L12N_2	H6	NC	
2	IO_L12P_2	H5	NC	
2	IO_L13N_2	G2	NC	
2	IO_L13P_2	G1	NC	
2	IO_L15N_2	J6	NC	
2	IO_L15P_2	J5	NC	
2	IO_L16N_2/VREF_2	J4	NC	
2	IO_L16P_2	J3	NC	
2	IO_L17N_2	K8	NC	
2	IO_L17P_2	K7	NC	
2	IO_L18N_2	H4	NC	
2	IO_L18P_2	H3	NC	
2	IO_L31N_2	H2		
2	IO_L31P_2	H1		
2	IO_L32N_2	M10		
2	IO_L32P_2	M9		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
2	IO_L33N_2	K5		
2	IO_L33P_2	K4		
2	IO_L34N_2/VREF_2	J2		
2	IO_L34P_2	K2		
2	IO_L35N_2	L8		
2	IO_L35P_2	L7		
2	IO_L36N_2	L6		
2	IO_L36P_2	L5		
2	IO_L37N_2	K1		
2	IO_L37P_2	L1		
2	IO_L38N_2	N10		
2	IO_L38P_2	N9		
2	IO_L39N_2	M7		
2	IO_L39P_2	M6		
2	IO_L40N_2/VREF_2	L2		
2	IO_L40P_2	M2		
2	IO_L41N_2	N8		
2	IO_L41P_2	N7		
2	IO_L42N_2	L4		
2	IO_L42P_2	L3		
2	IO_L43N_2	M4		
2	IO_L43P_2	M3		
2	IO_L44N_2	P10		
2	IO_L44P_2	P9		
2	IO_L45N_2	N6		
2	IO_L45P_2	N5		
2	IO_L46N_2/VREF_2	M1		
2	IO_L46P_2	N1		
2	IO_L47N_2	P8		
2	IO_L47P_2	P7		
2	IO_L48N_2	N4		
2	IO_L48P_2	N3		
2	IO_L49N_2	N2		
2	IO_L49P_2	P2		
2	IO_L50N_2	R10		
2	IO_L50P_2	R9		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
2	IO_L51N_2	P6		
2	IO_L51P_2	P5		
2	IO_L52N_2/VREF_2	P4		
2	IO_L52P_2	P3		
2	IO_L53N_2	T11		
2	IO_L53P_2	U11		
2	IO_L54N_2	R7		
2	IO_L54P_2	R6		
2	IO_L55N_2	P1		
2	IO_L55P_2	R1		
2	IO_L56N_2	T10		
2	IO_L56P_2	T9		
2	IO_L57N_2	R4		
2	IO_L57P_2	R3		
2	IO_L58N_2/VREF_2	R2		
2	IO_L58P_2	T2		
2	IO_L59N_2	T8		
2	IO_L59P_2	T7		
2	IO_L60N_2	T6		
2	IO_L60P_2	T5		
2	IO_L85N_2	T4		
2	IO_L85P_2	T3		
2	IO_L86N_2	U10		
2	IO_L86P_2	U9		
2	IO_L87N_2	U6		
2	IO_L87P_2	U5		
2	IO_L88N_2/VREF_2	U2		
2	IO_L88P_2	V2		
2	IO_L89N_2	U8		
2	IO_L89P_2	U7		
2	IO_L90N_2	U4		
2	IO_L90P_2	U3		
3	IO_L90N_3	V3		
3	IO_L90P_3	V4		
3	IO_L89N_3	V7		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
3	IO_L89P_3	V8		
3	IO_L88N_3	V5		
3	IO_L88P_3	V6		
3	IO_L87N_3/VREF_3	W2		
3	IO_L87P_3	Y2		
3	IO_L86N_3	V9		
3	IO_L86P_3	V10		
3	IO_L85N_3	W3		
3	IO_L85P_3	W4		
3	IO_L60N_3	Y1		
3	IO_L60P_3	AA1		
3	IO_L59N_3	V11		
3	IO_L59P_3	W11		
3	IO_L58N_3	W5		
3	IO_L58P_3	W6		
3	IO_L57N_3/VREF_3	Y3		
3	IO_L57P_3	Y4		
3	IO_L56N_3	W7		
3	IO_L56P_3	W8		
3	IO_L55N_3	Y6		
3	IO_L55P_3	Y7		
3	IO_L54N_3	AA2		
3	IO_L54P_3	AB2		
3	IO_L53N_3	W9		
3	IO_L53P_3	W10		
3	IO_L52N_3	AA3		
3	IO_L52P_3	AA4		
3	IO_L51N_3/VREF_3	AB1		
3	IO_L51P_3	AC1		
3	IO_L50N_3	Y9		
3	IO_L50P_3	Y10		
3	IO_L49N_3	AA5		
3	IO_L49P_3	AA6		
3	IO_L48N_3	AB3		
3	IO_L48P_3	AB4		
3	IO_L47N_3	AA7		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
3	IO_L47P_3	AA8		
3	IO_L46N_3	AB5		
3	IO_L46P_3	AB6		
3	IO_L45N_3/VREF_3	AC2		
3	IO_L45P_3	AD2		
3	IO_L44N_3	AA9		
3	IO_L44P_3	AA10		
3	IO_L43N_3	AC3		
3	IO_L43P_3	AC4		
3	IO_L42N_3	AD1		
3	IO_L42P_3	AE1		
3	IO_L41N_3	AB7		
3	IO_L41P_3	AB8		
3	IO_L40N_3	AC6		
3	IO_L40P_3	AC7		
3	IO_L39N_3/VREF_3	AD3		
3	IO_L39P_3	AD4		
3	IO_L38N_3	AB9		
3	IO_L38P_3	AB10		
3	IO_L37N_3	AD5		
3	IO_L37P_3	AD6		
3	IO_L36N_3	AE2		
3	IO_L36P_3	AF2		
3	IO_L35N_3	AD7		
3	IO_L35P_3	AD8		
3	IO_L34N_3	AE4		
3	IO_L34P_3	AE5		
3	IO_L33N_3/VREF_3	AG1		
3	IO_L33P_3	AG2		
3	IO_L32N_3	AC9		
3	IO_L32P_3	AC10		
3	IO_L31N_3	AF3		
3	IO_L31P_3	AF4		
3	IO_L18N_3	AH1	NC	
3	IO_L18P_3	AH2	NC	
3	IO_L17N_3	AE7	NC	

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
3	IO_L17P_3	AE8	NC	
3	IO_L16N_3	AF5	NC	
3	IO_L16P_3	AF6	NC	
3	IO_L15N_3/VREF_3	AG3	NC	
3	IO_L15P_3	AG4	NC	
3	IO_L14N_3	AD9	NC	
3	IO_L14P_3	AD10	NC	
3	IO_L13N_3	AH3	NC	
3	IO_L13P_3	AH4	NC	
3	IO_L12N_3	AJ1	NC	
3	IO_L12P_3	AJ2	NC	
3	IO_L11N_3	AF7	NC	
3	IO_L11P_3	AF8	NC	
3	IO_L09N_3/VREF_3	AK1	NC	
3	IO_L09P_3	AK2	NC	
3	IO_L07N_3	AG5	NC	
3	IO_L07P_3	AG6	NC	
3	IO_L06N_3	AL1		
3	IO_L06P_3	AL2		
3	IO_L05N_3	AG7		
3	IO_L05P_3	AH8		
3	IO_L04N_3	AH5		
3	IO_L04P_3	AH6		
3	IO_L03N_3/VREF_3	AK3		
3	IO_L03P_3	AK4		
3	IO_L02N_3	AJ7		
3	IO_L02P_3	AJ8		
3	IO_L01N_3/VRP_3	AJ4		
3	IO_L01P_3/VRN_3	AJ5		
4	IO_L01N_4/DOUT	AL5		
4	IO_L01P_4/INIT_B	AL6		
4	IO_L02N_4/D0	AG9		
4	IO_L02P_4/D1	AH9		
4	IO_L03N_4/D2	AK6		
4	IO_L03P_4/D3	AK7		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
4	IO_L05_4/No_Pair	AF10		
4	IO_L06N_4/VRP_4	AL7		
4	IO_L06P_4/VRN_4	AM7		
4	IO_L07N_4	AE11		
4	IO_L07P_4/VREF_4	AF11		
4	IO_L08N_4	AG10		
4	IO_L08P_4	AH10		
4	IO_L09N_4	AK8		
4	IO_L09P_4/VREF_4	AL8		
4	IO_L19N_4	AE12	NC	
4	IO_L19P_4	AF12	NC	
4	IO_L20N_4	AJ9	NC	
4	IO_L20P_4	AK9	NC	
4	IO_L21N_4	AL9	NC	
4	IO_L21P_4	AM9	NC	
4	IO_L25N_4	AG11	NC	
4	IO_L25P_4	AH11	NC	
4	IO_L26N_4	AH12	NC	
4	IO_L26P_4	AJ12	NC	
4	IO_L27N_4	AK10	NC	
4	IO_L27P_4/VREF_4	AL10	NC	
4	IO_L37N_4	AE13		
4	IO_L37P_4	AF13		
4	IO_L38N_4	AG13		
4	IO_L38P_4	AH13		
4	IO_L39N_4	AJ11		
4	IO_L39P_4	AK11		
4	IO_L43N_4	AE14		
4	IO_L43P_4	AF14		
4	IO_L44N_4	AJ13		
4	IO_L44P_4	AK13		
4	IO_L45N_4	AL11		
4	IO_L45P_4/VREF_4	AM11		
4	IO_L46N_4	AE15		
4	IO_L46P_4	AF15		
4	IO_L47N_4	AG14		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
4	IO_L47P_4	AH14		
4	IO_L48N_4	AL13		
4	IO_L48P_4	AL12		
4	IO_L49N_4	AD16		
4	IO_L49P_4	AE16		
4	IO_L50_4/No_Pair	AJ14		
4	IO_L53_4/No_Pair	AK14		
4	IO_L54N_4	AM14		
4	IO_L54P_4	AM13		
4	IO_L55N_4	AF16		
4	IO_L55P_4	AG16		
4	IO_L56N_4	AH15		
4	IO_L56P_4	AJ15		
4	IO_L57N_4	AL14		
4	IO_L57P_4/VREF_4	AL15		
4	IO_L67N_4	AD17		
4	IO_L67P_4	AE17		
4	IO_L68N_4	AH16		
4	IO_L68P_4	AJ16		
4	IO_L69N_4	AK16		
4	IO_L69P_4/VREF_4	AL16		
4	IO_L73N_4	AF17		
4	IO_L73P_4	AG17		
4	IO_L74N_4/GCLK3S	AH17		
4	IO_L74P_4/GCLK2P	AJ17		
4	IO_L75N_4/GCLK1S	AK17		
4	IO_L75P_4/GCLK0P	AL17		
5	IO_L75N_5/GCLK7S	AL18		
5	IO_L75P_5/GCLK6P	AK18		
5	IO_L74N_5/GCLK5S	AJ18		
5	IO_L74P_5/GCLK4P	AH18		
5	IO_L73N_5	AG18		
5	IO_L73P_5	AF18		
5	IO_L69N_5/VREF_5	AL19		
5	IO_L69P_5	AK19		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
5	IO_L68N_5	AJ19		
5	IO_L68P_5	AH19		
5	IO_L67N_5	AE18		
5	IO_L67P_5	AD18		
5	IO_L57N_5/VREF_5	AL20		
5	IO_L57P_5	AL21		
5	IO_L56N_5	AJ20		
5	IO_L56P_5	AH20		
5	IO_L55N_5	AG19		
5	IO_L55P_5	AF19		
5	IO_L54N_5	AM22		
5	IO_L54P_5	AM21		
5	IO_L53_5/No_Pair	AK21		
5	IO_L50_5/No_Pair	AJ21		
5	IO_L49N_5	AE19		
5	IO_L49P_5	AD19		
5	IO_L48N_5	AL23		
5	IO_L48P_5	AL22		
5	IO_L47N_5	AH21		
5	IO_L47P_5	AG21		
5	IO_L46N_5	AF20		
5	IO_L46P_5	AE20		
5	IO_L45N_5/VREF_5	AM24		
5	IO_L45P_5	AL24		
5	IO_L44N_5	AK22		
5	IO_L44P_5	AJ22		
5	IO_L43N_5	AF21		
5	IO_L43P_5	AE21		
5	IO_L39N_5	AK24		
5	IO_L39P_5	AJ24		
5	IO_L38N_5	AH22		
5	IO_L38P_5	AG22		
5	IO_L37N_5	AF22		
5	IO_L37P_5	AE22		
5	IO_L27N_5/VREF_5	AL25	NC	
5	IO_L27P_5	AK25	NC	

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
5	IO_L26N_5	AJ23	NC	
5	IO_L26P_5	AH23	NC	
5	IO_L25N_5	AH24	NC	
5	IO_L25P_5	AG24	NC	
5	IO_L21N_5	AM26	NC	
5	IO_L21P_5	AL26	NC	
5	IO_L20N_5	AK26	NC	
5	IO_L20P_5	AJ26	NC	
5	IO_L19N_5	AF23	NC	
5	IO_L19P_5	AE23	NC	
5	IO_L09N_5/VREF_5	AL27		
5	IO_L09P_5	AK27		
5	IO_L08N_5	AH25		
5	IO_L08P_5	AG25		
5	IO_L07N_5/VREF_5	AF24		
5	IO_L07P_5	AE24		
5	IO_L06N_5/VRP_5	AM28		
5	IO_L06P_5/VRN_5	AL28		
5	IO_L05_5/No_Pair	AF25		
5	IO_L03N_5/D4	AK28		
5	IO_L03P_5/D5	AK29		
5	IO_L02N_5/D6	AH26		
5	IO_L02P_5/D7	AG26		
5	IO_L01N_5/RDWR_B	AL29		
5	IO_L01P_5/CS_B	AL30		
6	IO_L01P_6/VRN_6	AJ30		
6	IO_L01N_6/VRP_6	AJ31		
6	IO_L02P_6	AJ27		
6	IO_L02N_6	AJ28		
6	IO_L03P_6	AK31		
6	IO_L03N_6/VREF_6	AK32		
6	IO_L04P_6	AH29		
6	IO_L04N_6	AH30		
6	IO_L05P_6	AH27		
6	IO_L05N_6	AG28		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
6	IO_L06P_6	AL33		
6	IO_L06N_6	AL34		
6	IO_L07P_6	AG29	NC	
6	IO_L07N_6	AG30	NC	
6	IO_L09P_6	AK33	NC	
6	IO_L09N_6/VREF_6	AK34	NC	
6	IO_L11P_6	AF27	NC	
6	IO_L11N_6	AF28	NC	
6	IO_L12P_6	AJ33	NC	
6	IO_L12N_6	AJ34	NC	
6	IO_L13P_6	AH31	NC	
6	IO_L13N_6	AH32	NC	
6	IO_L14P_6	AD25	NC	
6	IO_L14N_6	AD26	NC	
6	IO_L15P_6	AG31	NC	
6	IO_L15N_6/VREF_6	AG32	NC	
6	IO_L16P_6	AF29	NC	
6	IO_L16N_6	AF30	NC	
6	IO_L17P_6	AE27	NC	
6	IO_L17N_6	AE28	NC	
6	IO_L18P_6	AH33	NC	
6	IO_L18N_6	AH34	NC	
6	IO_L31P_6	AF31		
6	IO_L31N_6	AF32		
6	IO_L32P_6	AC25		
6	IO_L32N_6	AC26		
6	IO_L33P_6	AG33		
6	IO_L33N_6/VREF_6	AG34		
6	IO_L34P_6	AE30		
6	IO_L34N_6	AE31		
6	IO_L35P_6	AD27		
6	IO_L35N_6	AD28		
6	IO_L36P_6	AF33		
6	IO_L36N_6	AE33		
6	IO_L37P_6	AD29		
6	IO_L37N_6	AD30		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
6	IO_L38P_6	AB25		
6	IO_L38N_6	AB26		
6	IO_L39P_6	AD31		
6	IO_L39N_6/VREF_6	AD32		
6	IO_L40P_6	AC28		
6	IO_L40N_6	AC29		
6	IO_L41P_6	AB27		
6	IO_L41N_6	AB28		
6	IO_L42P_6	AE34		
6	IO_L42N_6	AD34		
6	IO_L43P_6	AC31		
6	IO_L43N_6	AC32		
6	IO_L44P_6	AA25		
6	IO_L44N_6	AA26		
6	IO_L45P_6	AD33		
6	IO_L45N_6/VREF_6	AC33		
6	IO_L46P_6	AB29		
6	IO_L46N_6	AB30		
6	IO_L47P_6	AA27		
6	IO_L47N_6	AA28		
6	IO_L48P_6	AB31		
6	IO_L48N_6	AB32		
6	IO_L49P_6	AA29		
6	IO_L49N_6	AA30		
6	IO_L50P_6	Y25		
6	IO_L50N_6	Y26		
6	IO_L51P_6	AC34		
6	IO_L51N_6/VREF_6	AB34		
6	IO_L52P_6	AA31		
6	IO_L52N_6	AA32		
6	IO_L53P_6	W25		
6	IO_L53N_6	W26		
6	IO_L54P_6	AB33		
6	IO_L54N_6	AA33		
6	IO_L55P_6	Y28		
6	IO_L55N_6	Y29		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
6	IO_L56P_6	W27		
6	IO_L56N_6	W28		
6	IO_L57P_6	Y31		
6	IO_L57N_6/VREF_6	Y32		
6	IO_L58P_6	W29		
6	IO_L58N_6	W30		
6	IO_L59P_6	W24		
6	IO_L59N_6	V24		
6	IO_L60P_6	AA34		
6	IO_L60N_6	Y34		
6	IO_L85P_6	W31		
6	IO_L85N_6	W32		
6	IO_L86P_6	V25		
6	IO_L86N_6	V26		
6	IO_L87P_6	Y33		
6	IO_L87N_6/VREF_6	W33		
6	IO_L88P_6	V29		
6	IO_L88N_6	V30		
6	IO_L89P_6	V27		
6	IO_L89N_6	V28		
6	IO_L90P_6	V31		
6	IO_L90N_6	V32		
7	IO_L90P_7	U32		
7	IO_L90N_7	U31		
7	IO_L89P_7	U28		
7	IO_L89N_7	U27		
7	IO_L88P_7	V33		
7	IO_L88N_7/VREF_7	U33		
7	IO_L87P_7	U30		
7	IO_L87N_7	U29		
7	IO_L86P_7	U26		
7	IO_L86N_7	U25		
7	IO_L85P_7	T32		
7	IO_L85N_7	T31		
7	IO_L60P_7	T30		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
7	IO_L60N_7	T29		
7	IO_L59P_7	T28		
7	IO_L59N_7	T27		
7	IO_L58P_7	T33		
7	IO_L58N_7/VREF_7	R33		
7	IO_L57P_7	R32		
7	IO_L57N_7	R31		
7	IO_L56P_7	T26		
7	IO_L56N_7	T25		
7	IO_L55P_7	R34		
7	IO_L55N_7	P34		
7	IO_L54P_7	R29		
7	IO_L54N_7	R28		
7	IO_L53P_7	U24		
7	IO_L53N_7	T24		
7	IO_L52P_7	P32		
7	IO_L52N_7/VREF_7	P31		
7	IO_L51P_7	P30		
7	IO_L51N_7	P29		
7	IO_L50P_7	R26		
7	IO_L50N_7	R25		
7	IO_L49P_7	P33		
7	IO_L49N_7	N33		
7	IO_L48P_7	N32		
7	IO_L48N_7	N31		
7	IO_L47P_7	P28		
7	IO_L47N_7	P27		
7	IO_L46P_7	N34		
7	IO_L46N_7/VREF_7	M34		
7	IO_L45P_7	N30		
7	IO_L45N_7	N29		
7	IO_L44P_7	P26		
7	IO_L44N_7	P25		
7	IO_L43P_7	M32		
7	IO_L43N_7	M31		
7	IO_L42P_7	L32		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
7	IO_L42N_7	L31		
7	IO_L41P_7	N28		
7	IO_L41N_7	N27		
7	IO_L40P_7	M33		
7	IO_L40N_7/VREF_7	L33		
7	IO_L39P_7	M29		
7	IO_L39N_7	M28		
7	IO_L38P_7	N26		
7	IO_L38N_7	N25		
7	IO_L37P_7	L34		
7	IO_L37N_7	K34		
7	IO_L36P_7	L30		
7	IO_L36N_7	L29		
7	IO_L35P_7	L28		
7	IO_L35N_7	L27		
7	IO_L34P_7	K33		
7	IO_L34N_7/VREF_7	J33		
7	IO_L33P_7	K31		
7	IO_L33N_7	K30		
7	IO_L32P_7	M26		
7	IO_L32N_7	M25		
7	IO_L31P_7	H34		
7	IO_L31N_7	H33		
7	IO_L18P_7	H32	NC	
7	IO_L18N_7	H31	NC	
7	IO_L17P_7	K28	NC	
7	IO_L17N_7	K27	NC	
7	IO_L16P_7	J32	NC	
7	IO_L16N_7/VREF_7	J31	NC	
7	IO_L15P_7	J30	NC	
7	IO_L15N_7	J29	NC	
7	IO_L13P_7	G34	NC	
7	IO_L13N_7	G33	NC	
7	IO_L12P_7	H30	NC	
7	IO_L12N_7	H29	NC	
7	IO_L11P_7	L26	NC	

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
7	IO_L11N_7	L25	NC	
7	IO_L10P_7	F34	NC	
7	IO_L10N_7/VREF_7	F33	NC	
7	IO_L09P_7	G30	NC	
7	IO_L09N_7	G29	NC	
7	IO_L07P_7	G32	NC	
7	IO_L07N_7	G31	NC	
7	IO_L06P_7	F31		
7	IO_L06N_7	F30		
7	IO_L05P_7	J28		
7	IO_L05N_7	J27		
7	IO_L04P_7	E34		
7	IO_L04N_7/VREF_7	E33		
7	IO_L03P_7	E32		
7	IO_L03N_7	E31		
7	IO_L02P_7	F28		
7	IO_L02N_7	F27		
7	IO_L01P_7/VRN_7	D34		
7	IO_L01N_7/VRP_7	D33		
0	VCCO_0	C29		
0	VCCO_0	E20		
0	VCCO_0	F25		
0	VCCO_0	L20		
0	VCCO_0	L21		
0	VCCO_0	L22		
0	VCCO_0	L23		
0	VCCO_0	M18		
0	VCCO_0	M19		
0	VCCO_0	M20		
0	VCCO_0	M21		
0	VCCO_0	M22		
1	VCCO_1	C6		
1	VCCO_1	E15		
1	VCCO_1	F10		
1	VCCO_1	L12		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
1	VCCO_1	L13		
1	VCCO_1	L14		
1	VCCO_1	L15		
1	VCCO_1	M13		
1	VCCO_1	M14		
1	VCCO_1	M15		
1	VCCO_1	M16		
1	VCCO_1	M17		
2	VCCO_2	F3		
2	VCCO_2	K6		
2	VCCO_2	M11		
2	VCCO_2	N11		
2	VCCO_2	N12		
2	VCCO_2	P11		
2	VCCO_2	P12		
2	VCCO_2	R5		
2	VCCO_2	R11		
2	VCCO_2	R12		
2	VCCO_2	T12		
2	VCCO_2	U12		
3	VCCO_3	V12		
3	VCCO_3	W12		
3	VCCO_3	Y5		
3	VCCO_3	Y11		
3	VCCO_3	Y12		
3	VCCO_3	AA11		
3	VCCO_3	AA12		
3	VCCO_3	AB11		
3	VCCO_3	AB12		
3	VCCO_3	AC11		
3	VCCO_3	AE6		
3	VCCO_3	AJ3		
4	VCCO_4	AC13		
4	VCCO_4	AC14		
4	VCCO_4	AC15		
4	VCCO_4	AC16		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
4	VCCO_4	AC17		
4	VCCO_4	AD12		
4	VCCO_4	AD13		
4	VCCO_4	AD14		
4	VCCO_4	AD15		
4	VCCO_4	AJ10		
4	VCCO_4	AK15		
4	VCCO_4	AM6		
5	VCCO_5	AC18		
5	VCCO_5	AC19		
5	VCCO_5	AC20		
5	VCCO_5	AC21		
5	VCCO_5	AC22		
5	VCCO_5	AD20		
5	VCCO_5	AD21		
5	VCCO_5	AD22		
5	VCCO_5	AD23		
5	VCCO_5	AJ25		
5	VCCO_5	AK20		
5	VCCO_5	AM29		
6	VCCO_6	V23		
6	VCCO_6	W23		
6	VCCO_6	Y23		
6	VCCO_6	Y24		
6	VCCO_6	Y30		
6	VCCO_6	AA23		
6	VCCO_6	AA24		
6	VCCO_6	AB23		
6	VCCO_6	AB24		
6	VCCO_6	AC24		
6	VCCO_6	AE29		
6	VCCO_6	AJ32		
7	VCCO_7	F32		
7	VCCO_7	K29		
7	VCCO_7	M24		
7	VCCO_7	N23		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
7	VCCO_7	N24		
7	VCCO_7	P23		
7	VCCO_7	P24		
7	VCCO_7	R23		
7	VCCO_7	R24		
7	VCCO_7	R30		
7	VCCO_7	T23		
7	VCCO_7	U23		
N/A	CCLK	AE9		
N/A	PROG_B	J26		
N/A	DONE	AE10		
N/A	M0	AF26		
N/A	M1	AE26		
N/A	M2	AE25		
N/A	TCK	J9		
N/A	TDI	H28		
N/A	TDO	H7		
N/A	TMS	K10		
N/A	PWRDWN_B	AF9		
N/A	HSWAP_EN	K25		
N/A	RSVD	G8		
N/A	VBATT	K9		
N/A	DXP	K26		
N/A	DXN	G27		
N/A	AVCCAUXTX2	B32	NC	
N/A	VTTXPAD2	B33	NC	
N/A	TXNPAD2	A33	NC	
N/A	TXPPAD2	A32	NC	
N/A	GND A2	C30	NC	
N/A	GND A2	C30	NC	
N/A	RXPPAD2	A31	NC	
N/A	RXNPAD2	A30	NC	
N/A	VTRXPAD2	B31	NC	
N/A	AVCCAUXRX2	B30	NC	
N/A	AVCCAUXTX4	B28		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	VTTXPAD4	B29		
N/A	TXNPAD4	A29		
N/A	TXPPAD4	A28		
N/A	GND4	C27		
N/A	GND4	C27		
N/A	RXPPAD4	A27		
N/A	RXNPAD4	A26		
N/A	VTRXPAD4	B27		
N/A	AVCCAUXXRX4	B26		
N/A	AVCCAUXTX5	B24	NC	
N/A	VTTXPAD5	B25	NC	
N/A	TXNPAD5	A25	NC	
N/A	TXPPAD5	A24	NC	
N/A	GND5	C23	NC	
N/A	GND5	C23	NC	
N/A	RXPPAD5	A23	NC	
N/A	RXNPAD5	A22	NC	
N/A	VTRXPAD5	B23	NC	
N/A	AVCCAUXXRX5	B22	NC	
N/A	AVCCAUXTX6	B20		
N/A	VTTXPAD6	B21		
N/A	TXNPAD6	A21		
N/A	TXPPAD6	A20		
N/A	GND6	C20		
N/A	GND6	C20		
N/A	RXPPAD6	A19		
N/A	RXNPAD6	A18		
N/A	VTRXPAD6	B19		
N/A	AVCCAUXXRX6	B18		
N/A	AVCCAUXTX7	B16		
N/A	VTTXPAD7	B17		
N/A	TXNPAD7	A17		
N/A	TXPPAD7	A16		
N/A	GND7	C15		
N/A	GND7	C15		
N/A	RXPPAD7	A15		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	RXNPAD7	A14		
N/A	VTRXPAD7	B15		
N/A	AVCCAUXRX7	B14		
N/A	AVCCAUXTX8	B12	NC	
N/A	VTTXPAD8	B13	NC	
N/A	TXNPAD8	A13	NC	
N/A	TXPPAD8	A12	NC	
N/A	GND A8	C12	NC	
N/A	GND A8	C12	NC	
N/A	RXPPAD8	A11	NC	
N/A	RXNPAD8	A10	NC	
N/A	VTRXPAD8	B11	NC	
N/A	AVCCAUXRX8	B10	NC	
N/A	AVCCAUXTX9	B8		
N/A	VTTXPAD9	B9		
N/A	TXNPAD9	A9		
N/A	TXPPAD9	A8		
N/A	GND A9	C8		
N/A	GND A9	C8		
N/A	RXPPAD9	A7		
N/A	RXNPAD9	A6		
N/A	VTRXPAD9	B7		
N/A	AVCCAUXRX9	B6		
N/A	AVCCAUXTX11	B4	NC	
N/A	VTTXPAD11	B5	NC	
N/A	TXNPAD11	A5	NC	
N/A	TXPPAD11	A4	NC	
N/A	GND A11	C5	NC	
N/A	GND A11	C5	NC	
N/A	RXPPAD11	A3	NC	
N/A	RXNPAD11	A2	NC	
N/A	VTRXPAD11	B3	NC	
N/A	AVCCAUXRX11	B2	NC	
N/A	AVCCAUXRX14	AN2	NC	
N/A	VTRXPAD14	AN3	NC	
N/A	RXNPAD14	AP2	NC	

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	RXPPAD14	AP3	NC	
N/A	GND A14	AM5	NC	
N/A	GND A14	AM5	NC	
N/A	TXPPAD14	AP4	NC	
N/A	TXNPAD14	AP5	NC	
N/A	VTTX PAD14	AN5	NC	
N/A	AVCCA UXTX14	AN4	NC	
N/A	AVCCA UXRX16	AN6		
N/A	VTRX PAD16	AN7		
N/A	RXNPAD16	AP6		
N/A	RXPPAD16	AP7		
N/A	GND A16	AM8		
N/A	GND A16	AM8		
N/A	TXPPAD16	AP8		
N/A	TXNPAD16	AP9		
N/A	VTTX PAD16	AN9		
N/A	AVCCA UXTX16	AN8		
N/A	AVCCA UXRX17	AN10	NC	
N/A	VTRX PAD17	AN11	NC	
N/A	RXNPAD17	AP10	NC	
N/A	RXPPAD17	AP11	NC	
N/A	GND A17	AM12	NC	
N/A	GND A17	AM12	NC	
N/A	TXPPAD17	AP12	NC	
N/A	TXNPAD17	AP13	NC	
N/A	VTTX PAD17	AN13	NC	
N/A	AVCCA UXTX17	AN12	NC	
N/A	AVCCA UXRX18	AN14		
N/A	VTRX PAD18	AN15		
N/A	RXNPAD18	AP14		
N/A	RXPPAD18	AP15		
N/A	GND A18	AM15		
N/A	GND A18	AM15		
N/A	TXPPAD18	AP16		
N/A	TXNPAD18	AP17		
N/A	VTTX PAD18	AN17		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	AVCCAUXTX18	AN16		
N/A	AVCCAUXRX19	AN18		
N/A	VTRXPAD19	AN19		
N/A	RXNPAD19	AP18		
N/A	RXPPAD19	AP19		
N/A	GND A19	AM20		
N/A	GND A19	AM20		
N/A	TXPPAD19	AP20		
N/A	TXNPAD19	AP21		
N/A	VTTXPAD19	AN21		
N/A	AVCCAUXTX19	AN20		
N/A	AVCCAUXRX20	AN22	NC	
N/A	VTRXPAD20	AN23	NC	
N/A	RXNPAD20	AP22	NC	
N/A	RXPPAD20	AP23	NC	
N/A	GND A20	AM23	NC	
N/A	GND A20	AM23	NC	
N/A	TXPPAD20	AP24	NC	
N/A	TXNPAD20	AP25	NC	
N/A	VTTXPAD20	AN25	NC	
N/A	AVCCAUXTX20	AN24	NC	
N/A	AVCCAUXRX21	AN26		
N/A	VTRXPAD21	AN27		
N/A	RXNPAD21	AP26		
N/A	RXPPAD21	AP27		
N/A	GND A21	AM27		
N/A	GND A21	AM27		
N/A	TXPPAD21	AP28		
N/A	TXNPAD21	AP29		
N/A	VTTXPAD21	AN29		
N/A	AVCCAUXTX21	AN28		
N/A	AVCCAUXRX23	AN30	NC	
N/A	VTRXPAD23	AN31	NC	
N/A	RXNPAD23	AP30	NC	
N/A	RXPPAD23	AP31	NC	
N/A	GND A23	AM30	NC	

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	GND A23	AM30	NC	
N/A	TXPPAD23	AP32	NC	
N/A	TXNPAD23	AP33	NC	
N/A	VTTXPAD23	AN33	NC	
N/A	AVCCAUXTX23	AN32	NC	
N/A	VCCINT	L11		
N/A	VCCINT	L24		
N/A	VCCINT	M12		
N/A	VCCINT	M23		
N/A	VCCINT	N13		
N/A	VCCINT	N14		
N/A	VCCINT	N15		
N/A	VCCINT	N16		
N/A	VCCINT	N17		
N/A	VCCINT	N18		
N/A	VCCINT	N19		
N/A	VCCINT	N20		
N/A	VCCINT	N21		
N/A	VCCINT	N22		
N/A	VCCINT	P13		
N/A	VCCINT	P22		
N/A	VCCINT	R13		
N/A	VCCINT	R22		
N/A	VCCINT	T13		
N/A	VCCINT	T22		
N/A	VCCINT	U13		
N/A	VCCINT	U22		
N/A	VCCINT	V13		
N/A	VCCINT	V22		
N/A	VCCINT	W13		
N/A	VCCINT	W22		
N/A	VCCINT	Y13		
N/A	VCCINT	Y22		
N/A	VCCINT	AA13		
N/A	VCCINT	AA22		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	VCCINT	AB13		
N/A	VCCINT	AB14		
N/A	VCCINT	AB15		
N/A	VCCINT	AB16		
N/A	VCCINT	AB17		
N/A	VCCINT	AB18		
N/A	VCCINT	AB19		
N/A	VCCINT	AB20		
N/A	VCCINT	AB21		
N/A	VCCINT	AB22		
N/A	VCCINT	AC12		
N/A	VCCINT	AC23		
N/A	VCCINT	AD11		
N/A	VCCINT	AD24		
N/A	VCCAUX	C3		
N/A	VCCAUX	C4		
N/A	VCCAUX	C17		
N/A	VCCAUX	C18		
N/A	VCCAUX	C31		
N/A	VCCAUX	C32		
N/A	VCCAUX	D3		
N/A	VCCAUX	D32		
N/A	VCCAUX	U1		
N/A	VCCAUX	V1		
N/A	VCCAUX	U34		
N/A	VCCAUX	V34		
N/A	VCCAUX	AL3		
N/A	VCCAUX	AL32		
N/A	VCCAUX	AM3		
N/A	VCCAUX	AM4		
N/A	VCCAUX	AM17		
N/A	VCCAUX	AM18		
N/A	VCCAUX	AM31		
N/A	VCCAUX	AM32		
N/A	GND	AF34		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	GND	B34		
N/A	GND	C1		
N/A	GND	C2		
N/A	GND	C10		
N/A	GND	C16		
N/A	GND	C19		
N/A	GND	C25		
N/A	GND	C33		
N/A	GND	C34		
N/A	GND	D4		
N/A	GND	D31		
N/A	GND	E5		
N/A	GND	E12		
N/A	GND	E23		
N/A	GND	E30		
N/A	GND	F6		
N/A	GND	F29		
N/A	GND	G7		
N/A	GND	G28		
N/A	GND	B1		
N/A	GND	H8		
N/A	GND	H12		
N/A	GND	H15		
N/A	GND	H20		
N/A	GND	J1		
N/A	GND	H27		
N/A	GND	AF1		
N/A	GND	K3		
N/A	GND	K32		
N/A	GND	M5		
N/A	GND	M8		
N/A	GND	M27		
N/A	GND	M30		
N/A	GND	P14		
N/A	GND	P15		
N/A	GND	P16		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	GND	P17		
N/A	GND	P18		
N/A	GND	P19		
N/A	GND	P20		
N/A	GND	P21		
N/A	GND	R8		
N/A	GND	R14		
N/A	GND	R15		
N/A	GND	R16		
N/A	GND	R17		
N/A	GND	R18		
N/A	GND	R19		
N/A	GND	R20		
N/A	GND	R21		
N/A	GND	R27		
N/A	GND	T1		
N/A	GND	T14		
N/A	GND	T15		
N/A	GND	T16		
N/A	GND	T17		
N/A	GND	T18		
N/A	GND	T19		
N/A	GND	T20		
N/A	GND	T21		
N/A	GND	T34		
N/A	GND	U14		
N/A	GND	U15		
N/A	GND	U16		
N/A	GND	U17		
N/A	GND	U18		
N/A	GND	U19		
N/A	GND	U20		
N/A	GND	U21		
N/A	GND	V14		
N/A	GND	V15		
N/A	GND	V16		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	GND	V17		
N/A	GND	V18		
N/A	GND	V19		
N/A	GND	V20		
N/A	GND	V21		
N/A	GND	W1		
N/A	GND	W14		
N/A	GND	W15		
N/A	GND	W16		
N/A	GND	W17		
N/A	GND	W18		
N/A	GND	W19		
N/A	GND	W20		
N/A	GND	W21		
N/A	GND	W34		
N/A	GND	Y8		
N/A	GND	Y14		
N/A	GND	Y15		
N/A	GND	Y16		
N/A	GND	Y17		
N/A	GND	Y18		
N/A	GND	Y19		
N/A	GND	Y20		
N/A	GND	Y21		
N/A	GND	Y27		
N/A	GND	AA14		
N/A	GND	AA15		
N/A	GND	AA16		
N/A	GND	AA17		
N/A	GND	AA18		
N/A	GND	AA19		
N/A	GND	AA20		
N/A	GND	AA21		
N/A	GND	AC5		
N/A	GND	AC8		
N/A	GND	AC27		

Table 10: FF1152 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	GND	AC30		
N/A	GND	AE3		
N/A	GND	AE32		
N/A	GND	H23		
N/A	GND	AG8		
N/A	GND	AG12		
N/A	GND	AG15		
N/A	GND	AG20		
N/A	GND	AG23		
N/A	GND	AG27		
N/A	GND	J34		
N/A	GND	AH7		
N/A	GND	AH28		
N/A	GND	AJ6		
N/A	GND	AJ29		
N/A	GND	AK5		
N/A	GND	AK12		
N/A	GND	AK23		
N/A	GND	AK30		
N/A	GND	AL4		
N/A	GND	AL31		
N/A	GND	AM1		
N/A	GND	AM2		
N/A	GND	AM10		
N/A	GND	AM16		
N/A	GND	AM19		
N/A	GND	AM25		
N/A	GND	AM33		
N/A	GND	AM34		
N/A	GND	AN1		
N/A	GND	AN34		

FF1152 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)

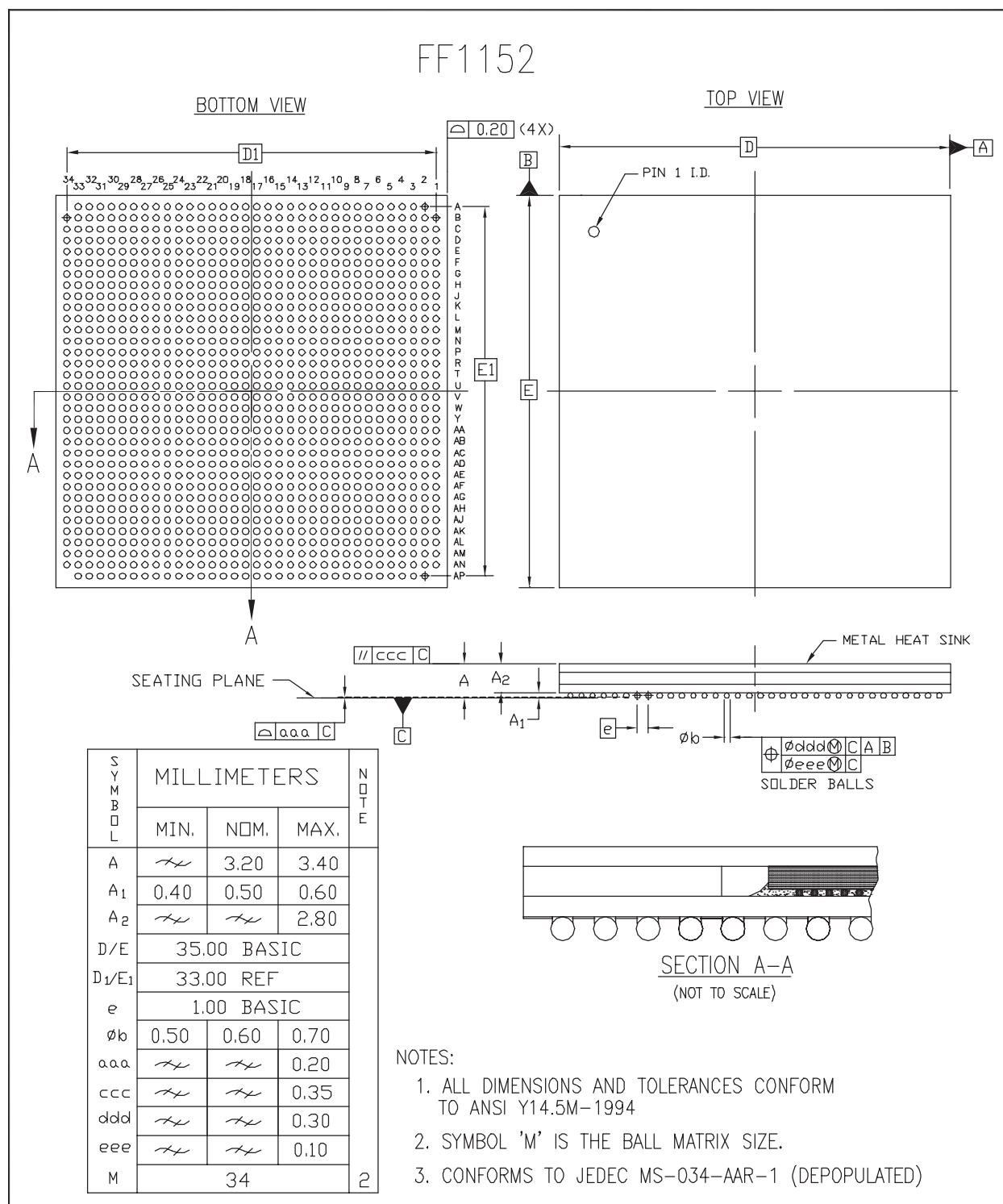


Figure 5: FF1152 Flip-Chip Fine-Pitch BGA Package Specifications

FF1517 Flip-Chip Fine-Pitch BGA Package

As shown in [Table 11](#), the XC2VP50 Virtex-II Pro device is available in the FF1517 flip-chip fine-pitch BGA package. Following this table are the **FF1517 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)**.

Table 11: **FF1517 — XC2VP50**

Bank	Pin Description	Pin Number
0	IO_L01N_0/VRP_0	F32
0	IO_L01P_0/VRN_0	E32
0	IO_L02N_0	K29
0	IO_L02P_0	J29
0	IO_L03N_0	K28
0	IO_L03P_0/VREF_0	K27
0	IO_L05_0/No_Pair	H30
0	IO_L06N_0	H29
0	IO_L06P_0	G29
0	IO_L07N_0	G31
0	IO_L07P_0	F31
0	IO_L08N_0	D32
0	IO_L08P_0	C32
0	IO_L09N_0	J28
0	IO_L09P_0/VREF_0	H28
0	IO_L19N_0	G30
0	IO_L19P_0	F30
0	IO_L20N_0	E31
0	IO_L20P_0	D31
0	IO_L21N_0	J27
0	IO_L21P_0	H27
0	IO_L25N_0	F29
0	IO_L25P_0	E29
0	IO_L26N_0	E30
0	IO_L26P_0	D30
0	IO_L27N_0	K26
0	IO_L27P_0/VREF_0	J26
0	IO_L37N_0	G28
0	IO_L37P_0	F28
0	IO_L38N_0	D29
0	IO_L38P_0	C29
0	IO_L39N_0	K25
0	IO_L39P_0	J25
0	IO_L43N_0	G27
0	IO_L43P_0	F27

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
0	IO_L44N_0	D28
0	IO_L44P_0	C28
0	IO_L45N_0	L24
0	IO_L45P_0/VREF_0	K24
0	IO_L46N_0	H26
0	IO_L46P_0	G26
0	IO_L47N_0	E27
0	IO_L47P_0	D27
0	IO_L48N_0	H25
0	IO_L48P_0	G25
0	IO_L49N_0	F25
0	IO_L49P_0	E25
0	IO_L50_0/No_Pair	E26
0	IO_L53_0/No_Pair	D26
0	IO_L54N_0	J24
0	IO_L54P_0	H24
0	IO_L55N_0	G24
0	IO_L55P_0	F24
0	IO_L56N_0	D25
0	IO_L56P_0	C25
0	IO_L57N_0	K23
0	IO_L57P_0/VREF_0	J23
0	IO_L58N_0	G23
0	IO_L58P_0	F23
0	IO_L59N_0	E24
0	IO_L59P_0	D24
0	IO_L60N_0	K22
0	IO_L60P_0	J22
0	IO_L64N_0	H22
0	IO_L64P_0	G22
0	IO_L65N_0	D23
0	IO_L65P_0	C23
0	IO_L66N_0	K21
0	IO_L66P_0/VREF_0	J21
0	IO_L67N_0	F22
0	IO_L67P_0	E22
0	IO_L68N_0	D22
0	IO_L68P_0	C22

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
0	IO_L69N_0	H21
0	IO_L69P_0/VREF_0	G21
0	IO_L73N_0	F21
0	IO_L73P_0	E21
0	IO_L74N_0/GCLK7P	D21
0	IO_L74P_0/GCLK6S	C21
0	IO_L75N_0/GCLK5P	F20
0	IO_L75P_0/GCLK4S	E20
1	IO_L75N_1/GCLK3P	H20
1	IO_L75P_1/GCLK2S	J20
1	IO_L74N_1/GCLK1P	C19
1	IO_L74P_1/GCLK0S	D19
1	IO_L73N_1	E19
1	IO_L73P_1	F19
1	IO_L69N_1/VREF_1	G19
1	IO_L69P_1	H19
1	IO_L68N_1	C18
1	IO_L68P_1	D18
1	IO_L67N_1	E18
1	IO_L67P_1	F18
1	IO_L66N_1/VREF_1	J19
1	IO_L66P_1	K19
1	IO_L65N_1	C17
1	IO_L65P_1	D17
1	IO_L64N_1	G18
1	IO_L64P_1	H18
1	IO_L60N_1	J18
1	IO_L60P_1	K18
1	IO_L59N_1	D16
1	IO_L59P_1	E16
1	IO_L58N_1	F17
1	IO_L58P_1	G17
1	IO_L57N_1/VREF_1	J17
1	IO_L57P_1	K17
1	IO_L56N_1	C15
1	IO_L56P_1	D15
1	IO_L55N_1	F16

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
1	IO_L55P_1	G16
1	IO_L54N_1	H16
1	IO_L54P_1	J16
1	IO_L53_1/No_Pair	D14
1	IO_L50_1/No_Pair	E14
1	IO_L49N_1	E15
1	IO_L49P_1	F15
1	IO_L48N_1	G15
1	IO_L48P_1	H15
1	IO_L47N_1	D13
1	IO_L47P_1	E13
1	IO_L46N_1	G14
1	IO_L46P_1	H14
1	IO_L45N_1/VREF_1	K16
1	IO_L45P_1	L16
1	IO_L44N_1	C12
1	IO_L44P_1	D12
1	IO_L43N_1	F13
1	IO_L43P_1	G13
1	IO_L39N_1	J15
1	IO_L39P_1	K15
1	IO_L38N_1	C11
1	IO_L38P_1	D11
1	IO_L37N_1	F12
1	IO_L37P_1	G12
1	IO_L27N_1/VREF_1	J14
1	IO_L27P_1	K14
1	IO_L26N_1	D10
1	IO_L26P_1	E10
1	IO_L25N_1	E11
1	IO_L25P_1	F11
1	IO_L21N_1	H13
1	IO_L21P_1	J13
1	IO_L20N_1	D9
1	IO_L20P_1	E9
1	IO_L19N_1	F10
1	IO_L19P_1	G10
1	IO_L09N_1/VREF_1	H12

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
1	IO_L09P_1	J12
1	IO_L08N_1	C8
1	IO_L08P_1	D8
1	IO_L07N_1	F9
1	IO_L07P_1	G9
1	IO_L06N_1	G11
1	IO_L06P_1	H11
1	IO_L05_1/No_Pair	H10
1	IO_L03N_1/VREF_1	K13
1	IO_L03P_1	K12
1	IO_L02N_1	J11
1	IO_L02P_1	K11
1	IO_L01N_1/VRP_1	E8
1	IO_L01P_1/VRN_1	F8
2	IO_L01N_2/VRP_2	E4
2	IO_L01P_2/VRN_2	E3
2	IO_L02N_2	G8
2	IO_L02P_2	H7
2	IO_L03N_2	C5
2	IO_L03P_2	D5
2	IO_L04N_2/VREF_2	D2
2	IO_L04P_2	D1
2	IO_L05N_2	J8
2	IO_L05P_2	J7
2	IO_L06N_2	E6
2	IO_L06P_2	F5
2	IO_L07N_2	E2
2	IO_L07P_2	E1
2	IO_L08N_2	K9
2	IO_L08P_2	K8
2	IO_L09N_2	G6
2	IO_L09P_2	G5
2	IO_L10N_2/VREF_2	G4
2	IO_L10P_2	G3
2	IO_L11N_2	N12
2	IO_L11P_2	N11
2	IO_L12N_2	F4

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
2	IO_L12P_2	F3
2	IO_L13N_2	F2
2	IO_L13P_2	F1
2	IO_L14N_2	L8
2	IO_L14P_2	L7
2	IO_L15N_2	H6
2	IO_L15P_2	H5
2	IO_L16N_2/VREF_2	H4
2	IO_L16P_2	H3
2	IO_L17N_2	P12
2	IO_L17P_2	P11
2	IO_L18N_2	J6
2	IO_L18P_2	J5
2	IO_L19N_2	G2
2	IO_L19P_2	G1
2	IO_L20N_2	N10
2	IO_L20P_2	N9
2	IO_L21N_2	K7
2	IO_L21P_2	K6
2	IO_L22N_2/VREF_2	H2
2	IO_L22P_2	H1
2	IO_L23N_2	R12
2	IO_L23P_2	R11
2	IO_L24N_2	J4
2	IO_L24P_2	J3
2	IO_L25N_2	J2
2	IO_L25P_2	J1
2	IO_L26N_2	P10
2	IO_L26P_2	P9
2	IO_L27N_2	K5
2	IO_L27P_2	K4
2	IO_L28N_2/VREF_2	K2
2	IO_L28P_2	K1
2	IO_L29N_2	P8
2	IO_L29P_2	P7
2	IO_L30N_2	L6
2	IO_L30P_2	L5
2	IO_L31N_2	L3

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
2	IO_L31P_2	L2
2	IO_L32N_2	R10
2	IO_L32P_2	R9
2	IO_L33N_2	N8
2	IO_L33P_2	N7
2	IO_L34N_2/VREF_2	M4
2	IO_L34P_2	M3
2	IO_L35N_2	T12
2	IO_L35P_2	T11
2	IO_L36N_2	M6
2	IO_L36P_2	M5
2	IO_L37N_2	M2
2	IO_L37P_2	M1
2	IO_L38N_2	T10
2	IO_L38P_2	T9
2	IO_L39N_2	N6
2	IO_L39P_2	N5
2	IO_L40N_2/VREF_2	N4
2	IO_L40P_2	N3
2	IO_L41N_2	U12
2	IO_L41P_2	U11
2	IO_L42N_2	P5
2	IO_L42P_2	P4
2	IO_L43N_2	N2
2	IO_L43P_2	N1
2	IO_L44N_2	T8
2	IO_L44P_2	T7
2	IO_L45N_2	R7
2	IO_L45P_2	R6
2	IO_L46N_2/VREF_2	P2
2	IO_L46P_2	P1
2	IO_L47N_2	U10
2	IO_L47P_2	U9
2	IO_L48N_2	R5
2	IO_L48P_2	R4
2	IO_L49N_2	R3
2	IO_L49P_2	R2
2	IO_L50N_2	V12

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
2	IO_L50P_2	V11
2	IO_L51N_2	T6
2	IO_L51P_2	T5
2	IO_L52N_2/VREF_2	T2
2	IO_L52P_2	T1
2	IO_L53N_2	V10
2	IO_L53P_2	V9
2	IO_L54N_2	T4
2	IO_L54P_2	T3
2	IO_L55N_2	U4
2	IO_L55P_2	U3
2	IO_L56N_2	V8
2	IO_L56P_2	V7
2	IO_L57N_2	U7
2	IO_L57P_2	U6
2	IO_L58N_2/VREF_2	U2
2	IO_L58P_2	U1
2	IO_L59N_2	W12
2	IO_L59P_2	W11
2	IO_L60N_2	V6
2	IO_L60P_2	V5
2	IO_L85N_2	V4
2	IO_L85P_2	V3
2	IO_L86N_2	W10
2	IO_L86P_2	W9
2	IO_L87N_2	W6
2	IO_L87P_2	W5
2	IO_L88N_2/VREF_2	V2
2	IO_L88P_2	V1
2	IO_L89N_2	W8
2	IO_L89P_2	W7
2	IO_L90N_2	W4
2	IO_L90P_2	W3
3	IO_L90N_3	AA3
3	IO_L90P_3	AA4
3	IO_L89N_3	AA7
3	IO_L89P_3	AA8

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
3	IO_L88N_3	AA5
3	IO_L88P_3	AA6
3	IO_L87N_3/VREF_3	AB1
3	IO_L87P_3	AB2
3	IO_L86N_3	AA9
3	IO_L86P_3	AA10
3	IO_L85N_3	AB3
3	IO_L85P_3	AB4
3	IO_L60N_3	AC1
3	IO_L60P_3	AC2
3	IO_L59N_3	AA11
3	IO_L59P_3	AA12
3	IO_L58N_3	AB5
3	IO_L58P_3	AB6
3	IO_L57N_3/VREF_3	AC3
3	IO_L57P_3	AC4
3	IO_L56N_3	AB7
3	IO_L56P_3	AB8
3	IO_L55N_3	AC6
3	IO_L55P_3	AC7
3	IO_L54N_3	AD1
3	IO_L54P_3	AD2
3	IO_L53N_3	AB9
3	IO_L53P_3	AB10
3	IO_L52N_3	AD5
3	IO_L52P_3	AD6
3	IO_L51N_3/VREF_3	AD3
3	IO_L51P_3	AD4
3	IO_L50N_3	AB11
3	IO_L50P_3	AB12
3	IO_L49N_3	AE4
3	IO_L49P_3	AE5
3	IO_L48N_3	AE2
3	IO_L48P_3	AE3
3	IO_L47N_3	AC9
3	IO_L47P_3	AC10
3	IO_L46N_3	AE6
3	IO_L46P_3	AE7

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
3	IO_L45N_3/VREF_3	AF1
3	IO_L45P_3	AF2
3	IO_L44N_3	AD7
3	IO_L44P_3	AD8
3	IO_L43N_3	AF4
3	IO_L43P_3	AF5
3	IO_L42N_3	AG1
3	IO_L42P_3	AG2
3	IO_L41N_3	AC11
3	IO_L41P_3	AC12
3	IO_L40N_3	AG3
3	IO_L40P_3	AG4
3	IO_L39N_3/VREF_3	AH1
3	IO_L39P_3	AH2
3	IO_L38N_3	AD9
3	IO_L38P_3	AD10
3	IO_L37N_3	AF7
3	IO_L37P_3	AF8
3	IO_L36N_3	AH3
3	IO_L36P_3	AH4
3	IO_L35N_3	AD11
3	IO_L35P_3	AD12
3	IO_L34N_3	AG5
3	IO_L34P_3	AG6
3	IO_L33N_3/VREF_3	AJ2
3	IO_L33P_3	AJ3
3	IO_L32N_3	AE9
3	IO_L32P_3	AE10
3	IO_L31N_3	AH5
3	IO_L31P_3	AH6
3	IO_L30N_3	AK1
3	IO_L30P_3	AK2
3	IO_L29N_3	AG7
3	IO_L29P_3	AG8
3	IO_L28N_3	AJ5
3	IO_L28P_3	AJ6
3	IO_L27N_3/VREF_3	AL1
3	IO_L27P_3	AL2

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
3	IO_L26N_3	AF9
3	IO_L26P_3	AF10
3	IO_L25N_3	AK4
3	IO_L25P_3	AK5
3	IO_L24N_3	AM1
3	IO_L24P_3	AM2
3	IO_L23N_3	AE11
3	IO_L23P_3	AE12
3	IO_L22N_3	AM3
3	IO_L22P_3	AM4
3	IO_L21N_3/VREF_3	AL3
3	IO_L21P_3	AL4
3	IO_L20N_3	AG9
3	IO_L20P_3	AG10
3	IO_L19N_3	AK6
3	IO_L19P_3	AK7
3	IO_L18N_3	AN1
3	IO_L18P_3	AN2
3	IO_L17N_3	AF11
3	IO_L17P_3	AF12
3	IO_L16N_3	AL5
3	IO_L16P_3	AL6
3	IO_L15N_3/VREF_3	AP1
3	IO_L15P_3	AP2
3	IO_L14N_3	AJ7
3	IO_L14P_3	AJ8
3	IO_L13N_3	AM5
3	IO_L13P_3	AM6
3	IO_L12N_3	AN3
3	IO_L12P_3	AN4
3	IO_L11N_3	AG11
3	IO_L11P_3	AG12
3	IO_L10N_3	AN5
3	IO_L10P_3	AN6
3	IO_L09N_3/VREF_3	AR1
3	IO_L09P_3	AR2
3	IO_L08N_3	AK8
3	IO_L08P_3	AK9

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
3	IO_L07N_3	AR3
3	IO_L07P_3	AR4
3	IO_L06N_3	AP3
3	IO_L06P_3	AP4
3	IO_L05N_3	AL7
3	IO_L05P_3	AL8
3	IO_L04N_3	AP5
3	IO_L04P_3	AR6
3	IO_L03N_3/VREF_3	AT1
3	IO_L03P_3	AT2
3	IO_L02N_3	AM7
3	IO_L02P_3	AN8
3	IO_L01N_3/VRP_3	AT5
3	IO_L01P_3/VRN_3	AU5
4	IO_L01N_4/DOUT	AP7
4	IO_L01P_4/INIT_B	AR7
4	IO_L02N_4/D0	AP8
4	IO_L02P_4/D1	AR8
4	IO_L03N_4/D2	AT8
4	IO_L03P_4/D3	AU8
4	IO_L05_4/No_Pair	AM10
4	IO_L06N_4/VRP_4	AR9
4	IO_L06P_4/VRN_4	AT9
4	IO_L07N_4	AK11
4	IO_L07P_4/VREF_4	AL11
4	IO_L08N_4	AN9
4	IO_L08P_4	AP9
4	IO_L09N_4	AR10
4	IO_L09P_4/VREF_4	AT10
4	IO_L19N_4	AK12
4	IO_L19P_4	AK13
4	IO_L20N_4	AN10
4	IO_L20P_4	AP10
4	IO_L21N_4	AP11
4	IO_L21P_4	AR11
4	IO_L25N_4	AL12
4	IO_L25P_4	AM12

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
4	IO_L26N_4	AM11
4	IO_L26P_4	AN11
4	IO_L27N_4	AT11
4	IO_L27P_4/VREF_4	AU11
4	IO_L37N_4	AL13
4	IO_L37P_4	AM13
4	IO_L38N_4	AN12
4	IO_L38P_4	AP12
4	IO_L39N_4	AT12
4	IO_L39P_4	AU12
4	IO_L43N_4	AK14
4	IO_L43P_4	AL14
4	IO_L44N_4	AN13
4	IO_L44P_4	AP13
4	IO_L45N_4	AR13
4	IO_L45P_4/VREF_4	AT13
4	IO_L46N_4	AK15
4	IO_L46P_4	AL15
4	IO_L47N_4	AM14
4	IO_L47P_4	AN14
4	IO_L48N_4	AR14
4	IO_L48P_4	AT14
4	IO_L49N_4	AJ16
4	IO_L49P_4	AK16
4	IO_L50_4/No_Pair	AP15
4	IO_L53_4/No_Pair	AR15
4	IO_L54N_4	AT15
4	IO_L54P_4	AU15
4	IO_L55N_4	AM15
4	IO_L55P_4	AN15
4	IO_L56N_4	AN16
4	IO_L56P_4	AP16
4	IO_L57N_4	AR16
4	IO_L57P_4/VREF_4	AT16
4	IO_L58N_4	AL16
4	IO_L58P_4	AM16
4	IO_L59N_4	AN17
4	IO_L59P_4	AP17

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
4	IO_L60N_4	AT17
4	IO_L60P_4	AU17
4	IO_L64N_4	AK17
4	IO_L64P_4	AL17
4	IO_L65N_4	AM18
4	IO_L65P_4	AN18
4	IO_L66N_4	AP18
4	IO_L66P_4/VREF_4	AR18
4	IO_L67N_4	AK18
4	IO_L67P_4	AL18
4	IO_L68N_4	AM19
4	IO_L68P_4	AN19
4	IO_L69N_4	AT18
4	IO_L69P_4/VREF_4	AU18
4	IO_L73N_4	AK19
4	IO_L73P_4	AL19
4	IO_L74N_4/GCLK3S	AP19
4	IO_L74P_4/GCLK2P	AR19
4	IO_L75N_4/GCLK1S	AT19
4	IO_L75P_4/GCLK0P	AU19
5	IO_L75N_5/GCLK7S	AU21
5	IO_L75P_5/GCLK6P	AT21
5	IO_L74N_5/GCLK5S	AR21
5	IO_L74P_5/GCLK4P	AP21
5	IO_L73N_5	AL21
5	IO_L73P_5	AK21
5	IO_L69N_5/VREF_5	AU22
5	IO_L69P_5	AT22
5	IO_L68N_5	AN21
5	IO_L68P_5	AM21
5	IO_L67N_5	AL22
5	IO_L67P_5	AK22
5	IO_L66N_5/VREF_5	AR22
5	IO_L66P_5	AP22
5	IO_L65N_5	AN22
5	IO_L65P_5	AM22
5	IO_L64N_5	AL23

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
5	IO_L64P_5	AK23
5	IO_L60N_5	AU23
5	IO_L60P_5	AT23
5	IO_L59N_5	AP23
5	IO_L59P_5	AN23
5	IO_L58N_5	AM24
5	IO_L58P_5	AL24
5	IO_L57N_5/VREF_5	AT24
5	IO_L57P_5	AR24
5	IO_L56N_5	AP24
5	IO_L56P_5	AN24
5	IO_L55N_5	AN25
5	IO_L55P_5	AM25
5	IO_L54N_5	AU25
5	IO_L54P_5	AT25
5	IO_L53_5/No_Pair	AR25
5	IO_L50_5/No_Pair	AP25
5	IO_L49N_5	AK24
5	IO_L49P_5	AJ24
5	IO_L48N_5	AT26
5	IO_L48P_5	AR26
5	IO_L47N_5	AN26
5	IO_L47P_5	AM26
5	IO_L46N_5	AL25
5	IO_L46P_5	AK25
5	IO_L45N_5/VREF_5	AT27
5	IO_L45P_5	AR27
5	IO_L44N_5	AP27
5	IO_L44P_5	AN27
5	IO_L43N_5	AL26
5	IO_L43P_5	AK26
5	IO_L39N_5	AU28
5	IO_L39P_5	AT28
5	IO_L38N_5	AP28
5	IO_L38P_5	AN28
5	IO_L37N_5	AM27
5	IO_L37P_5	AL27
5	IO_L27N_5/VREF_5	AU29

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
5	IO_L27P_5	AT29
5	IO_L26N_5	AN29
5	IO_L26P_5	AM29
5	IO_L25N_5	AM28
5	IO_L25P_5	AL28
5	IO_L21N_5	AR29
5	IO_L21P_5	AP29
5	IO_L20N_5	AP30
5	IO_L20P_5	AN30
5	IO_L19N_5	AK27
5	IO_L19P_5	AK28
5	IO_L09N_5/VREF_5	AT30
5	IO_L09P_5	AR30
5	IO_L08N_5	AP31
5	IO_L08P_5	AN31
5	IO_L07N_5/VREF_5	AL29
5	IO_L07P_5	AK29
5	IO_L06N_5/VRP_5	AT31
5	IO_L06P_5/VRN_5	AR31
5	IO_L05_5/No_Pair	AM30
5	IO_L03N_5/D4	AU32
5	IO_L03P_5/D5	AT32
5	IO_L02N_5/D6	AR32
5	IO_L02P_5/D7	AP32
5	IO_L01N_5/RDWR_B	AR33
5	IO_L01P_5/CS_B	AP33
6	IO_L01P_6/VRN_6	AU35
6	IO_L01N_6/VRP_6	AT35
6	IO_L02P_6	AN32
6	IO_L02N_6	AM33
6	IO_L03P_6	AT38
6	IO_L03N_6/VREF_6	AT39
6	IO_L04P_6	AR34
6	IO_L04N_6	AP35
6	IO_L05P_6	AL32
6	IO_L05N_6	AL33
6	IO_L06P_6	AP36

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
6	IO_L06N_6	AP37
6	IO_L07P_6	AR36
6	IO_L07N_6	AR37
6	IO_L08P_6	AK31
6	IO_L08N_6	AK32
6	IO_L09P_6	AR38
6	IO_L09N_6/VREF_6	AR39
6	IO_L10P_6	AN34
6	IO_L10N_6	AN35
6	IO_L11P_6	AG28
6	IO_L11N_6	AG29
6	IO_L12P_6	AN36
6	IO_L12N_6	AN37
6	IO_L13P_6	AM34
6	IO_L13N_6	AM35
6	IO_L14P_6	AJ32
6	IO_L14N_6	AJ33
6	IO_L15P_6	AP38
6	IO_L15N_6/VREF_6	AP39
6	IO_L16P_6	AL34
6	IO_L16N_6	AL35
6	IO_L17P_6	AF28
6	IO_L17N_6	AF29
6	IO_L18P_6	AN38
6	IO_L18N_6	AN39
6	IO_L19P_6	AK33
6	IO_L19N_6	AK34
6	IO_L20P_6	AG30
6	IO_L20N_6	AG31
6	IO_L21P_6	AL36
6	IO_L21N_6/VREF_6	AL37
6	IO_L22P_6	AM36
6	IO_L22N_6	AM37
6	IO_L23P_6	AE28
6	IO_L23N_6	AE29
6	IO_L24P_6	AM38
6	IO_L24N_6	AM39
6	IO_L25P_6	AK35

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
6	IO_L25N_6	AK36
6	IO_L26P_6	AF30
6	IO_L26N_6	AF31
6	IO_L27P_6	AL38
6	IO_L27N_6/VREF_6	AL39
6	IO_L28P_6	AJ34
6	IO_L28N_6	AJ35
6	IO_L29P_6	AG32
6	IO_L29N_6	AG33
6	IO_L30P_6	AK38
6	IO_L30N_6	AK39
6	IO_L31P_6	AH34
6	IO_L31N_6	AH35
6	IO_L32P_6	AE30
6	IO_L32N_6	AE31
6	IO_L33P_6	AJ37
6	IO_L33N_6/VREF_6	AJ38
6	IO_L34P_6	AG34
6	IO_L34N_6	AG35
6	IO_L35P_6	AD28
6	IO_L35N_6	AD29
6	IO_L36P_6	AH36
6	IO_L36N_6	AH37
6	IO_L37P_6	AF32
6	IO_L37N_6	AF33
6	IO_L38P_6	AD30
6	IO_L38N_6	AD31
6	IO_L39P_6	AH38
6	IO_L39N_6/VREF_6	AH39
6	IO_L40P_6	AG36
6	IO_L40N_6	AG37
6	IO_L41P_6	AC28
6	IO_L41N_6	AC29
6	IO_L42P_6	AG38
6	IO_L42N_6	AG39
6	IO_L43P_6	AF35
6	IO_L43N_6	AF36
6	IO_L44P_6	AD32

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
6	IO_L44N_6	AD33
6	IO_L45P_6	AF38
6	IO_L45N_6/VREF_6	AF39
6	IO_L46P_6	AE33
6	IO_L46N_6	AE34
6	IO_L47P_6	AC30
6	IO_L47N_6	AC31
6	IO_L48P_6	AE37
6	IO_L48N_6	AE38
6	IO_L49P_6	AE35
6	IO_L49N_6	AE36
6	IO_L50P_6	AB28
6	IO_L50N_6	AB29
6	IO_L51P_6	AD36
6	IO_L51N_6/VREF_6	AD37
6	IO_L52P_6	AD34
6	IO_L52N_6	AD35
6	IO_L53P_6	AB30
6	IO_L53N_6	AB31
6	IO_L54P_6	AD38
6	IO_L54N_6	AD39
6	IO_L55P_6	AC33
6	IO_L55N_6	AC34
6	IO_L56P_6	AB32
6	IO_L56N_6	AB33
6	IO_L57P_6	AC36
6	IO_L57N_6/VREF_6	AC37
6	IO_L58P_6	AB34
6	IO_L58N_6	AB35
6	IO_L59P_6	AA28
6	IO_L59N_6	AA29
6	IO_L60P_6	AC38
6	IO_L60N_6	AC39
6	IO_L85P_6	AB36
6	IO_L85N_6	AB37
6	IO_L86P_6	AA30
6	IO_L86N_6	AA31
6	IO_L87P_6	AB38

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
6	IO_L87N_6/VREF_6	AB39
6	IO_L88P_6	AA34
6	IO_L88N_6	AA35
6	IO_L89P_6	AA32
6	IO_L89N_6	AA33
6	IO_L90P_6	AA36
6	IO_L90N_6	AA37
7	IO_L90P_7	W37
7	IO_L90N_7	W36
7	IO_L89P_7	W33
7	IO_L89N_7	W32
7	IO_L88P_7	V39
7	IO_L88N_7/VREF_7	V38
7	IO_L87P_7	W35
7	IO_L87N_7	W34
7	IO_L86P_7	W31
7	IO_L86N_7	W30
7	IO_L85P_7	V37
7	IO_L85N_7	V36
7	IO_L60P_7	V35
7	IO_L60N_7	V34
7	IO_L59P_7	W29
7	IO_L59N_7	W28
7	IO_L58P_7	U39
7	IO_L58N_7/VREF_7	U38
7	IO_L57P_7	U34
7	IO_L57N_7	U33
7	IO_L56P_7	V33
7	IO_L56N_7	V32
7	IO_L55P_7	U37
7	IO_L55N_7	U36
7	IO_L54P_7	T37
7	IO_L54N_7	T36
7	IO_L53P_7	V31
7	IO_L53N_7	V30
7	IO_L52P_7	T39
7	IO_L52N_7/VREF_7	T38

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
7	IO_L51P_7	T35
7	IO_L51N_7	T34
7	IO_L50P_7	V29
7	IO_L50N_7	V28
7	IO_L49P_7	R38
7	IO_L49N_7	R37
7	IO_L48P_7	R36
7	IO_L48N_7	R35
7	IO_L47P_7	U31
7	IO_L47N_7	U30
7	IO_L46P_7	P39
7	IO_L46N_7/VREF_7	P38
7	IO_L45P_7	R34
7	IO_L45N_7	R33
7	IO_L44P_7	T33
7	IO_L44N_7	T32
7	IO_L43P_7	N39
7	IO_L43N_7	N38
7	IO_L42P_7	P36
7	IO_L42N_7	P35
7	IO_L41P_7	U29
7	IO_L41N_7	U28
7	IO_L40P_7	N37
7	IO_L40N_7/VREF_7	N36
7	IO_L39P_7	N35
7	IO_L39N_7	N34
7	IO_L38P_7	T31
7	IO_L38N_7	T30
7	IO_L37P_7	M39
7	IO_L37N_7	M38
7	IO_L36P_7	M35
7	IO_L36N_7	M34
7	IO_L35P_7	T29
7	IO_L35N_7	T28
7	IO_L34P_7	M37
7	IO_L34N_7/VREF_7	M36
7	IO_L33P_7	N33
7	IO_L33N_7	N32

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
7	IO_L32P_7	R31
7	IO_L32N_7	R30
7	IO_L31P_7	L38
7	IO_L31N_7	L37
7	IO_L30P_7	L35
7	IO_L30N_7	L34
7	IO_L29P_7	P33
7	IO_L29N_7	P32
7	IO_L28P_7	K39
7	IO_L28N_7/VREF_7	K38
7	IO_L27P_7	K36
7	IO_L27N_7	K35
7	IO_L26P_7	P31
7	IO_L26N_7	P30
7	IO_L25P_7	J39
7	IO_L25N_7	J38
7	IO_L24P_7	J37
7	IO_L24N_7	J36
7	IO_L23P_7	R29
7	IO_L23N_7	R28
7	IO_L22P_7	H39
7	IO_L22N_7/VREF_7	H38
7	IO_L21P_7	K34
7	IO_L21N_7	K33
7	IO_L20P_7	N31
7	IO_L20N_7	N30
7	IO_L19P_7	G39
7	IO_L19N_7	G38
7	IO_L18P_7	J35
7	IO_L18N_7	J34
7	IO_L17P_7	P29
7	IO_L17N_7	P28
7	IO_L16P_7	H37
7	IO_L16N_7/VREF_7	H36
7	IO_L15P_7	H35
7	IO_L15N_7	H34
7	IO_L14P_7	L33
7	IO_L14N_7	L32

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
7	IO_L13P_7	F39
7	IO_L13N_7	F38
7	IO_L12P_7	F37
7	IO_L12N_7	F36
7	IO_L11P_7	N29
7	IO_L11N_7	N28
7	IO_L10P_7	G37
7	IO_L10N_7/VREF_7	G36
7	IO_L09P_7	G35
7	IO_L09N_7	G34
7	IO_L08P_7	K32
7	IO_L08N_7	K31
7	IO_L07P_7	E39
7	IO_L07N_7	E38
7	IO_L06P_7	F35
7	IO_L06N_7	E34
7	IO_L05P_7	J33
7	IO_L05N_7	J32
7	IO_L04P_7	D39
7	IO_L04N_7/VREF_7	D38
7	IO_L03P_7	D35
7	IO_L03N_7	C35
7	IO_L02P_7	H33
7	IO_L02N_7	G32
7	IO_L01P_7/VRN_7	E37
7	IO_L01N_7/VRP_7	E36
0	VCCO_0	P25
0	VCCO_0	P24
0	VCCO_0	P23
0	VCCO_0	P22
0	VCCO_0	P21
0	VCCO_0	N26
0	VCCO_0	N25
0	VCCO_0	N24
0	VCCO_0	N23
0	VCCO_0	N22
0	VCCO_0	N21

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
0	VCCO_0	N20
0	VCCO_0	H23
0	VCCO_0	F26
0	VCCO_0	E28
1	VCCO_1	P20
1	VCCO_1	P19
1	VCCO_1	P18
1	VCCO_1	P17
1	VCCO_1	P16
1	VCCO_1	P15
1	VCCO_1	N19
1	VCCO_1	N18
1	VCCO_1	N17
1	VCCO_1	N16
1	VCCO_1	N15
1	VCCO_1	N14
1	VCCO_1	H17
1	VCCO_1	F14
1	VCCO_1	E12
2	VCCO_2	Y13
2	VCCO_2	W14
2	VCCO_2	W13
2	VCCO_2	V14
2	VCCO_2	V13
2	VCCO_2	U14
2	VCCO_2	U13
2	VCCO_2	T14
2	VCCO_2	T13
2	VCCO_2	R14
2	VCCO_2	R13
2	VCCO_2	R8
2	VCCO_2	P13
2	VCCO_2	M7
2	VCCO_2	L4
3	VCCO_3	AJ4
3	VCCO_3	AH7
3	VCCO_3	AF13
3	VCCO_3	AE14

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
3	VCCO_3	AE13
3	VCCO_3	AE8
3	VCCO_3	AD14
3	VCCO_3	AD13
3	VCCO_3	AC14
3	VCCO_3	AC13
3	VCCO_3	AB14
3	VCCO_3	AB13
3	VCCO_3	AA14
3	VCCO_3	AA13
3	VCCO_3	Y14
4	VCCO_4	AR12
4	VCCO_4	AP14
4	VCCO_4	AM17
4	VCCO_4	AG20
4	VCCO_4	AG19
4	VCCO_4	AG18
4	VCCO_4	AG17
4	VCCO_4	AG16
4	VCCO_4	AG15
4	VCCO_4	AG14
4	VCCO_4	AF19
4	VCCO_4	AF18
4	VCCO_4	AF17
4	VCCO_4	AF16
4	VCCO_4	AF15
5	VCCO_5	AR28
5	VCCO_5	AP26
5	VCCO_5	AM23
5	VCCO_5	AG26
5	VCCO_5	AG25
5	VCCO_5	AG24
5	VCCO_5	AG23
5	VCCO_5	AG22
5	VCCO_5	AG21
5	VCCO_5	AF25
5	VCCO_5	AF24
5	VCCO_5	AF23

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
5	VCCO_5	AF22
5	VCCO_5	AF21
5	VCCO_5	AF20
6	VCCO_6	AJ36
6	VCCO_6	AH33
6	VCCO_6	AF27
6	VCCO_6	AE32
6	VCCO_6	AE27
6	VCCO_6	AE26
6	VCCO_6	AD27
6	VCCO_6	AD26
6	VCCO_6	AC27
6	VCCO_6	AC26
6	VCCO_6	AB27
6	VCCO_6	AB26
6	VCCO_6	AA27
6	VCCO_6	AA26
6	VCCO_6	Y27
7	VCCO_7	Y26
7	VCCO_7	W27
7	VCCO_7	W26
7	VCCO_7	V27
7	VCCO_7	V26
7	VCCO_7	U27
7	VCCO_7	U26
7	VCCO_7	T27
7	VCCO_7	T26
7	VCCO_7	R32
7	VCCO_7	R27
7	VCCO_7	R26
7	VCCO_7	P27
7	VCCO_7	M33
7	VCCO_7	L36
N/A	CCLK	AT6
N/A	PROG_B	E33
N/A	DONE	AL10
N/A	M0	AT33

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	M1	AT34
N/A	M2	AL30
N/A	TCK	E7
N/A	TDI	F33
N/A	TDO	F7
N/A	TMS	D6
N/A	PWRDWN_B	AT7
N/A	HSWAP_EN	D34
N/A	RSVD	D7
N/A	VBATT	J10
N/A	DXP	J30
N/A	DXN	D33
N/A	AVCCAUXTX2	B35
N/A	VTTXPAD2	B36
N/A	TXNPAD2	A36
N/A	TXPPAD2	A35
N/A	GND A2	C34
N/A	GND A2	C34
N/A	RXPPAD2	A34
N/A	RXNPAD2	A33
N/A	VTRXPAD2	B34
N/A	AVCCAUXRX2	B33
N/A	AVCCAUXTX4	B31
N/A	VTTXPAD4	B32
N/A	TXNPAD4	A32
N/A	TXPPAD4	A31
N/A	GND A4	C31
N/A	GND A4	C31
N/A	RXPPAD4	A30
N/A	RXNPAD4	A29
N/A	VTRXPAD4	B30
N/A	AVCCAUXRX4	B29
N/A	AVCCAUXTX5	B27
N/A	VTTXPAD5	B28
N/A	TXNPAD5	A28
N/A	TXPPAD5	A27
N/A	GND A5	C27
N/A	GND A5	C27

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	RXPPAD5	A26
N/A	RXNPAD5	A25
N/A	VTRXPAD5	B26
N/A	AVCCAUXRX5	B25
N/A	AVCCAUXTX6	B23
N/A	VTTXPAD6	B24
N/A	TXNPAD6	A24
N/A	TXPPAD6	A23
N/A	GND A6	C24
N/A	GND A6	C24
N/A	RXPPAD6	A22
N/A	RXNPAD6	A21
N/A	VTRXPAD6	B22
N/A	AVCCAUXRX6	B21
N/A	AVCCAUXTX7	B18
N/A	VTTXPAD7	B19
N/A	TXNPAD7	A19
N/A	TXPPAD7	A18
N/A	GND A7	C16
N/A	GND A7	C16
N/A	RXPPAD7	A17
N/A	RXNPAD7	A16
N/A	VTRXPAD7	B17
N/A	AVCCAUXRX7	B16
N/A	AVCCAUXTX8	B14
N/A	VTTXPAD8	B15
N/A	TXNPAD8	A15
N/A	TXPPAD8	A14
N/A	GND A8	C13
N/A	GND A8	C13
N/A	RXPPAD8	A13
N/A	RXNPAD8	A12
N/A	VTRXPAD8	B13
N/A	AVCCAUXRX8	B12
N/A	AVCCAUXTX9	B10
N/A	VTTXPAD9	B11
N/A	TXNPAD9	A11
N/A	TXPPAD9	A10

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	GNDA9	C9
N/A	GNDA9	C9
N/A	RXPPAD9	A9
N/A	RXNPAD9	A8
N/A	VTRXPAD9	B9
N/A	AVCCAUXRX9	B8
N/A	AVCCAUXTX11	B6
N/A	VTTXPAD11	B7
N/A	TXNPAD11	A7
N/A	TXPPAD11	A6
N/A	GNDA11	C6
N/A	GNDA11	C6
N/A	RXPPAD11	A5
N/A	RXNPAD11	A4
N/A	VTRXPAD11	B5
N/A	AVCCAUXRX11	B4
N/A	AVCCAUXRX14	AV4
N/A	VTRXPAD14	AV5
N/A	RXNPAD14	AW4
N/A	RXPPAD14	AW5
N/A	GNDA14	AU6
N/A	GNDA14	AU6
N/A	TXPPAD14	AW6
N/A	TXNPAD14	AW7
N/A	VTTXPAD14	AV7
N/A	AVCCAUXTX14	AV6
N/A	AVCCAUXRX16	AV8
N/A	VTRXPAD16	AV9
N/A	RXNPAD16	AW8
N/A	RXPPAD16	AW9
N/A	GNDA16	AU9
N/A	GNDA16	AU9
N/A	TXPPAD16	AW10
N/A	TXNPAD16	AW11
N/A	VTTXPAD16	AV11
N/A	AVCCAUXTX16	AV10
N/A	AVCCAUXRX17	AV12
N/A	VTRXPAD17	AV13

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	RXNPAD17	AW12
N/A	RXPPAD17	AW13
N/A	GNDA17	AU13
N/A	GNDA17	AU13
N/A	TXPPAD17	AW14
N/A	TXNPAD17	AW15
N/A	VTTXPAD17	AV15
N/A	AVCCAUXTX17	AV14
N/A	AVCCAUXRX18	AV16
N/A	VTRXPAD18	AV17
N/A	RXNPAD18	AW16
N/A	RXPPAD18	AW17
N/A	GNDA18	AU16
N/A	GNDA18	AU16
N/A	TXPPAD18	AW18
N/A	TXNPAD18	AW19
N/A	VTTXPAD18	AV19
N/A	AVCCAUXTX18	AV18
N/A	AVCCAUXRX19	AV21
N/A	VTRXPAD19	AV22
N/A	RXNPAD19	AW21
N/A	RXPPAD19	AW22
N/A	GNDA19	AU24
N/A	GNDA19	AU24
N/A	TXPPAD19	AW23
N/A	TXNPAD19	AW24
N/A	VTTXPAD19	AV24
N/A	AVCCAUXTX19	AV23
N/A	AVCCAUXRX20	AV25
N/A	VTRXPAD20	AV26
N/A	RXNPAD20	AW25
N/A	RXPPAD20	AW26
N/A	GNDA20	AU27
N/A	GNDA20	AU27
N/A	TXPPAD20	AW27
N/A	TXNPAD20	AW28
N/A	VTTXPAD20	AV28
N/A	AVCCAUXTX20	AV27

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	AVCCAUXRX21	AV29
N/A	VTRXPAD21	AV30
N/A	RXNPAD21	AW29
N/A	RXPPAD21	AW30
N/A	GND A21	AU31
N/A	GND A21	AU31
N/A	TXPPAD21	AW31
N/A	TXNPAD21	AW32
N/A	VTTXPAD21	AV32
N/A	AVCCAUTX21	AV31
N/A	AVCCAUXRX23	AV33
N/A	VTRXPAD23	AV34
N/A	RXNPAD23	AW33
N/A	RXPPAD23	AW34
N/A	GND A23	AU34
N/A	GND A23	AU34
N/A	TXPPAD23	AW35
N/A	TXNPAD23	AW36
N/A	VTTXPAD23	AV36
N/A	AVCCAUTX23	AV35
N/A	VCCINT	AH28
N/A	VCCINT	AH12
N/A	VCCINT	AG27
N/A	VCCINT	AG13
N/A	VCCINT	AF26
N/A	VCCINT	AF14
N/A	VCCINT	AE25
N/A	VCCINT	AE24
N/A	VCCINT	AE23
N/A	VCCINT	AE22
N/A	VCCINT	AE21
N/A	VCCINT	AE20
N/A	VCCINT	AE19
N/A	VCCINT	AE18
N/A	VCCINT	AE17
N/A	VCCINT	AE16
N/A	VCCINT	AE15

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	VCCINT	AD25
N/A	VCCINT	AD24
N/A	VCCINT	AD16
N/A	VCCINT	AD15
N/A	VCCINT	AC25
N/A	VCCINT	AC15
N/A	VCCINT	AB25
N/A	VCCINT	AB15
N/A	VCCINT	AA25
N/A	VCCINT	AA15
N/A	VCCINT	Y25
N/A	VCCINT	Y15
N/A	VCCINT	W25
N/A	VCCINT	W15
N/A	VCCINT	V25
N/A	VCCINT	V15
N/A	VCCINT	U25
N/A	VCCINT	U15
N/A	VCCINT	T25
N/A	VCCINT	T24
N/A	VCCINT	T16
N/A	VCCINT	T15
N/A	VCCINT	R25
N/A	VCCINT	R24
N/A	VCCINT	R23
N/A	VCCINT	R22
N/A	VCCINT	R21
N/A	VCCINT	R20
N/A	VCCINT	R19
N/A	VCCINT	R18
N/A	VCCINT	R17
N/A	VCCINT	R16
N/A	VCCINT	R15
N/A	VCCINT	P26
N/A	VCCINT	P14
N/A	VCCINT	N27
N/A	VCCINT	N13
N/A	VCCINT	M28

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	VCCINT	M12
N/A	VCCAUX	AV20
N/A	VCCAUX	AU36
N/A	VCCAUX	AU20
N/A	VCCAUX	AU4
N/A	VCCAUX	AT37
N/A	VCCAUX	AT3
N/A	VCCAUX	AL31
N/A	VCCAUX	AL9
N/A	VCCAUX	AK30
N/A	VCCAUX	AK10
N/A	VCCAUX	AA39
N/A	VCCAUX	AA1
N/A	VCCAUX	Y39
N/A	VCCAUX	Y38
N/A	VCCAUX	Y2
N/A	VCCAUX	Y1
N/A	VCCAUX	W39
N/A	VCCAUX	W1
N/A	VCCAUX	K30
N/A	VCCAUX	K10
N/A	VCCAUX	J31
N/A	VCCAUX	J9
N/A	VCCAUX	D37
N/A	VCCAUX	D3
N/A	VCCAUX	C36
N/A	VCCAUX	C20
N/A	VCCAUX	C4
N/A	VCCAUX	B20
N/A	GND	AW38
N/A	GND	AW37
N/A	GND	AW20
N/A	GND	AW3
N/A	GND	AW2
N/A	GND	AV39
N/A	GND	AV38
N/A	GND	AV37
N/A	GND	AV3

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	GND	AV2
N/A	GND	AV1
N/A	GND	AU39
N/A	GND	AU38
N/A	GND	AU37
N/A	GND	AU30
N/A	GND	AU26
N/A	GND	AU14
N/A	GND	AU10
N/A	GND	AU3
N/A	GND	AU2
N/A	GND	AU1
N/A	GND	AT36
N/A	GND	AT20
N/A	GND	AT4
N/A	GND	AR35
N/A	GND	AR23
N/A	GND	AR17
N/A	GND	AR5
N/A	GND	AP34
N/A	GND	AP6
N/A	GND	AN33
N/A	GND	AN20
N/A	GND	AN7
N/A	GND	AM32
N/A	GND	AM8
N/A	GND	AK37
N/A	GND	AK20
N/A	GND	AK3
N/A	GND	AJ39
N/A	GND	AJ1
N/A	GND	AF37
N/A	GND	AF34
N/A	GND	AF6
N/A	GND	AF3
N/A	GND	AE39
N/A	GND	AE1
N/A	GND	AD23

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	GND	AD22
N/A	GND	AD21
N/A	GND	AD20
N/A	GND	AD19
N/A	GND	AD18
N/A	GND	AD17
N/A	GND	AC35
N/A	GND	AC32
N/A	GND	AC24
N/A	GND	AC23
N/A	GND	AC22
N/A	GND	AC21
N/A	GND	AC20
N/A	GND	AC19
N/A	GND	AC18
N/A	GND	AC17
N/A	GND	AC16
N/A	GND	AC8
N/A	GND	AC5
N/A	GND	AB24
N/A	GND	AB23
N/A	GND	AB22
N/A	GND	AB21
N/A	GND	AB20
N/A	GND	AB19
N/A	GND	AB18
N/A	GND	AB17
N/A	GND	AB16
N/A	GND	AA38
N/A	GND	AA24
N/A	GND	AA23
N/A	GND	AA22
N/A	GND	AA21
N/A	GND	AA20
N/A	GND	AA19
N/A	GND	AA18
N/A	GND	AA17
N/A	GND	AA16

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	GND	AA2
N/A	GND	Y37
N/A	GND	Y36
N/A	GND	Y33
N/A	GND	Y30
N/A	GND	Y24
N/A	GND	Y23
N/A	GND	Y22
N/A	GND	Y21
N/A	GND	Y20
N/A	GND	Y19
N/A	GND	Y18
N/A	GND	Y17
N/A	GND	Y16
N/A	GND	Y10
N/A	GND	Y7
N/A	GND	Y4
N/A	GND	Y3
N/A	GND	W38
N/A	GND	W24
N/A	GND	W23
N/A	GND	W22
N/A	GND	W21
N/A	GND	W20
N/A	GND	W19
N/A	GND	W18
N/A	GND	W17
N/A	GND	W16
N/A	GND	W2
N/A	GND	V24
N/A	GND	V23
N/A	GND	V22
N/A	GND	V21
N/A	GND	V20
N/A	GND	V19
N/A	GND	V18
N/A	GND	V17
N/A	GND	V16

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	GND	U35
N/A	GND	U32
N/A	GND	U24
N/A	GND	U23
N/A	GND	U22
N/A	GND	U21
N/A	GND	U20
N/A	GND	U19
N/A	GND	U18
N/A	GND	U17
N/A	GND	U16
N/A	GND	U8
N/A	GND	U5
N/A	GND	T23
N/A	GND	T22
N/A	GND	T21
N/A	GND	T20
N/A	GND	T19
N/A	GND	T18
N/A	GND	T17
N/A	GND	R39
N/A	GND	R1
N/A	GND	P37
N/A	GND	P34
N/A	GND	P6
N/A	GND	P3
N/A	GND	L39
N/A	GND	L1
N/A	GND	K37
N/A	GND	K20
N/A	GND	K3
N/A	GND	H32
N/A	GND	H8
N/A	GND	G33
N/A	GND	G20
N/A	GND	G7
N/A	GND	F34
N/A	GND	F6

Table 11: FF1517 — XC2VP50

Bank	Pin Description	Pin Number
N/A	GND	E35
N/A	GND	E23
N/A	GND	E17
N/A	GND	E5
N/A	GND	D36
N/A	GND	D20
N/A	GND	D4
N/A	GND	C39
N/A	GND	C38
N/A	GND	C37
N/A	GND	C30
N/A	GND	C26
N/A	GND	C14
N/A	GND	C10
N/A	GND	C3
N/A	GND	C2
N/A	GND	C1
N/A	GND	B39
N/A	GND	B38
N/A	GND	B37
N/A	GND	B3
N/A	GND	B2
N/A	GND	B1
N/A	GND	A38
N/A	GND	A37
N/A	GND	A20
N/A	GND	A3
N/A	GND	A2

FF1517 Flip-Chip Fine-Pitch BGA Package Specifications (1.00mm pitch)

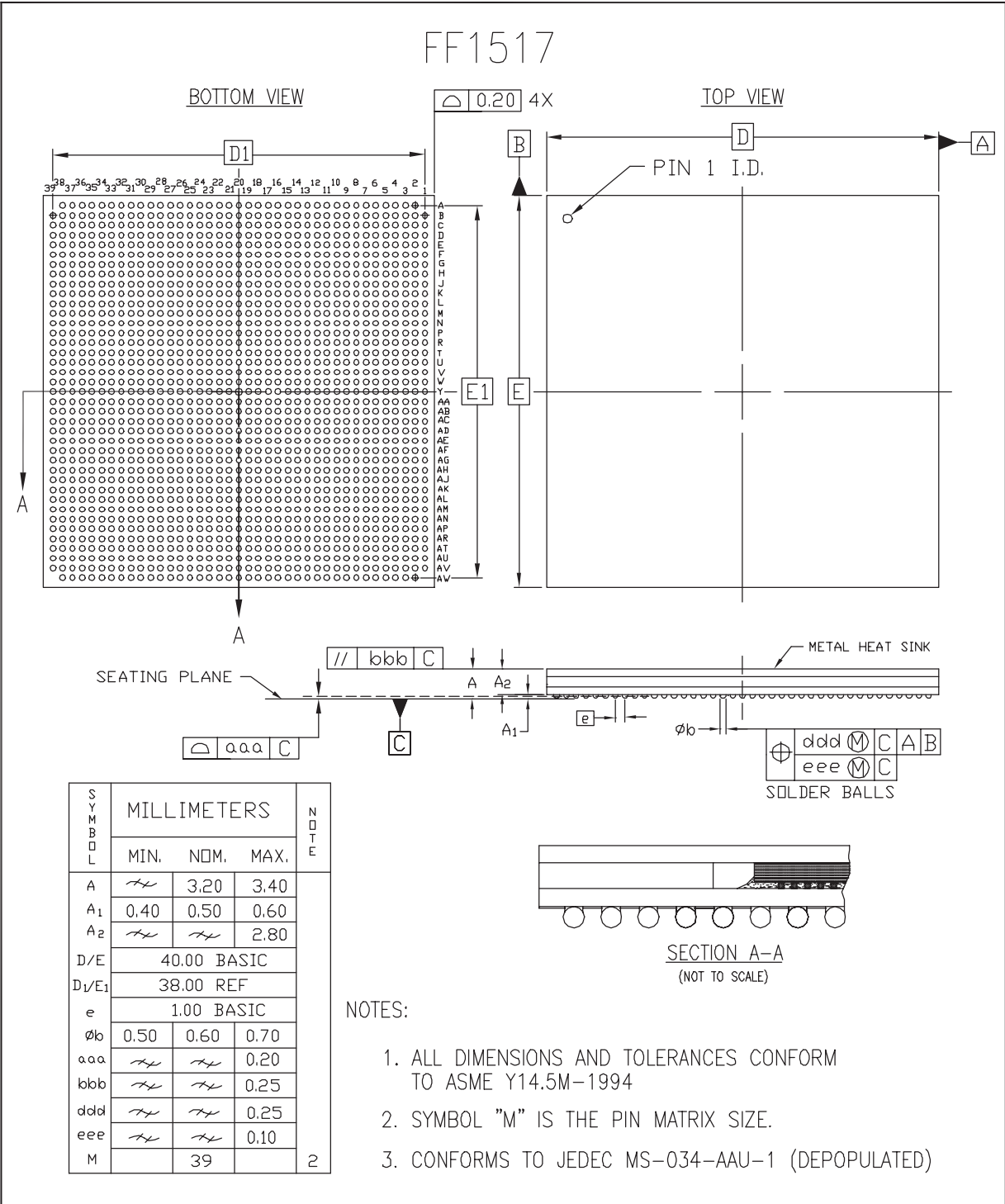


Figure 6: FF1517 Flip-Chip Fine-Pitch BGA Package Specifications

BF957 Flip-Chip BGA Package

As shown in Table 12, XC2VP20 and XC2VP50 Virtex-II Pro devices are available in the BF957 flip-chip BGA package. Pins in each of these devices are the same, except for the differences shown in the "No Connects" column. Following this table are the **BF957 Flip-Chip BGA Package Specifications (1.27mm pitch)**.

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
0	IO_L01N_0/VRP_0	E26		
0	IO_L01P_0/VRN_0	E25		
0	IO_L02N_0	H23		
0	IO_L02P_0	G23		
0	IO_L03N_0	F25		
0	IO_L03P_0/VREF_0	F24		
0	IO_L05_0/No_Pair	G24		
0	IO_L06N_0	J22		
0	IO_L06P_0	H22		
0	IO_L07N_0	F23		
0	IO_L07P_0	E23		
0	IO_L08N_0	D25		
0	IO_L08P_0	C25		
0	IO_L09N_0	K21		
0	IO_L09P_0/VREF_0	J21		
0	IO_L19N_0	G22	NC	
0	IO_L19P_0	F22	NC	
0	IO_L37N_0	H21		
0	IO_L37P_0	G21		
0	IO_L38N_0	E24		
0	IO_L38P_0	D24		
0	IO_L39N_0	K20		
0	IO_L39P_0	J20		
0	IO_L43N_0	F21		
0	IO_L43P_0	E21		
0	IO_L44N_0	D23		
0	IO_L44P_0	D22		
0	IO_L45N_0	H20		
0	IO_L45P_0/VREF_0	G20		
0	IO_L46N_0	F20		
0	IO_L46P_0	E20		
0	IO_L47N_0	C22		
0	IO_L47P_0	C21		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
0	IO_L48N_0	K19		
0	IO_L48P_0	J19		
0	IO_L49N_0	H19		
0	IO_L49P_0	G19		
0	IO_L50_0/No_Pair	D21		
0	IO_L53_0/No_Pair	D20		
0	IO_L54N_0	K18		
0	IO_L54P_0	J18		
0	IO_L55N_0	F18		
0	IO_L55P_0	E18		
0	IO_L56N_0	E19		
0	IO_L56P_0	D19		
0	IO_L57N_0	H18		
0	IO_L57P_0/VREF_0	G18		
0	IO_L67N_0	H17		
0	IO_L67P_0	G17		
0	IO_L68N_0	D18		
0	IO_L68P_0	C18		
0	IO_L69N_0	K17		
0	IO_L69P_0/VREF_0	J17		
0	IO_L73N_0	F17		
0	IO_L73P_0	E17		
0	IO_L74N_0/GCLK7P	D17		
0	IO_L74P_0/GCLK6S	C17		
0	IO_L75N_0/GCLK5P	F16		
0	IO_L75P_0/GCLK4S	E16		
1	IO_L75N_1/GCLK3P	H16		
1	IO_L75P_1/GCLK2S	J16		
1	IO_L74N_1/GCLK1P	C15		
1	IO_L74P_1/GCLK0S	D15		
1	IO_L73N_1	E15		
1	IO_L73P_1	F15		
1	IO_L69N_1/VREF_1	J15		
1	IO_L69P_1	K15		
1	IO_L68N_1	C14		
1	IO_L68P_1	D14		
1	IO_L67N_1	G15		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
1	IO_L67P_1	H15		
1	IO_L57N_1/VREF_1	G14		
1	IO_L57P_1	H14		
1	IO_L56N_1	D13		
1	IO_L56P_1	E13		
1	IO_L55N_1	E14		
1	IO_L55P_1	F14		
1	IO_L54N_1	J14		
1	IO_L54P_1	K14		
1	IO_L53_1/No_Pair	D12		
1	IO_L50_1/No_Pair	D11		
1	IO_L49N_1	G13		
1	IO_L49P_1	H13		
1	IO_L48N_1	J13		
1	IO_L48P_1	K13		
1	IO_L47N_1	C11		
1	IO_L47P_1	C10		
1	IO_L46N_1	E12		
1	IO_L46P_1	F12		
1	IO_L45N_1/VREF_1	G12		
1	IO_L45P_1	H12		
1	IO_L44N_1	D10		
1	IO_L44P_1	D9		
1	IO_L43N_1	E11		
1	IO_L43P_1	F11		
1	IO_L39N_1	J12		
1	IO_L39P_1	K12		
1	IO_L38N_1	D8		
1	IO_L38P_1	E8		
1	IO_L37N_1	G11		
1	IO_L37P_1	H11		
1	IO_L19N_1	F10	NC	
1	IO_L19P_1	G10	NC	
1	IO_L09N_1/VREF_1	J11		
1	IO_L09P_1	K11		
1	IO_L08N_1	C7		
1	IO_L08P_1	D7		
1	IO_L07N_1	E9		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
1	IO_L07P_1	F9		
1	IO_L06N_1	H10		
1	IO_L06P_1	J10		
1	IO_L05_1/No_Pair	G8		
1	IO_L03N_1/VREF_1	F8		
1	IO_L03P_1	F7		
1	IO_L02N_1	G9		
1	IO_L02P_1	H9		
1	IO_L01N_1/VRP_1	E7		
1	IO_L01P_1/VRN_1	E6		
2	IO_L01N_2/VRP_2	D2		
2	IO_L01P_2/VRN_2	D1		
2	IO_L02N_2	K9		
2	IO_L02P_2	K8		
2	IO_L03N_2	C4		
2	IO_L03P_2	D3		
2	IO_L04N_2/VREF_2	E2		
2	IO_L04P_2	E1		
2	IO_L05N_2	L10		
2	IO_L05P_2	L9		
2	IO_L06N_2	E4		
2	IO_L06P_2	E3		
2	IO_L18N_2	F5	NC	
2	IO_L18P_2	F4	NC	
2	IO_L31N_2	G4		
2	IO_L31P_2	G3		
2	IO_L32N_2	J7		
2	IO_L32P_2	J6		
2	IO_L33N_2	G6		
2	IO_L33P_2	G5		
2	IO_L34N_2/VREF_2	F2		
2	IO_L34P_2	F1		
2	IO_L35N_2	K7		
2	IO_L35P_2	K6		
2	IO_L36N_2	H5		
2	IO_L36P_2	H4		
2	IO_L37N_2	G2		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
2	IO_L37P_2	G1		
2	IO_L38N_2	M10		
2	IO_L38P_2	M9		
2	IO_L39N_2	J4		
2	IO_L39P_2	J3		
2	IO_L40N_2/VREF_2	H2		
2	IO_L40P_2	H1		
2	IO_L41N_2	L8		
2	IO_L41P_2	L7		
2	IO_L42N_2	K5		
2	IO_L42P_2	K4		
2	IO_L43N_2	J2		
2	IO_L43P_2	J1		
2	IO_L44N_2	M8		
2	IO_L44P_2	M7		
2	IO_L45N_2	L6		
2	IO_L45P_2	L5		
2	IO_L46N_2/VREF_2	K2		
2	IO_L46P_2	K1		
2	IO_L47N_2	N10		
2	IO_L47P_2	N9		
2	IO_L48N_2	L4		
2	IO_L48P_2	L3		
2	IO_L49N_2	L2		
2	IO_L49P_2	L1		
2	IO_L50N_2	M6		
2	IO_L50P_2	M5		
2	IO_L51N_2	M4		
2	IO_L51P_2	M3		
2	IO_L52N_2/VREF_2	M2		
2	IO_L52P_2	M1		
2	IO_L53N_2	N8		
2	IO_L53P_2	N7		
2	IO_L54N_2	N5		
2	IO_L54P_2	N4		
2	IO_L55N_2	N2		
2	IO_L55P_2	N1		
2	IO_L56N_2	P10		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
2	IO_L56P_2	P9		
2	IO_L57N_2	P6		
2	IO_L57P_2	P5		
2	IO_L58N_2/VREF_2	P4		
2	IO_L58P_2	P3		
2	IO_L59N_2	P8		
2	IO_L59P_2	P7		
2	IO_L60N_2	R6		
2	IO_L60P_2	R5		
2	IO_L85N_2	P2		
2	IO_L85P_2	P1		
2	IO_L86N_2	R10		
2	IO_L86P_2	R9		
2	IO_L87N_2	R4		
2	IO_L87P_2	R3		
2	IO_L88N_2/VREF_2	R2		
2	IO_L88P_2	R1		
2	IO_L89N_2	R8		
2	IO_L89P_2	R7		
2	IO_L90N_2	T5		
2	IO_L90P_2	T6		
3	IO_L90N_3	U1		
3	IO_L90P_3	U2		
3	IO_L89N_3	T8		
3	IO_L89P_3	T9		
3	IO_L88N_3	U3		
3	IO_L88P_3	U4		
3	IO_L87N_3/VREF_3	V1		
3	IO_L87P_3	V2		
3	IO_L86N_3	U7		
3	IO_L86P_3	U8		
3	IO_L85N_3	U5		
3	IO_L85P_3	U6		
3	IO_L60N_3	V3		
3	IO_L60P_3	V4		
3	IO_L59N_3	U9		
3	IO_L59P_3	U10		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
3	IO_L58N_3	V5		
3	IO_L58P_3	V6		
3	IO_L57N_3/VREF_3	W1		
3	IO_L57P_3	W2		
3	IO_L56N_3	V7		
3	IO_L56P_3	V8		
3	IO_L55N_3	W4		
3	IO_L55P_3	W5		
3	IO_L54N_3	Y1		
3	IO_L54P_3	Y2		
3	IO_L53N_3	V9		
3	IO_L53P_3	V10		
3	IO_L52N_3	Y3		
3	IO_L52P_3	Y4		
3	IO_L51N_3/VREF_3	AA1		
3	IO_L51P_3	AA2		
3	IO_L50N_3	W7		
3	IO_L50P_3	W8		
3	IO_L49N_3	Y5		
3	IO_L49P_3	Y6		
3	IO_L48N_3	AB1		
3	IO_L48P_3	AB2		
3	IO_L47N_3	W9		
3	IO_L47P_3	W10		
3	IO_L46N_3	AA3		
3	IO_L46P_3	AA4		
3	IO_L45N_3/VREF_3	AC1		
3	IO_L45P_3	AC2		
3	IO_L44N_3	Y7		
3	IO_L44P_3	Y8		
3	IO_L43N_3	AA5		
3	IO_L43P_3	AA6		
3	IO_L42N_3	AD1		
3	IO_L42P_3	AD2		
3	IO_L41N_3	AA7		
3	IO_L41P_3	AA8		
3	IO_L40N_3	AB4		
3	IO_L40P_3	AB5		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
3	IO_L39N_3/VREF_3	AC3		
3	IO_L39P_3	AC4		
3	IO_L38N_3	Y9		
3	IO_L38P_3	Y10		
3	IO_L37N_3	AD4		
3	IO_L37P_3	AD5		
3	IO_L36N_3	AE1		
3	IO_L36P_3	AE2		
3	IO_L35N_3	AB6		
3	IO_L35P_3	AB7		
3	IO_L34N_3	AE3		
3	IO_L34P_3	AE4		
3	IO_L33N_3/VREF_3	AF1		
3	IO_L33P_3	AF2		
3	IO_L32N_3	AC6		
3	IO_L32P_3	AC7		
3	IO_L31N_3	AE5		
3	IO_L31P_3	AE6		
3	IO_L18N_3	AG1	NC	
3	IO_L18P_3	AG2	NC	
3	IO_L17N_3	AA9	NC	
3	IO_L17P_3	AA10	NC	
3	IO_L06N_3	AH1		
3	IO_L06P_3	AH2		
3	IO_L05N_3	AB8		
3	IO_L05P_3	AB9		
3	IO_L04N_3	AF4		
3	IO_L04P_3	AF5		
3	IO_L03N_3/VREF_3	AG3		
3	IO_L03P_3	AG4		
3	IO_L02N_3	AD6		
3	IO_L02P_3	AD7		
3	IO_L01N_3/VRP_3	AH3		
3	IO_L01P_3/VRN_3	AJ4		
4	IO_L01N_4/DOUT	AG6		
4	IO_L01P_4/INIT_B	AG7		
4	IO_L02N_4/D0	AF7		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
4	IO_L02P_4/D1	AF8		
4	IO_L03N_4/D2	AH7		
4	IO_L03P_4/D3	AJ7		
4	IO_L05_4/No_Pair	AE8		
4	IO_L06N_4/VRP_4	AG8		
4	IO_L06P_4/VRN_4	AH8		
4	IO_L07N_4	AC10		
4	IO_L07P_4/VREF_4	AD10		
4	IO_L08N_4	AD9		
4	IO_L08P_4	AE9		
4	IO_L09N_4	AF9		
4	IO_L09P_4/VREF_4	AG9		
4	IO_L19N_4	AB11	NC	
4	IO_L19P_4	AC11	NC	
4	IO_L37N_4	AB12		
4	IO_L37P_4	AC12		
4	IO_L38N_4	AE10		
4	IO_L38P_4	AF10		
4	IO_L39N_4	AH9		
4	IO_L39P_4	AH10		
4	IO_L43N_4	AD11		
4	IO_L43P_4	AE11		
4	IO_L44N_4	AF11		
4	IO_L44P_4	AG11		
4	IO_L45N_4	AJ10		
4	IO_L45P_4/VREF_4	AJ11		
4	IO_L46N_4	AB13		
4	IO_L46P_4	AC13		
4	IO_L47N_4	AD12		
4	IO_L47P_4	AE12		
4	IO_L48N_4	AH11		
4	IO_L48P_4	AH12		
4	IO_L49N_4	AB14		
4	IO_L49P_4	AC14		
4	IO_L50_4/No_Pair	AF12		
4	IO_L53_4/No_Pair	AG12		
4	IO_L54N_4	AG13		
4	IO_L54P_4	AH13		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
4	IO_L55N_4	AD14		
4	IO_L55P_4	AE14		
4	IO_L56N_4	AD13		
4	IO_L56P_4	AE13		
4	IO_L57N_4	AF14		
4	IO_L57P_4/VREF_4	AG14		
4	IO_L67N_4	AB15		
4	IO_L67P_4	AC15		
4	IO_L68N_4	AD15		
4	IO_L68P_4	AE15		
4	IO_L69N_4	AH14		
4	IO_L69P_4/VREF_4	AJ14		
4	IO_L73N_4	AC16		
4	IO_L73P_4	AD16		
4	IO_L74N_4/GCLK3S	AF15		
4	IO_L74P_4/GCLK2P	AG15		
4	IO_L75N_4/GCLK1S	AH15		
4	IO_L75P_4/GCLK0P	AJ15		
5	IO_L75N_5/GCLK7S	AJ17		
5	IO_L75P_5/GCLK6P	AH17		
5	IO_L74N_5/GCLK5S	AG17		
5	IO_L74P_5/GCLK4P	AF17		
5	IO_L73N_5	AG16		
5	IO_L73P_5	AF16		
5	IO_L69N_5/VREF_5	AJ18		
5	IO_L69P_5	AH18		
5	IO_L68N_5	AE17		
5	IO_L68P_5	AD17		
5	IO_L67N_5	AC17		
5	IO_L67P_5	AB17		
5	IO_L57N_5/VREF_5	AG18		
5	IO_L57P_5	AF18		
5	IO_L56N_5	AE19		
5	IO_L56P_5	AD19		
5	IO_L55N_5	AE18		
5	IO_L55P_5	AD18		
5	IO_L54N_5	AH19		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
5	IO_L54P_5	AG19		
5	IO_L53_5/No_Pair	AG20		
5	IO_L50_5/No_Pair	AF20		
5	IO_L49N_5	AC18		
5	IO_L49P_5	AB18		
5	IO_L48N_5	AH20		
5	IO_L48P_5	AH21		
5	IO_L47N_5	AE20		
5	IO_L47P_5	AD20		
5	IO_L46N_5	AC19		
5	IO_L46P_5	AB19		
5	IO_L45N_5/VREF_5	AJ21		
5	IO_L45P_5	AJ22		
5	IO_L44N_5	AG21		
5	IO_L44P_5	AF21		
5	IO_L43N_5	AE21		
5	IO_L43P_5	AD21		
5	IO_L39N_5	AH22		
5	IO_L39P_5	AH23		
5	IO_L38N_5	AF22		
5	IO_L38P_5	AE22		
5	IO_L37N_5	AC20		
5	IO_L37P_5	AB20		
5	IO_L19N_5	AC21	NC	
5	IO_L19P_5	AB21	NC	
5	IO_L09N_5/VREF_5	AG23		
5	IO_L09P_5	AF23		
5	IO_L08N_5	AE23		
5	IO_L08P_5	AD23		
5	IO_L07N_5/VREF_5	AD22		
5	IO_L07P_5	AC22		
5	IO_L06N_5/VRP_5	AH24		
5	IO_L06P_5/VRN_5	AG24		
5	IO_L05_5/No_Pair	AE24		
5	IO_L03N_5/D4	AJ25		
5	IO_L03P_5/D5	AH25		
5	IO_L02N_5/D6	AF24		
5	IO_L02P_5/D7	AF25		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
5	IO_L01N_5/RDWR_B	AG25		
5	IO_L01P_5/CS_B	AG26		
6	IO_L01P_6/VRN_6	AJ28		
6	IO_L01N_6/VRP_6	AH29		
6	IO_L02P_6	AD25		
6	IO_L02N_6	AD26		
6	IO_L03P_6	AG28		
6	IO_L03N_6/VREF_6	AG29		
6	IO_L04P_6	AF27		
6	IO_L04N_6	AF28		
6	IO_L05P_6	AB23		
6	IO_L05N_6	AB24		
6	IO_L06P_6	AH30		
6	IO_L06N_6	AH31		
6	IO_L17P_6	AA22	NC	
6	IO_L17N_6	AA23	NC	
6	IO_L18P_6	AG30	NC	
6	IO_L18N_6	AG31	NC	
6	IO_L31P_6	AE26		
6	IO_L31N_6	AE27		
6	IO_L32P_6	AC25		
6	IO_L32N_6	AC26		
6	IO_L33P_6	AF30		
6	IO_L33N_6/VREF_6	AF31		
6	IO_L34P_6	AE28		
6	IO_L34N_6	AE29		
6	IO_L35P_6	AB25		
6	IO_L35N_6	AB26		
6	IO_L36P_6	AE30		
6	IO_L36N_6	AE31		
6	IO_L37P_6	AD27		
6	IO_L37N_6	AD28		
6	IO_L38P_6	Y22		
6	IO_L38N_6	Y23		
6	IO_L39P_6	AC28		
6	IO_L39N_6/VREF_6	AC29		
6	IO_L40P_6	AB27		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
6	IO_L40N_6	AB28		
6	IO_L41P_6	AA24		
6	IO_L41N_6	AA25		
6	IO_L42P_6	AD30		
6	IO_L42N_6	AD31		
6	IO_L43P_6	AA26		
6	IO_L43N_6	AA27		
6	IO_L44P_6	Y24		
6	IO_L44N_6	Y25		
6	IO_L45P_6	AC30		
6	IO_L45N_6/VREF_6	AC31		
6	IO_L46P_6	AA28		
6	IO_L46N_6	AA29		
6	IO_L47P_6	W22		
6	IO_L47N_6	W23		
6	IO_L48P_6	AB30		
6	IO_L48N_6	AB31		
6	IO_L49P_6	Y26		
6	IO_L49N_6	Y27		
6	IO_L50P_6	W24		
6	IO_L50N_6	W25		
6	IO_L51P_6	AA30		
6	IO_L51N_6/VREF_6	AA31		
6	IO_L52P_6	Y28		
6	IO_L52N_6	Y29		
6	IO_L53P_6	V22		
6	IO_L53N_6	V23		
6	IO_L54P_6	Y30		
6	IO_L54N_6	Y31		
6	IO_L55P_6	W27		
6	IO_L55N_6	W28		
6	IO_L56P_6	V24		
6	IO_L56N_6	V25		
6	IO_L57P_6	W30		
6	IO_L57N_6/VREF_6	W31		
6	IO_L58P_6	V26		
6	IO_L58N_6	V27		
6	IO_L59P_6	U22		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
6	IO_L59N_6	U23		
6	IO_L60P_6	V28		
6	IO_L60N_6	V29		
6	IO_L85P_6	U26		
6	IO_L85N_6	U27		
6	IO_L86P_6	U24		
6	IO_L86N_6	U25		
6	IO_L87P_6	V30		
6	IO_L87N_6/VREF_6	V31		
6	IO_L88P_6	U28		
6	IO_L88N_6	U29		
6	IO_L89P_6	T23		
6	IO_L89N_6	T24		
6	IO_L90P_6	U30		
6	IO_L90N_6	U31		
7	IO_L90P_7	T26		
7	IO_L90N_7	T27		
7	IO_L89P_7	R25		
7	IO_L89N_7	R24		
7	IO_L88P_7	R31		
7	IO_L88N_7/VREF_7	R30		
7	IO_L87P_7	R29		
7	IO_L87N_7	R28		
7	IO_L86P_7	R23		
7	IO_L86N_7	R22		
7	IO_L85P_7	P31		
7	IO_L85N_7	P30		
7	IO_L60P_7	R27		
7	IO_L60N_7	R26		
7	IO_L59P_7	P25		
7	IO_L59N_7	P24		
7	IO_L58P_7	P29		
7	IO_L58N_7/VREF_7	P28		
7	IO_L57P_7	P27		
7	IO_L57N_7	P26		
7	IO_L56P_7	P23		
7	IO_L56N_7	P22		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
7	IO_L55P_7	N31		
7	IO_L55N_7	N30		
7	IO_L54P_7	N28		
7	IO_L54N_7	N27		
7	IO_L53P_7	N25		
7	IO_L53N_7	N24		
7	IO_L52P_7	M31		
7	IO_L52N_7/VREF_7	M30		
7	IO_L51P_7	M29		
7	IO_L51N_7	M28		
7	IO_L50P_7	M27		
7	IO_L50N_7	M26		
7	IO_L49P_7	L31		
7	IO_L49N_7	L30		
7	IO_L48P_7	L29		
7	IO_L48N_7	L28		
7	IO_L47P_7	N23		
7	IO_L47N_7	N22		
7	IO_L46P_7	K31		
7	IO_L46N_7/VREF_7	K30		
7	IO_L45P_7	L27		
7	IO_L45N_7	L26		
7	IO_L44P_7	M25		
7	IO_L44N_7	M24		
7	IO_L43P_7	J31		
7	IO_L43N_7	J30		
7	IO_L42P_7	K28		
7	IO_L42N_7	K27		
7	IO_L41P_7	L25		
7	IO_L41N_7	L24		
7	IO_L40P_7	H31		
7	IO_L40N_7/VREF_7	H30		
7	IO_L39P_7	J29		
7	IO_L39N_7	J28		
7	IO_L38P_7	M23		
7	IO_L38N_7	M22		
7	IO_L37P_7	G31		
7	IO_L37N_7	G30		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
7	IO_L36P_7	H28		
7	IO_L36N_7	H27		
7	IO_L35P_7	K26		
7	IO_L35N_7	K25		
7	IO_L34P_7	F31		
7	IO_L34N_7/VREF_7	F30		
7	IO_L33P_7	G27		
7	IO_L33N_7	G26		
7	IO_L32P_7	J26		
7	IO_L32N_7	J25		
7	IO_L31P_7	G29		
7	IO_L31N_7	G28		
7	IO_L18P_7	F28	NC	
7	IO_L18N_7	F27	NC	
7	IO_L06P_7	E29		
7	IO_L06N_7	E28		
7	IO_L05P_7	L23		
7	IO_L05N_7	L22		
7	IO_L04P_7	E31		
7	IO_L04N_7/VREF_7	E30		
7	IO_L03P_7	D29		
7	IO_L03N_7	C28		
7	IO_L02P_7	K24		
7	IO_L02N_7	K23		
7	IO_L01P_7/VRN_7	D31		
7	IO_L01N_7/VRP_7	D30		
0	VCCO_0	M19		
0	VCCO_0	M18		
0	VCCO_0	M17		
0	VCCO_0	L20		
0	VCCO_0	L19		
0	VCCO_0	L18		
0	VCCO_0	L17		
0	VCCO_0	E22		
0	VCCO_0	C26		
0	VCCO_0	C19		
1	VCCO_1	M15		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
1	VCCO_1	M14		
1	VCCO_1	M13		
1	VCCO_1	L15		
1	VCCO_1	L14		
1	VCCO_1	L13		
1	VCCO_1	L12		
1	VCCO_1	E10		
1	VCCO_1	C13		
1	VCCO_1	C6		
2	VCCO_2	R12		
2	VCCO_2	R11		
2	VCCO_2	P12		
2	VCCO_2	P11		
2	VCCO_2	N12		
2	VCCO_2	N11		
2	VCCO_2	N3		
2	VCCO_2	M11		
2	VCCO_2	J5		
2	VCCO_2	F3		
3	VCCO_3	AF3		
3	VCCO_3	AC5		
3	VCCO_3	Y11		
3	VCCO_3	W12		
3	VCCO_3	W11		
3	VCCO_3	W3		
3	VCCO_3	V12		
3	VCCO_3	V11		
3	VCCO_3	U12		
3	VCCO_3	U11		
4	VCCO_4	AJ13		
4	VCCO_4	AJ6		
4	VCCO_4	AG10		
4	VCCO_4	AA15		
4	VCCO_4	AA14		
4	VCCO_4	AA13		
4	VCCO_4	AA12		
4	VCCO_4	Y15		
4	VCCO_4	Y14		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
4	VCCO_4	Y13		
5	VCCO_5	AJ26		
5	VCCO_5	AJ19		
5	VCCO_5	AG22		
5	VCCO_5	AA20		
5	VCCO_5	AA19		
5	VCCO_5	AA18		
5	VCCO_5	AA17		
5	VCCO_5	Y19		
5	VCCO_5	Y18		
5	VCCO_5	Y17		
6	VCCO_6	AF29		
6	VCCO_6	AC27		
6	VCCO_6	Y21		
6	VCCO_6	W29		
6	VCCO_6	W21		
6	VCCO_6	W20		
6	VCCO_6	V21		
6	VCCO_6	V20		
6	VCCO_6	U21		
6	VCCO_6	U20		
7	VCCO_7	R21		
7	VCCO_7	R20		
7	VCCO_7	P21		
7	VCCO_7	P20		
7	VCCO_7	N29		
7	VCCO_7	N21		
7	VCCO_7	N20		
7	VCCO_7	M21		
7	VCCO_7	J27		
7	VCCO_7	F29		
N/A	CCLK	AC8		
N/A	PROG_B	J24		
N/A	DONE	AH6		
N/A	M0	AH27		
N/A	M1	AC24		
N/A	M2	AH26		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	TCK	J8		
N/A	TDI	H26		
N/A	TDO	H6		
N/A	TMS	H7		
N/A	PWRDWN_B	AH5		
N/A	HSWAP_EN	H25		
N/A	RSVD	D6		
N/A	VBATT	D5		
N/A	DXP	D27		
N/A	DXN	D26		
N/A	AVCCAUXTX2	B27	NC	
N/A	VTTXPAD2	B28	NC	
N/A	TXNPAD2	A28	NC	
N/A	TXPPAD2	A27	NC	
N/A	GND A2	C27	NC	
N/A	GND A2	C27	NC	
N/A	RXPPAD2	A26	NC	
N/A	RXNPAD2	A25	NC	
N/A	VTRXPAD2	B26	NC	
N/A	AVCCAUXRX2	B25	NC	
N/A	AVCCAUXTX4	B23		
N/A	VTTXPAD4	B24		
N/A	TXNPAD4	A24		
N/A	TXPPAD4	A23		
N/A	GND A4	C24		
N/A	GND A4	C24		
N/A	RXPPAD4	A22		
N/A	RXNPAD4	A21		
N/A	VTRXPAD4	B22		
N/A	AVCCAUXRX4	B21		
N/A	AVCCAUXTX6	B19		
N/A	VTTXPAD6	B20		
N/A	TXNPAD6	A20		
N/A	TXPPAD6	A19		
N/A	GND A6	C20		
N/A	GND A6	C20		
N/A	RXPPAD6	A18		
N/A	RXNPAD6	A17		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	VTRXPAD6	B18		
N/A	AVCCAUXRX6	B17		
N/A	AVCCAUXTX7	B14		
N/A	VTTXPAD7	B15		
N/A	TXNPAD7	A15		
N/A	TXPPAD7	A14		
N/A	GND A7	C12		
N/A	GND A7	C12		
N/A	RXPPAD7	A13		
N/A	RXNPAD7	A12		
N/A	VTRXPAD7	B13		
N/A	AVCCAUXRX7	B12		
N/A	AVCCAUXTX9	B10		
N/A	VTTXPAD9	B11		
N/A	TXNPAD9	A11		
N/A	TXPPAD9	A10		
N/A	GND A9	C8		
N/A	GND A9	C8		
N/A	RXPPAD9	A9		
N/A	RXNPAD9	A8		
N/A	VTRXPAD9	B9		
N/A	AVCCAUXRX9	B8		
N/A	AVCCAUXTX11	B6	NC	
N/A	VTTXPAD11	B7	NC	
N/A	TXNPAD11	A7	NC	
N/A	TXPPAD11	A6	NC	
N/A	GND A11	C5	NC	
N/A	GND A11	C5	NC	
N/A	RXPPAD11	A5	NC	
N/A	RXNPAD11	A4	NC	
N/A	VTRXPAD11	B5	NC	
N/A	AVCCAUXRX11	B4	NC	
N/A	AVCCAUXRX14	AK4	NC	
N/A	VTRXPAD14	AK5	NC	
N/A	RXNPAD14	AL4	NC	
N/A	RXPPAD14	AL5	NC	
N/A	GND A14	AJ5	NC	
N/A	GND A14	AJ5	NC	

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	TXPPAD14	AL6	NC	
N/A	TXNPAD14	AL7	NC	
N/A	VTTXPAD14	AK7	NC	
N/A	AVCCAUXTX14	AK6	NC	
N/A	AVCCAUXRX16	AK8		
N/A	VTRXPAD16	AK9		
N/A	RXNPAD16	AL8		
N/A	RXPPAD16	AL9		
N/A	GND A16	AJ8		
N/A	GND A16	AJ8		
N/A	TXPPAD16	AL10		
N/A	TXNPAD16	AL11		
N/A	VTTXPAD16	AK11		
N/A	AVCCAUXTX16	AK10		
N/A	AVCCAUXRX18	AK12		
N/A	VTRXPAD18	AK13		
N/A	RXNPAD18	AL12		
N/A	RXPPAD18	AL13		
N/A	GND A18	AJ12		
N/A	GND A18	AJ12		
N/A	TXPPAD18	AL14		
N/A	TXNPAD18	AL15		
N/A	VTTXPAD18	AK15		
N/A	AVCCAUXTX18	AK14		
N/A	AVCCAUXRX19	AK17		
N/A	VTRXPAD19	AK18		
N/A	RXNPAD19	AL17		
N/A	RXPPAD19	AL18		
N/A	GND A19	AJ20		
N/A	GND A19	AJ20		
N/A	TXPPAD19	AL19		
N/A	TXNPAD19	AL20		
N/A	VTTXPAD19	AK20		
N/A	AVCCAUXTX19	AK19		
N/A	AVCCAUXRX21	AK21		
N/A	VTRXPAD21	AK22		
N/A	RXNPAD21	AL21		
N/A	RXPPAD21	AL22		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	GND A21	AJ24		
N/A	GND A21	AJ24		
N/A	TXPPAD21	AL23		
N/A	TXNPAD21	AL24		
N/A	VTTXPAD21	AK24		
N/A	AVCCAUXTX21	AK23		
N/A	AVCCAUXRX23	AK25	NC	
N/A	VTRXPAD23	AK26	NC	
N/A	RXNPAD23	AL25	NC	
N/A	RXPPAD23	AL26	NC	
N/A	GND A23	AJ27	NC	
N/A	GND A23	AJ27	NC	
N/A	TXPPAD23	AL27	NC	
N/A	TXNPAD23	AL28	NC	
N/A	VTTXPAD23	AK28	NC	
N/A	AVCCAUXTX23	AK27	NC	
N/A	VCCAUX	AK29		
N/A	VCCAUX	AK16		
N/A	VCCAUX	AK3		
N/A	VCCAUX	AJ30		
N/A	VCCAUX	AJ16		
N/A	VCCAUX	AJ2		
N/A	VCCAUX	T30		
N/A	VCCAUX	T29		
N/A	VCCAUX	T3		
N/A	VCCAUX	T2		
N/A	VCCAUX	C30		
N/A	VCCAUX	C16		
N/A	VCCAUX	C2		
N/A	VCCAUX	B29		
N/A	VCCAUX	B16		
N/A	VCCAUX	B3		
N/A	VCCINT	AA21		
N/A	VCCINT	AA16		
N/A	VCCINT	AA11		
N/A	VCCINT	Y20		
N/A	VCCINT	Y16		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	VCCINT	Y12		
N/A	VCCINT	W19		
N/A	VCCINT	W18		
N/A	VCCINT	W17		
N/A	VCCINT	W16		
N/A	VCCINT	W15		
N/A	VCCINT	W14		
N/A	VCCINT	W13		
N/A	VCCINT	V19		
N/A	VCCINT	V13		
N/A	VCCINT	U19		
N/A	VCCINT	U13		
N/A	VCCINT	T21		
N/A	VCCINT	T20		
N/A	VCCINT	T19		
N/A	VCCINT	T13		
N/A	VCCINT	T12		
N/A	VCCINT	T11		
N/A	VCCINT	R19		
N/A	VCCINT	R13		
N/A	VCCINT	P19		
N/A	VCCINT	P13		
N/A	VCCINT	N19		
N/A	VCCINT	N18		
N/A	VCCINT	N17		
N/A	VCCINT	N16		
N/A	VCCINT	N15		
N/A	VCCINT	N14		
N/A	VCCINT	N13		
N/A	VCCINT	M20		
N/A	VCCINT	M16		
N/A	VCCINT	M12		
N/A	VCCINT	L21		
N/A	VCCINT	L16		
N/A	VCCINT	L11		
N/A	GND	AL30		
N/A	GND	AL29		
N/A	GND	AL16		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	GND	AL3		
N/A	GND	AL2		
N/A	GND	AK31		
N/A	GND	AK30		
N/A	GND	AK2		
N/A	GND	AK1		
N/A	GND	AJ31		
N/A	GND	AJ29		
N/A	GND	AJ23		
N/A	GND	AJ9		
N/A	GND	AJ3		
N/A	GND	AJ1		
N/A	GND	AH28		
N/A	GND	AH16		
N/A	GND	AH4		
N/A	GND	AG27		
N/A	GND	AG5		
N/A	GND	AF26		
N/A	GND	AF19		
N/A	GND	AF13		
N/A	GND	AF6		
N/A	GND	AE25		
N/A	GND	AE16		
N/A	GND	AE7		
N/A	GND	AD29		
N/A	GND	AD24		
N/A	GND	AD8		
N/A	GND	AD3		
N/A	GND	AC23		
N/A	GND	AC9		
N/A	GND	AB29		
N/A	GND	AB22		
N/A	GND	AB16		
N/A	GND	AB10		
N/A	GND	AB3		
N/A	GND	W26		
N/A	GND	W6		
N/A	GND	V18		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	GND	V17		
N/A	GND	V16		
N/A	GND	V15		
N/A	GND	V14		
N/A	GND	U18		
N/A	GND	U17		
N/A	GND	U16		
N/A	GND	U15		
N/A	GND	U14		
N/A	GND	T31		
N/A	GND	T28		
N/A	GND	T25		
N/A	GND	T22		
N/A	GND	T18		
N/A	GND	T17		
N/A	GND	T16		
N/A	GND	T15		
N/A	GND	T14		
N/A	GND	T10		
N/A	GND	T7		
N/A	GND	T4		
N/A	GND	T1		
N/A	GND	R18		
N/A	GND	R17		
N/A	GND	R16		
N/A	GND	R15		
N/A	GND	R14		
N/A	GND	P18		
N/A	GND	P17		
N/A	GND	P16		
N/A	GND	P15		
N/A	GND	P14		
N/A	GND	N26		
N/A	GND	N6		
N/A	GND	K29		
N/A	GND	K22		
N/A	GND	K16		
N/A	GND	K10		

Table 12: BF957 — XC2VP20 and XC2VP50

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	GND	K3		
N/A	GND	J23		
N/A	GND	J9		
N/A	GND	H29		
N/A	GND	H24		
N/A	GND	H8		
N/A	GND	H3		
N/A	GND	G25		
N/A	GND	G16		
N/A	GND	G7		
N/A	GND	F26		
N/A	GND	F19		
N/A	GND	F13		
N/A	GND	F6		
N/A	GND	E27		
N/A	GND	E5		
N/A	GND	D28		
N/A	GND	D16		
N/A	GND	D4		
N/A	GND	C31		
N/A	GND	C29		
N/A	GND	C23		
N/A	GND	C9		
N/A	GND	C3		
N/A	GND	C1		
N/A	GND	B31		
N/A	GND	B30		
N/A	GND	B2		
N/A	GND	B1		
N/A	GND	A30		
N/A	GND	A29		
N/A	GND	A16		
N/A	GND	A3		
N/A	GND	A2		

BF957 Flip-Chip BGA Package Specifications (1.27mm pitch)

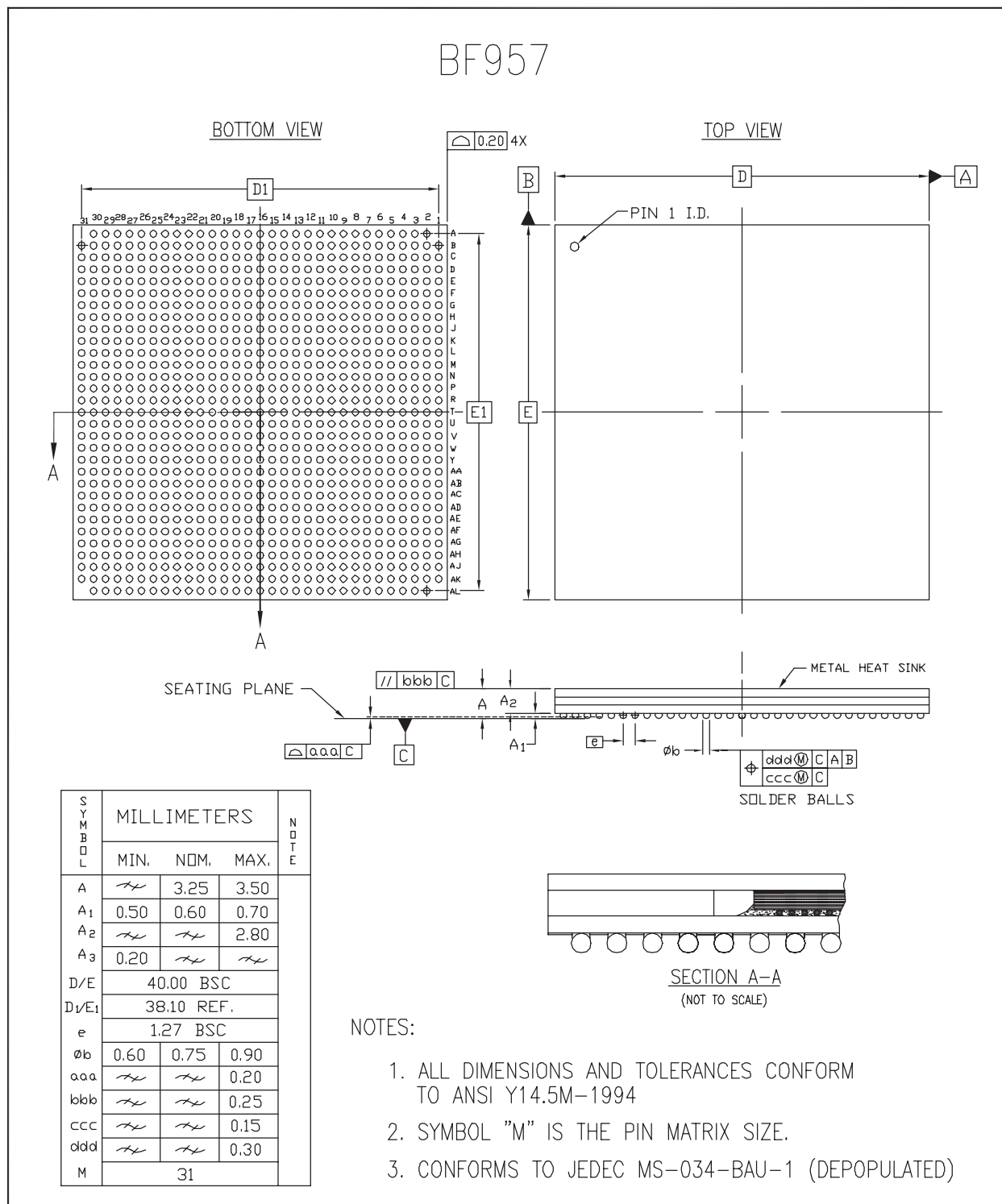


Figure 7: BF957 Flip-Chip BGA Package Specifications

Revision History

This section records the change history for this module of the data sheet.

Date	Version	Revision
1/31/02	1.0	Initial Xilinx release.

Virtex-II Pro Data Sheet Modules

The Virtex-II Pro Data Sheet contains the following modules:

- **Virtex-II Pro™ Platform FPGAs: Introduction and Overview (Module 1)**
- **Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics (Module 3)**
- **Virtex-II Pro™ Platform FPGAs: Functional Description (Module 2)**
- **Virtex-II Pro Platform FPGAs: Pinout Information (Module 4)**

Volume 2: PowerPC[®] 405 Processor

***Virtex-II Pro[™] Platform FPGA
Developer's Kit***

March 2002 Release





The Xilinx logo shown above is a registered trademark of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

"Xilinx" and the Xilinx logo are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, XC5210 are registered Trademarks of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Bencher, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Bencher, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, Nano-Blaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, Rocket I/O, SelectI/O, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM IBM Logo PowerPC PowerPC Logo Blue Logic CoreConnect CodePack

All other trademarks are the property of their respective owners.

Xilinx does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 2002 Xilinx, Inc. All Rights Reserved.

Volume 2(a): PPC405 User Manual

Virtex-II Pro™ Platform FPGA Developer's Kit

March 2002 Release





The Xilinx logo shown above is a registered trademark of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

"Xilinx" and the Xilinx logo are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, XC5210 are registered Trademarks of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Benchner, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Benchner, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, Nano-Blaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, Rocket I/O, SelectI/O, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM IBM Logo PowerPC PowerPC Logo Blue Logic CoreConnect CodePack

All other trademarks are the property of their respective owners.

Xilinx does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 2002 Xilinx, Inc. All Rights Reserved.

About This Book

This document is intended to serve as a stand-alone reference for application and system programmers of the PowerPC® 405D5 processor. It combines information from the following documents:

- *PowerPC 405 Embedded Processor Core User's Manual* published by IBM Corporation (IBM order number SA14-2339-01).
- *The IBM PowerPC Embedded Environment Architectural Specifications for IBM PowerPC Embedded Controllers*, published by IBM Corporation.
- *PowerPC Microprocessor Family: The Programming Environments* published by IBM Corporation (IBM order number G522-0290-01).
- IBM PowerPC Embedded Processors Application Note: *PowerPC 400 Series Caches: Programming and Coherency Issues*.
- IBM PowerPC Embedded Processors Application Note: *PowerPC 40x Watch Dog Timer*.
- IBM PowerPC Embedded Processors Application Note: *Programming Model Differences of the IBM PowerPC 400 Family and 600/700 Family Processors*.

Document Organization

- **Chapter 1, Introduction to the PPC405**, provides a general understanding of the PPC405 as an implementation of the PowerPC embedded-environment architecture. This chapter also contains an overview of the features supported by the PPC405.
- **Chapter 2, Operational Concepts**, introduces the processor operating modes, execution model, synchronization, operand conventions, and instruction conventions.
- **Chapter 3, User Programming Model**, describes the registers and instructions available to application software.
- **Chapter 4, PPC405 Privileged-Mode Programming Model**, introduces the registers and instructions available to system software.
- **Chapter 5, Memory-System Management**, describes the operation of the memory system, including caches. Real-mode storage control is also described in this chapter.
- **Chapter 6, Virtual-Memory Management**, describes virtual-to-physical address translation as supported by the PPC405. Virtual-mode storage control is also described in this chapter.
- **Chapter 7, Exceptions and Interrupts**, provides details of all exceptions recognized by the PPC405 and how software can use the interrupt mechanism to handle exceptions.
- **Chapter 8, Timer Resources**, describes the timer registers and timer-interrupt controls available in the PPC405.
- **Chapter 9, Debugging**, describes the debug resources available to software and hardware debuggers.
- **Chapter 10, Reset and Initialization**, describes the state of the PPC405 following reset.

and the requirements for initializing the processor.

- **Chapter 11, Instruction Set**, provides a detailed description of each instruction supported by the PPC405.
- **Appendix A, Register Summary**, is a reference of all registers supported by the PPC405.
- **Appendix B, Instruction Summary**, lists all instructions sorted by mnemonic, opcode, function, and form. Each entry for an instruction shows its complete encoding. General instruction-set information is also provided.
- **Appendix C, Simplified Mnemonics**, lists the simplified mnemonics recognized by many PowerPC assemblers. These mnemonics provide a shorthand means of specifying frequently-used instruction encodings and can greatly improve assembler code readability.
- **Appendix D, Programming Considerations**, provides information on improving performance of software written for the PPC405.
- **Appendix E, PowerPC® 6xx/7xx Compatibility**, describes the programming model differences between the PPC405 and PowerPC 6xx and 7xx series processors.
- **Appendix F, PowerPC® Book-E Compatibility**, describes the programming model differences between the PPC405 and PowerPC Book-E processors.

Document Conventions

General Conventions

Table 1 lists the general notational conventions used throughout this document.

Table P-1: General Notational Conventions

Convention	Definition
mnemonic	Instruction mnemonics are shown in lower-case bold.
. (period)	Update. When used as a character in an instruction mnemonic, a period (.) means that the instruction updates the condition-register field.
! (exclamation)	In instruction listings, an exclamation (!) indicates the start of a comment.
<i>variable</i>	Variable items are shown in italic.
<optional>	Optional items are shown in angle brackets.
<u>ActiveLow</u>	An overbar indicates an active-low signal.
<i>n</i>	A decimal number.
0xn	A hexadecimal number.
0bn	A binary number.
(<i>rn</i>)	The contents of GPR <i>rn</i> .
(<i>rA</i> 0)	The contents of the register <i>rA</i> , or 0 if the <i>rA</i> instruction field is 0.
cr_bit	Used in simplified mnemonics to specify a CR-bit position (0 to 31) used as an operand.

Table P-1: General Notational Conventions (Continued)

Convention	Definition
cr_field	Used in simplified mnemonics to specify a CR field (0 to 7) used as an operand.
OBJECT _b	A single bit in any object (a register, an instruction, an address, or a field) is shown as a subscripted number or name.
OBJECT _{b:b}	A range of bits in any object (a register, an instruction, an address, or a field).
OBJECT _{b,b,...}	A list of bits in any object (a register, an instruction, an address, or a field).
REGISTER[FIELD]	Fields within any register are shown in square brackets.
REGISTER[FIELD, FIELD ...]	A list of fields in any register.
REGISTER[FIELD:FIELD]	A range of fields in any register.

Instruction Fields

Table 2 lists the instruction fields used in the various instruction formats. They are found in the instruction encodings and pseudocode, and are referred to throughout this document when describing instructions. The table includes the bit locations for the field within the instruction encoding.

Table P-2: Instruction Field Definitions

Field	Location	Description
AA	30	Absolute-address bit (branch instructions). 0—The immediate field represents an address <i>relative</i> to the current instruction address (CIA). The effective address (EA) of the branch is either the sum of the LI field sign-extended to 32 bits and the branch instruction address, or the sum of the BD field sign-extended to 32 bits and the branch instruction address. 1—The immediate field represents an <i>absolute</i> address. The EA of the branch is either the LI field or the BD field, sign-extended to 32 bits.
BD	16:29	An immediate field specifying a 14-bit signed two's-complement branch displacement. This field is concatenated on the right with 0b00 and sign-extended to 32 bits.
BI	11:15	Specifies a bit in the CR used as a source for the condition of a conditional-branch instruction.
BO	6:10	Specifies options for conditional-branch instructions. See Conditional Branch Control , page 403
crbA	11:15	Specifies a bit in the CR used as a source of a CR-logical instruction.
crbB	16:20	Specifies a bit in the CR used as a source of a CR-logical instruction.
crbD	6:10	Specifies a bit in the CR used as a destination of a CR-Logical instruction.

Table P-2: Instruction Field Definitions (Continued)

Field	Location	Description
crfD	6:8	Specifies a field in the CR used as a target in a compare or mcrf instruction.
crfS	11:13	Specifies a field in the CR used as a source in a mcrf instruction.
CRM	12:19	The field mask used to identify CR fields to be updated by the mtrcf instruction.
d	16:31	Specifies a 16-bit signed two's-complement integer displacement for load/store instructions.
DCRF	11:20	A split field used to specify a device control register (DCR). The field is used to form the DCR number (DCRN).
E	16	A single-bit immediate field in the wrtnei instruction specifying the value to be written to the MSR[EE] bit.
LI	6:29	An immediate field specifying a 24-bit signed two's-complement branch displacement. This field is concatenated on the right with 0b00 and sign-extended to 32 bits.
LK	31	Link bit. 0—Do not update the link register (LR). 1—Update the LR with the address of the next instruction.
MB	21:25	Mask begin. Used in rotate-and-mask instructions to specify the beginning bit of a mask.
ME	26:30	Mask end. Used in rotate-and-mask instructions to specify the ending bit of a mask.
NB	16:20	Specifies the number of bytes to move in an immediate-string load or immediate-string store.
OE	21	Enables setting the OV and SO fields in the fixed-point exception register (XER) for extended arithmetic.
OPCD	0:5	Primary opcode. Primary opcodes, in decimal, appear in the instruction format diagrams presented with individual instructions. The OPCODE field name does not appear in instruction descriptions.
rA	11:15	Specifies a GPR source operand and/or destination operand.
rB	16:20	Specifies a GPR source operand.
Rc	31	Record bit. 0—Instruction does not update the CR. 1—Instruction updates the CR to reflect the result of an operation. See Condition Register (CR) , page 397 for a further discussion of how the CR bits are set.
rD	6:10	Specifies a GPR destination operand.
rS	6:10	Specifies a GPR source operand.
SH	16:20	Specifies a shift amount.

Table P-2: Instruction Field Definitions (Continued)

Field	Location	Description
SIMM	16:31	An immediate field used to specify a 16-bit signed-integer value.
SPRF	11:20	A split field used to specify a special purpose register (SPR). The field is used to form the SPR number (SPRN).
TBRF	11:20	A split field used to specify a time-base register (TBR). The field is used to form the TBR number (TBRN).
TO	6:10	Specifies the trap conditions, as defined in the tw and twi instruction descriptions.
UIMM	16:31	An immediate field used to specify a 16-bit unsigned-integer value.
XO	21:30	Extended opcode for instructions <i>without</i> an OE field. Extended opcodes, in decimal, appear in the instruction format diagrams presented with individual instructions. The XO field name does not appear in instruction descriptions.
XO	22:30	Extended opcode for instructions <i>with</i> an OE field. Extended opcodes, in decimal, appear in the instruction format diagrams presented with individual instructions. The XO field name does not appear in instruction descriptions.

Pseudocode Conventions

Table 3 lists additional conventions used primarily in the pseudocode describing the operation of each instruction.

Table P-3: Pseudocode Conventions

Convention	Definition
$\leftarrow$	Assignment
$\wedge$	AND logical operator
$\neg$	NOT logical operator
$\vee$	OR logical operator
$\oplus$	Exclusive-OR (XOR) logical operator
+	Two's-complement addition
-	Two's-complement subtraction, unary minus
$\times$	Multiplication
$\div$	Division yielding a quotient
%	Remainder of an integer division. For example, $(33 \% 32) = 1$.
$\parallel$	Concatenation
$=, \neq$	Equal, not-equal relations
$<, >$	Signed comparison relations
$\underline{<}, \underline{>}$	Unsigned comparison relations
$c_{0:3}$	A four-bit object used to store condition results in compare instructions.

Table P-3: Pseudocode Conventions (Continued)

Convention	Definition
n b	The bit or bit value b is replicated n times.
x	Bit positions that are don't-cares.
CEIL(n)	Least integer $\geq n$.
CIA	Current instruction address. The 32-bit address of the instruction being described by a sequence of pseudocode. This address is used to set the next instruction address (NIA). Does not correspond to any architected register.
DCR(DCRN)	A specific device control register, as indicated by DCRN.
DCRN	The device control register number formed using the split DCRF field in a mfdcr or mtocr instruction.
do	Do loop. "to" and "by" clauses specify incrementing an iteration variable. "while" and "until" clauses specify terminating conditions. Indenting indicates the scope of a loop.
EA	Effective address. The 32-bit address that specifies a location in main storage. Derived by applying indexing or indirect addressing rules to the specified operand.
EXTS(n)	The result of extending n on the left with sign bits.
if...then...else...	Conditional execution: if <i>condition</i> then a else b , where a and b represent one or more pseudocode statements. Indenting indicates the ranges of a and b . If b is null, the else does not appear.
<i>instruction</i> (EA)	An instruction operating on a data-cache block or instruction-cache block associated with an EA.
leave	Leave innermost do-loop or the do-loop specified by the leave statement.
MASK(MB,ME)	Mask having 1's in positions MB through ME (wrapping if MB > ME) and 0's elsewhere.
MS(<i>addr</i> , n)	The number of bytes represented by n at the location in main storage represented by <i>addr</i> .
NIA	Next instruction address. The 32-bit address of the next instruction to be executed. In pseudocode, a successful branch is indicated by assigning a value to NIA. For instructions that do not branch, the NIA is CIA +4.
RESERVE	Reserve bit. Indicates whether a process has reserved a block of storage.
ROTL((RS), n)	Rotate left. The contents of RS are shifted left the number of bits specified by n .
SPR(SPRN)	A specific special-purpose register, as indicated by SPRN.

Table P-3: Pseudocode Conventions (Continued)

Convention	Definition
SPRN	The special-purpose register number formed using the split SPRF field in a mf spr or mt spr instruction
TBR(TBRN)	A specific time-base register, as indicated by TBRN.
TBRN	The time-base register number formed using the split TBRF field in a mftb instruction.

Operator Precedence

Table 4 lists the pseudocode operators and their associativity in descending order of precedence

Table P-4: Operator Precedence

Operators	Associativity
REGISTER _b , REGISTER[FIELD], function evaluation	Left to right
ⁿ b	Right to left
¬, − (unary minus)	Right to left
×, ÷	Left to right
+, −	Left to right
	Left to right
=, ≠, <, >, ≤, ≥	Left to right
∧, ⊕	Left to right
∨	Left to right
←	None

Registers

Table 5 lists the PPC405 registers and their descriptive names.

Table P-5: PPC405 Registers

Register	Descriptive Name
CCR0	Core-configuration register 0
CR	Condition register
CTR	Count register
DAC _n	Data-address compare <i>n</i>
DBCR _n	Debug-control register <i>n</i>
DBSR	Debug-status register
DCCR	Data-cache cacheability register
DCWR	Data-cache write-through register

Table P-5: PPC405 Registers (Continued)

Register	Descriptive Name
DEAR	Data-error address register
DVC n	Data-value compare n
ESR	Exception-syndrome register
EVPR	Exception-vector prefix register
GPR	General-purpose register. Specific GPRs are identified using the notational convention rn (see below)
IAC n	Instruction-address compare n
ICCR	Instruction-cache cacheability register
ICDBDR	Instruction-cache debug-data register
LR	Link register
MSR	Machine-state register
PID	Process ID
PIT	Programmable-interval timer
PVR	Processor-version register
rn	Specifies GPR n (r15, for example)
SGR	Storage-guarded register
SLER	Storage little-endian register
SPRG n	SPR general-purpose register n
SRR n	Save/restore register n
SU0R	Storage user-defined 0 register
TBL	Time-base lower
TBU	Time-base upper
TCR	Timer-control register
TSR	Timer-status register
USPRG n	User SPR general-purpose register n
XER	Fixed-point exception register
ZPR	Zone-protection register

Terms

atomic access

A memory access that attempts to read from and write to the same address uninterrupted by other accesses to that address. The term refers to the fact that such transactions are indivisible.

big endian

A memory byte ordering where the address of an item corresponds to the most-significant byte.

<i>Book-E</i>	An version of the PowerPC architecture designed specifically for embedded applications.
<i>cache block</i>	Synonym for <i>cacheline</i> .
<i>cacheline</i>	A portion of a cache array that contains a copy of contiguous system-memory addresses. Cachelines are 32-bytes long and aligned on a 32-byte address.
<i>clear</i>	To write a bit value of 0.
<i>cache set</i>	Synonym for <i>congruence class</i> .
<i>congruence class</i>	A collection of cachelines with the same index.
<i>dirty</i>	An indication that cache information is more recent than the copy in memory.
<i>doubleword</i>	Eight bytes, or 64 bits.
<i>effective address</i>	The untranslated memory address as seen by a program.
<i>exception</i>	An abnormal event or condition that requires the processor's attention. They can be caused by instruction execution or an external device. The processor records the occurrence of an exception and they often cause an <i>interrupt</i> to occur.
<i>fill buffer</i>	A buffer that receives and sends data and instructions between the processor and PLB. It is used when cache misses occur and when access to non-cacheable memory occurs.
<i>flush</i>	A cache or TLB operation that involves writing back a modified entry to memory, followed by an invalidation of the entry.
<i>GB</i>	Gigabyte, or one-billion bytes.
<i>halfword</i>	Two bytes, or 16 bits.
<i>hit</i>	For cache arrays and TLB arrays, an indication that requested information exists in the accessed array.
<i>interrupt</i>	The process of stopping the currently executing program so that an exception can be handled.
<i>invalidate</i>	A cache or TLB operation that causes an entry to be marked as invalid. An invalid entry can be subsequently replaced.
<i>KB</i>	Kilobyte, or one-thousand bytes.
<i>line buffer</i>	A buffer located in the cache array that can temporarily hold the contents of an entire cacheline. It is loaded with the contents of a cacheline when a cache hit occurs.
<i>little endian</i>	A memory byte ordering where the address of an item corresponds to the least-significant byte.
<i>logical address</i>	Synonym for <i>effective address</i> .
<i>MB</i>	Megabyte, or one-million bytes.
<i>memory</i>	Collectively, cache memory and system memory.
<i>miss</i>	For cache arrays and TLB arrays, an indication that requested information does not exist in the accessed array.

<i>OEA</i>	The PowerPC operating-environment architecture, which defines the memory-management model, supervisor-level registers and instructions, synchronization requirements, the exception model, and the time-base resources as seen by supervisor programs.
<i>on chip</i>	In system-on-chip implementations, this indicates on the same chip as the processor core, but external to the processor core.
<i>pending</i>	As applied to interrupts, this indicates that an exception occurred, but the interrupt is disabled. The interrupt occurs when it is later enabled.
<i>physical address</i>	The address used to access physically-implemented memory. This address can be translated from the effective address. When address translation is not used, this address is equal to the effective address.
<i>PLB</i>	Processor local bus.
<i>privileged mode</i>	The operating mode typically used by system software. Privileged operations are allowed and software can access all registers and memory.
<i>process</i>	A program (or portion of a program) and any data required for the program to run.
<i>problem state</i>	Synonym for <i>user mode</i> .
<i>real address</i>	Synonym for <i>physical address</i> .
<i>scalar</i>	Individual data objects and instructions. Scalars are of arbitrary size.
<i>set</i>	To write a bit value of 1.
<i>sticky</i>	A bit that can be set by software, but cleared only by the processor. Alternatively, a bit that can be cleared by software, but set only by the processor.
<i>string</i>	A sequence of consecutive bytes.
<i>supervisor state</i>	Synonym for <i>privileged mode</i> .
<i>system memory</i>	Physical memory installed in a computer system external to the processor core, such RAM, ROM, and flash.
<i>tag</i>	As applied to caches, a set of address bits used to uniquely identify a specific cacheline within a congruence class. As applied to TLBs, a set of address bits used to uniquely identify a specific entry within the TLB.
<i>UIISA</i>	The PowerPC user instruction-set architecture, which defines the base user-level instruction set, registers, data types, the memory model, the programming model, and the exception model as seen by user programs.
<i>user mode</i>	The operating mode typically used by application software. Privileged operations are not allowed in user mode, and software can access a restricted set of registers and memory.

<i>VEA</i>	The PowerPC virtual-environment architecture, which defines a multi-access memory model, the cache model, cache-control instructions, and the time-base resources as seen by user programs.
<i>virtual address</i>	An intermediate address used to translate an effective address into a physical address. It consists of a process ID and the effective address. It is only used when address translation is enabled.
<i>word</i>	Four bytes, or 32 bits.

Additional Reading

In addition to the source documents listed on [page 347](#), the following documents contain additional information of potential interest to readers of this manual:

- *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, IBM 5/1994. Published by Morgan Kaufmann Publishers, Inc. San Francisco (ASIN: 1558603166).
- *Book E: Enhanced PowerPC Architecture*, IBM 3/2000.
- *The PowerPC Compiler Writer's Guide*, IBM 1/1996. Published by Warthman Associates, Palo Alto, CA (ISBN 0-9649654-0-2).
- *Optimizing PowerPC Code : Programming the PowerPC Chip in Assembly Language*, by Gary Kacmarcik (ASIN: 0201408392)
- *PowerPC Programming Pocket Book*, by Steve Heath (ISBN 0750621117).
- *Computer Architecture: A Quantitative Approach*, by John L. Hennessy and David A. Patterson.
-

Introduction to the PPC405

The PPC405 is a 32-bit implementation of the *PowerPC® embedded-environment architecture* that is derived from the PowerPC architecture. Specifically, the PPC405 is an embedded PowerPC 405D5 processor core.

The PowerPC architecture provides a software model that ensures compatibility between implementations of the PowerPC family of microprocessors. The PowerPC architecture defines parameters that guarantee compatible processor implementations at the application-program level, allowing broad flexibility in the development of derivative PowerPC implementations that meet specific market requirements.

This chapter provides an overview of the PowerPC architecture and an introduction to the features of the PPC405 core.

PowerPC Architecture Overview

The PowerPC architecture is a 64-bit architecture with a 32-bit subset. The material in this document only covers aspects of the 32-bit architecture implemented by the PPC405.

In general, the PowerPC architecture defines the following:

- Instruction set
- Programming model
- Memory model
- Exception model
- Memory-management model
- Time-keeping model

Instruction Set

The *instruction set* specifies the types of instructions (such as load/store, integer arithmetic, and branch instructions), the specific instructions, and the encoding used for the instructions. The instruction set definition also specifies the addressing modes used for accessing memory.

Programming Model

The *programming model* defines the register set and the memory conventions, including details regarding the bit and byte ordering, and the conventions for how data are stored.

Memory Model

The *memory model* defines the address-space size and how it is subdivided into pages. It also defines attributes for specifying memory-region cacheability, byte ordering (big-endian or little-endian), coherency, and protection.

Exception Model

The *exception model* defines the set of exceptions and the conditions that can cause those exceptions. The model specifies exception characteristics, such as whether they are precise or imprecise, synchronous or asynchronous, and maskable or non-maskable. The model defines the exception vectors and a set of registers used when interrupts occur as a result of an exception. The model also provides memory space for implementation-specific exceptions.

Memory-Management Model

The *memory-management model* defines how memory is partitioned, configured, and protected. The model also specifies how memory translation is performed, defines special memory-control instructions, and specifies other memory-management characteristics.

Time-Keeping Model

The *time-keeping model* defines resources that permit the time of day to be determined and the resources and mechanisms required for supporting timer-related exceptions.

PowerPC Architecture Levels

These above aspects of the PowerPC architecture are defined at three levels. This layering provides flexibility by allowing degrees of software compatibility across a wide range of implementations. For example, an implementation such as an embedded controller can support the user instruction set, but not the memory management, exception, and cache models where it might be impractical to do so.

The three levels of the PowerPC architecture are defined in [Table 1-1](#).

Table 1-1: Three Levels of PowerPC Architecture

User Instruction-Set Architecture (UISA)	Virtual Environment Architecture (VEA)	Operating Environment Architecture (OEA)
- Defines the architecture level to which user-level (sometimes referred to as problem state) software should conform - Defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions, exception model as seen by user programs, memory model, and the programming model Note: All PowerPC implementations adhere to the UISA.	- Defines additional user-level functionality that falls outside typical user-level software requirements - Describes the memory model for an environment in which multiple devices can access memory - Defines aspects of the cache model and cache-control instructions - Defines the time-base resources from a user-level perspective Note: Implementations that conform to the VEA level are guaranteed to conform to the UISA level.	- Defines supervisor-level resources typically required by an operating system - Defines the memory-management model, supervisor-level registers, synchronization requirements, and the exception model - Defines the time-base resources from a supervisor-level perspective Note: Implementations that conform to the OEA level are guaranteed to conform to the UISA and VEA levels.

The PowerPC architecture requires that all PowerPC implementations adhere to the UISA, offering compatibility among all PowerPC application programs. However, different versions of the VEA and OEA are permitted.

Embedded applications written for the PPC405 are compatible with other PowerPC implementations. Privileged software generally is not compatible. The migration of

privileged software from the PowerPC architecture to the PPC405 is in many cases straightforward because of the simplifications made by the PowerPC embedded-environment architecture. Software developers who are concerned with cross-compatibility of privileged software between the PPC405 and other PowerPC implementations should refer to **Appendix E, PowerPC® 6xx/7xx Compatibility**.

Latitude Within the PowerPC Architecture Levels

Although the PowerPC architecture defines parameters necessary to ensure compatibility among PowerPC processors, it also allows a wide range of options for individual implementations. These are:

- Some resources are optional, such as certain registers, bits within registers, instructions, and exceptions.
- Implementations can define additional privileged special-purpose registers (SPRs), exceptions, and instructions to meet special system requirements, such as power management in processors designed for very low-power operation.
- Implementations can define many operating parameters. For example, the PowerPC architecture can define the possible condition causing an alignment exception. A particular implementation can choose to solve the alignment problem without causing an exception.
- Processors can implement any architectural resource or instruction with assistance from software (that is, they can trap and emulate) as long as the results (aside from performance) are identical to those specified by the architecture. In this case, a complete implementation requires both hardware and software.
- Some parameters are defined at one level of the architecture and defined more specifically at another. For example, the UISA defines conditions that can cause an alignment exception and the OEA specifies the exception itself.

Features Not Defined by the PowerPC Architecture

Because flexibility is an important feature of the PowerPC architecture, many aspects of processor design (typically relating to the hardware implementation) are not defined, including the following:

System-Bus Interface

Although many implementations can share similar interfaces, the PowerPC architecture does not define individual signals or the bus protocol. For example, the OEA allows each implementation to specify the signal or signals that trigger a machine-check exception.

Cache Design

The PowerPC architecture does not define the size, structure, replacement algorithm, or mechanism used for maintaining cache coherency. The PowerPC architecture supports, but does not require, the use of separate instruction and data caches.

Execution Units

The PowerPC architecture is a RISC architecture, and as such has been designed to facilitate the design of processors that use pipelining and parallel execution units to maximize instruction throughput. However, the PowerPC architecture does not define the internal hardware details of an implementation. For example, one processor might implement two units dedicated to executing integer-arithmetic instructions and another might implement a single unit for executing all integer instructions.

Other Internal Microarchitecture Issues

The PowerPC architecture does not specify the execution unit responsible for executing a particular instruction. The architecture does not define details regarding the instruction-fetch mechanism, how instructions are decoded and dispatched, and how results are written to registers. Dispatch and write-back can occur in-order or out-of-order. Although

the architecture specifies certain registers, such as the GPRs and FPRs, implementations can use register renaming or other schemes to reduce the impact of data dependencies and register contention.

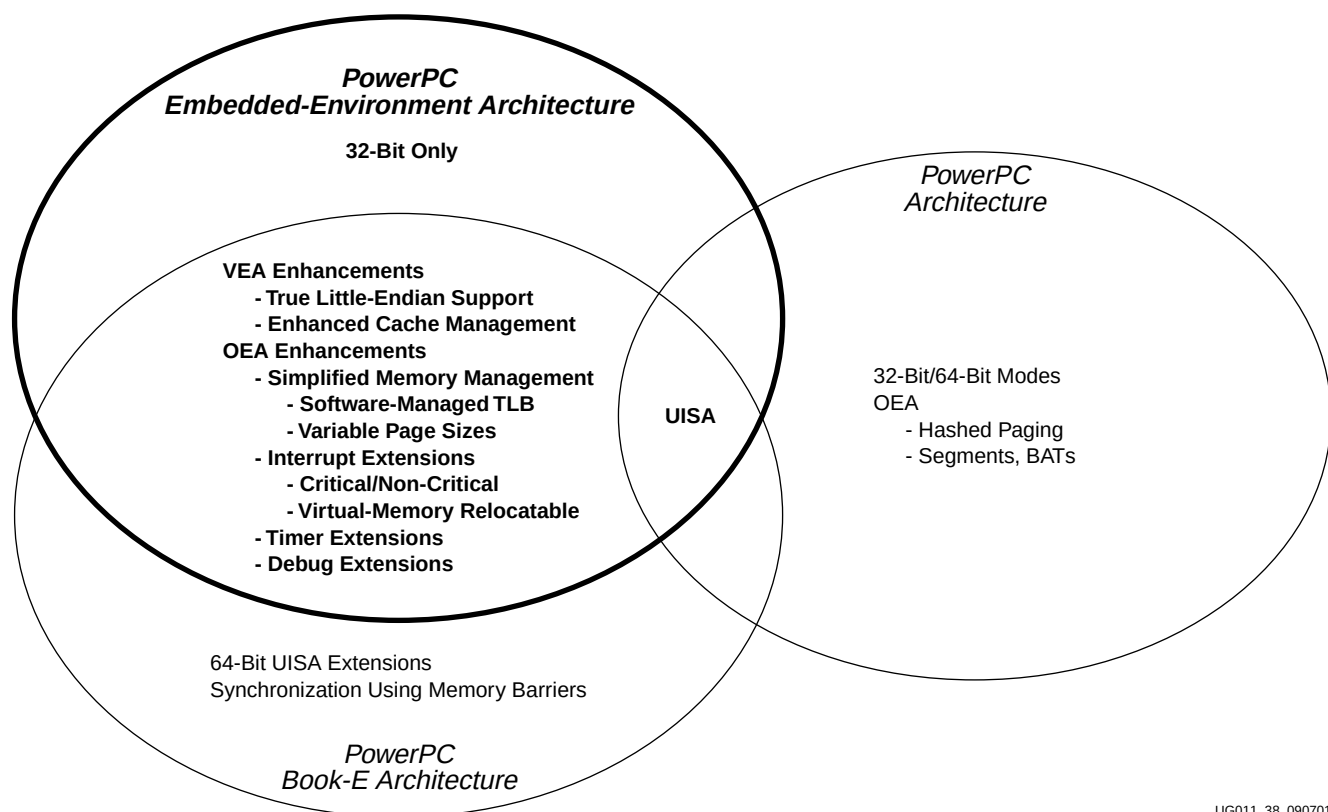
Implementation-Specific Registers

Each implementation can have its own unique set of implementation registers that are not defined by the architecture.

PowerPC Embedded-Environment Architecture

The PowerPC embedded-environment architecture is optimized for embedded controllers. This architecture is a forerunner to the PowerPC Book-E architecture. The PowerPC embedded-environment architecture provides an alternative definition for certain features specified by the PowerPC VEA and OIA. Implementations that adhere to the PowerPC embedded-environment architecture also adhere to the PowerPC UISA. PowerPC embedded-environment processors are 32-bit only implementations and thus do not include the special 64-bit extensions to the PowerPC UISA. Also, floating-point support can be provided either in hardware or software by PowerPC embedded-environment processors.

Figure 1-1 shows the relationship between the PowerPC embedded-environment architecture, the PowerPC architecture, and the PowerPC Book-E architecture.



UG011_38_090701

Figure 1-1: Relationship of PowerPC Architectures

The PowerPC embedded-environment architecture features:

- Memory management optimized for embedded software environments.
- Cache-management instructions for optimizing performance and memory control in complex applications that are graphically and numerically intensive.
- Storage attributes for controlling memory-system behavior.

- Special-purpose registers for controlling the use of debug resources, timer resources, interrupts, real-mode storage attributes, memory-management facilities, and other architected processor resources.
- A device-control-register address space for managing on-chip peripherals such as memory controllers.
- A dual-level interrupt structure and interrupt-control instructions.
- Multiple timer resources.
- Debug resources that enable hardware-debug and software-debug functions such as instruction breakpoints, data breakpoints, and program single-stepping.

Virtual Environment

The virtual environment defines architectural features that enable application programs to create or modify code, to manage storage coherency, and to optimize memory-access performance. It defines the cache and memory models, the timekeeping resources from a user perspective, and resources that are accessible in user mode but are primarily used by system-library routines. The following summarizes the virtual-environment features of the PowerPC embedded-environment architecture:

- Storage model:
 - Storage-control instructions as defined in the PowerPC virtual-environment architecture. These instructions are used to manage instruction caches and data caches, and for synchronizing and ordering instruction execution.
 - Storage attributes for controlling memory-system behavior. These are: write-through, cacheability, memory coherence (optional), guarded, and endian.
 - Operand-placement requirements and their effect on performance.
- The time-base function as defined by the PowerPC virtual-environment architecture, for user-mode read access to the 64-bit time base.

Operating Environment

The operating environment describes features of the architecture that enable operating systems to allocate and manage storage, to handle errors encountered by application programs, to support I/O devices, and to provide operating-system services. It specifies the resources and mechanisms that require privileged access, including the memory-protection and address-translation mechanisms, the exception-handling model, and privileged timer resources. **Table 1-2** summarizes the operating-environment features of the PowerPC embedded-environment architecture.

Table 1-2: Operating-Environment Features of the PowerPC Embedded-Environment Architecture

Operating Environment	Features
Register model	- Privileged special-purpose registers (SPRs) and instructions for accessing those registers - Device control registers (DCRs) and instructions for accessing those registers
Storage model	- Privileged cache-management instructions - Storage-attribute controls - Address translation and memory protection - Privileged TLB-management instructions
Exception model	- Dual-level interrupt structure supporting various exception types - Specification of interrupt priorities and masking - Privileged SPRs for controlling and handling exceptions - Interrupt-control instructions - Specification of how partially executed instructions are handled when an interrupt occurs
Debug model	- Privileged SPRs for controlling debug modes and debug events - Specification for seven types of debug events - Specification for allowing a debug event to cause a reset - The ability of the debug mechanism to freeze the timer resources
Time-keeping model	64-bit time base 32-bit decremter (the programmable-interval timer) - Three timer-event interrupts: - Programmable-interval timer (PIT) - Fixed-interval timer (FIT) - Watchdog timer (WDT) - Privileged SPRs for controlling the timer resources - The ability to freeze the timer resources using the debug mechanism
Synchronization requirements	- Requirements for special registers and the TLB - Requirements for instruction fetch and for data access - Specifications for context synchronization and execution synchronization
Reset and initialization requirements	- Specification for two internal mechanisms that can cause a reset: - Debug-control register (DBCR) - Timer-control register (TCR) - Contents of processor resources after a reset - The software-initialization requirements, including an initialization code example

PowerPC Book-E Architecture

The PowerPC Book-E architecture extends the capabilities introduced in the PowerPC embedded-environment architecture. Although not a PowerPC Book-E implementation, many of the features available in the 32-bit subset of the PowerPC Book-E architecture are available in the PPC405. The PowerPC Book-E architecture and the PowerPC embedded-environment architecture differ in the following general ways:

- 64-bit addressing and 64-bit operands are available. Unlike 64-bit mode in the PowerPC UIISA, 64-bit support in PowerPC Book-E architecture is non-modal and instead defines new 64-bit instructions and flags.
- Real mode is eliminated, and the memory-management unit is active at all times. The elimination of real mode results in the elimination of real-mode storage-attribute registers.
- Memory synchronization requirements are changed in the architecture and a memory-barrier instruction is introduced.
- A small number of new instructions are added to the architecture and several instructions are removed.
- Several SPR addresses and names are changed in the architecture, as are the assignment and meanings of some bits within certain SPRs.

Embedded applications written for the PPC405 are compatible with PowerPC Book-E implementations. Privileged software is, in general, not compatible, but the differences are relatively minor. Software developers who are concerned with cross-compatibility of privileged software between the PPC405 and PowerPC Book-E implementations should refer to [Appendix F, PowerPC® Book-E Compatibility](#).

PPC405 Features

The PPC405 processor core is an implementation of the PowerPC embedded-environment architecture. The processor provides fixed-point embedded applications with high performance at low power consumption. It is compatible with the PowerPC UIISA. Much of the PPC405 VEA and OEA support is also available in implementations of the PowerPC Book-E architecture. Key features of the PPC405 include:

- A fixed-point execution unit fully compliant with the PowerPC UIISA:
 - 32-bit architecture, containing thirty-two 32-bit general purpose registers (GPRs).
- PowerPC embedded-environment architecture extensions providing additional support for embedded-systems applications:
 - True little-endian operation
 - Flexible memory management
 - Multiply-accumulate instructions for computationally intensive applications
 - Enhanced debug capabilities
 - 64-bit time base
 - 3 timers: programmable interval timer (PIT), fixed interval timer (FIT), and watchdog timer (All are synchronous with the time base)
- Performance-enhancing features, including:
 - Static branch prediction
 - Five-stage pipeline with single-cycle execution of most instructions, including loads and stores
 - Multiply-accumulate instructions
 - Hardware multiply/divide for faster integer arithmetic (4-cycle multiply, 35-cycle divide)
 - Enhanced string and multiple-word handling

- Support for unaligned loads and unaligned stores to cache arrays, main memory, and on-chip memory (OCM)
- Minimized interrupt latency
- Integrated instruction-cache:
 - 16 KB, 2-way set associative
 - Eight words (32 bytes) per cacheline
 - Fetch line buffer
 - Instruction-fetch hits are supplied from the fetch line buffer
 - Programmable prefetch of next-sequential line into the fetch line buffer
 - Programmable prefetch of non-cacheable instructions: full line (eight words) or half line (four words)
 - Non-blocking during fetch line fills
- Integrated data-cache:
 - 16 KB, 2-way set associative
 - Eight words (32 bytes) per cacheline
 - Read and write line buffers
 - Load and store hits are supplied from/to the line buffers
 - Write-back and write-through support
 - Programmable load and store cacheline allocation
 - Operand forwarding during cacheline fills
 - Non-blocking during cacheline fills and flushes
- Support for on-chip memory (OCM) that can provide memory-access performance identical to a cache hit
- Flexible memory management:
 - Translation of the 4 GB logical-address space into the physical-address space
 - Independent control over instruction translation and protection, and data translation and protection
 - Page-level access control using the translation mechanism
 - Software control over the page-replacement strategy
 - Write-through, cacheability, user-defined 0, guarded, and endian (WIU0GE) storage-attribute control for each virtual-memory region
 - WIU0GE storage-attribute control for thirty-two 128 MB regions in real mode
 - Additional protection control using zones
- Enhanced debug support with logical operators:
 - Four instruction-address compares
 - Two data-address compares
 - Two data-value compares
 - JTAG instruction for writing into the instruction cache
 - Forward and backward instruction tracing
- Advanced power management support

Privilege Modes

Software running on the PPC405 can do so in one of two privilege modes: privileged and user. The privilege modes supported by the PPC405 are described in **Processor Operating Modes**, [page 379](#).

Privileged Mode

Privileged mode allows programs to access all registers and execute all instructions supported by the processor. Normally, the operating system and low-level device drivers operate in this mode.

User Mode

User mode restricts access to some registers and instructions. Normally, application programs operate in this mode.

Address Translation Modes

The PPC405 also supports two modes of address translation: real and virtual. Refer to **Chapter 6, Virtual-Memory Management**, for more information on address translation.

Real Mode

In *real mode*, programs address physical memory directly.

Virtual Mode

In *virtual mode*, programs address virtual memory and virtual-memory addresses are translated by the processor into physical-memory addresses. This allows programs to access much larger address spaces than might be implemented in the system.

Addressing Modes

Whether the PPC405 is running in real mode or virtual mode, data addressing is supported by the load and store instructions using one of the following addressing modes:

- Register-indirect with immediate index—A base address is stored in a register, and a displacement from the base address is specified as an immediate value in the instruction.
- Register-indirect with index—A base address is stored in a register, and a displacement from the base address is stored in a second register.
- Register indirect—The data address is stored in a register.

Instructions that use the two indexed forms of addressing also allow for automatic updates to the base-address register. With these instruction forms, the new data address is calculated, used in the load or store data access, and stored in the base-address register.

The data-addressing modes are described in **Operand-Address Calculation**, [page 414](#).

With sequential-instruction execution, the next-instruction address is calculated by adding four bytes to the current-instruction address. In the case of branch instructions, however, the next-instruction address is determined using one of four branch-addressing modes:

- Branch to relative—The next-instruction address is at a location relative to the current-instruction address.
- Branch to absolute—The next-instruction address is at an absolute location in memory.
- Branch to link register—The next-instruction address is stored in the link register.
- Branch to count register—The next-instruction address is stored in the count register.

The branch-addressing modes are described in **Branch-Target Address Calculation**, [page 408](#).

Data Types

PPC405 instructions support byte, halfword, and word operands. Multiple-word operands are supported by the load/store multiple instructions and byte strings are supported by

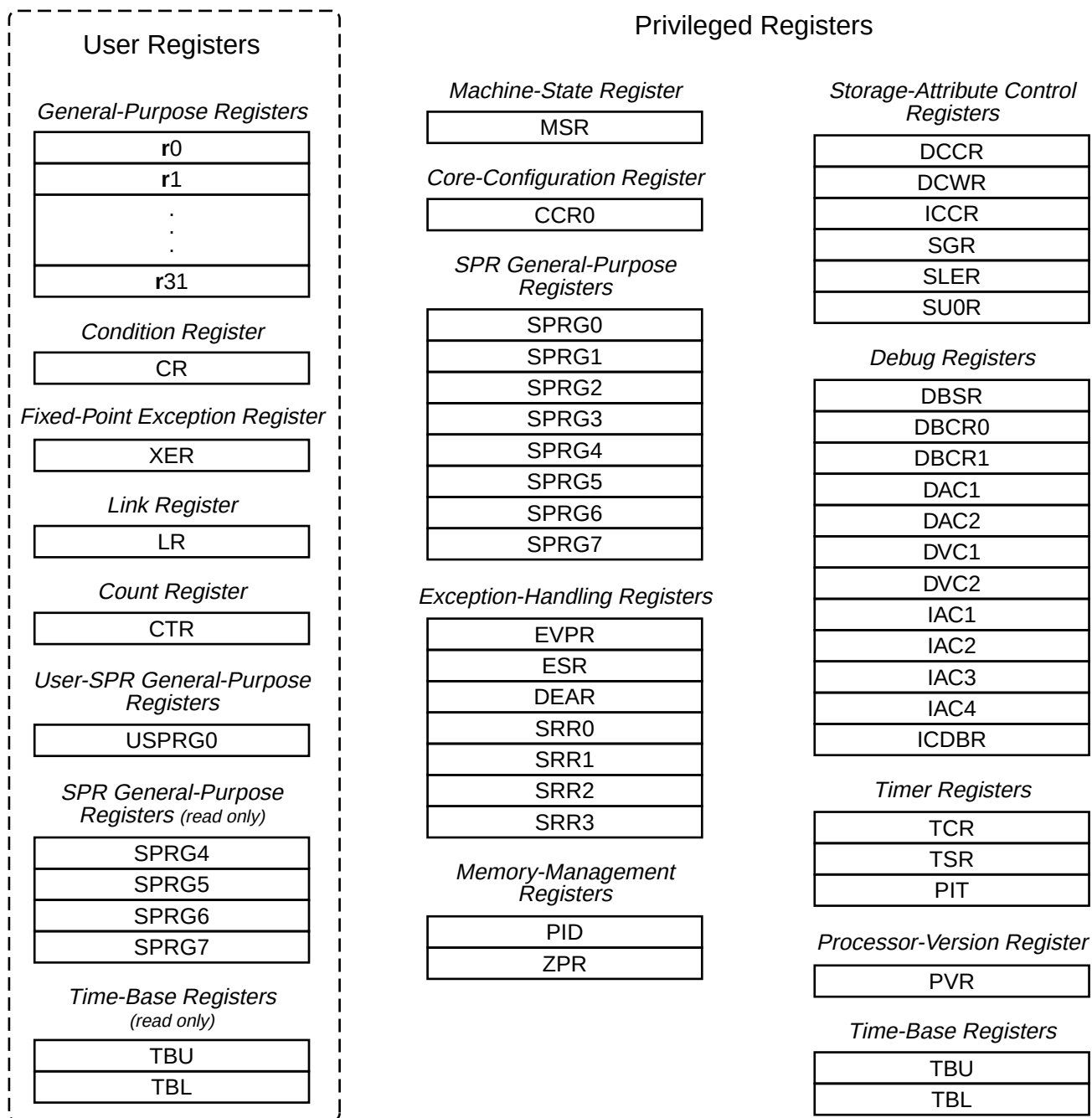
the load/store string instructions. Integer data are either signed or unsigned, and signed data is represented using two's-complement format.

The address of a multi-byte operand is determined using the lowest memory address occupied by that operand. For example, if the four bytes in a word operand occupy addresses 4, 5, 6, and 7, the word address is 4. The PPC405 supports both big-endian (an operand's *most-significant* byte is at the lowest memory address) and little-endian (an operand's *least-significant* byte is at the lowest memory address) addressing.

See **Operand Conventions**, [page 383](#), for more information on the supported data types and byte ordering.

Register Set Summary

[Figure 1-2, page 369](#) shows the registers contained in the PPC405. Descriptions of the registers are in the following sections.



UG011_51_033101

Figure 1-2: PPC405 Registers

General-Purpose Registers

The processor contains thirty-two 32-bit *general-purpose registers* (GPRs), identified as r0 through r31. The contents of the GPRs are read from memory using load instructions and written to memory using store instructions. Computational instructions often read operands from the GPRs and write their results in GPRs. Other instructions move data between the GPRs and other registers. GPRs can be accessed by all software. See **General-Purpose Registers (GPRs)**, page 396, for more information.

Special-Purpose Registers

The processor contains a number of 32-bit *special-purpose registers* (SPRs). SPRs provide access to additional processor resources, such as the count register, the link register, debug resources, timers, interrupt registers, and others. Most SPRs are accessed only by privileged software, but a few, such as the count register and link register, are accessed by all software. See **User Registers**, [page 395](#), and **Privileged Registers**, [page 465](#) for more information.

Machine-State Register

The 32-bit *machine-state register* (MSR) contains fields that control the operating state of the processor. This register can be accessed only by privileged software. See **Machine-State Register**, [page 467](#), for more information.

Condition Register

The 32-bit *condition register* (CR) contains eight 4-bit fields, CR0–CR7. The values in the CR fields can be used to control conditional branching. Arithmetic instructions can set CR0 and compare instructions can set any CR field. Additional instructions are provided to perform logical operations and tests on CR fields and bits within the fields. The CR can be accessed by all software. See **Condition Register (CR)**, [page 397](#), for more information.

Device Control Registers

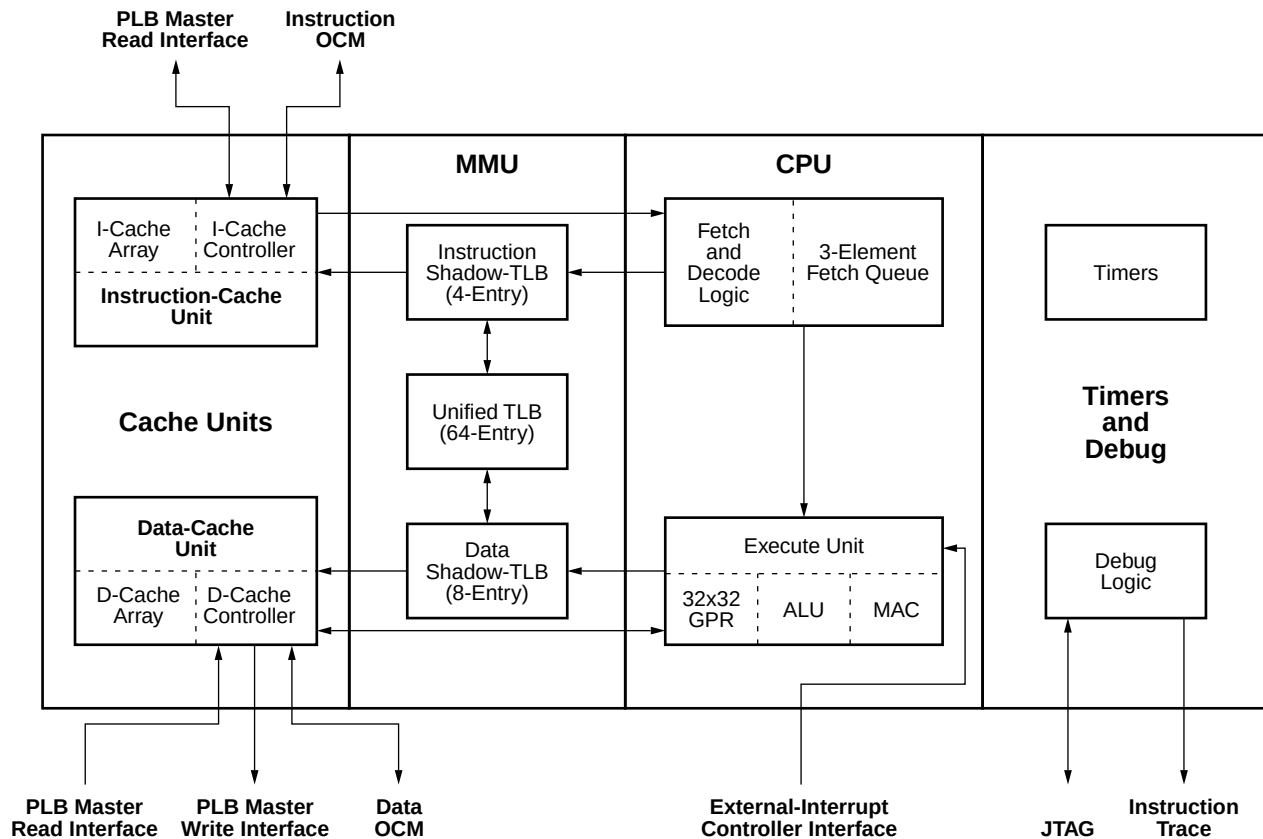
The 32-bit *device control registers* (not shown) are used to configure, control, and report status for various external devices that are not part of the PPC405 processor. Although the DCRs are not part of the PPC405 implementation, they are accessed using the **mtdcr** and **mfdcr** instructions. The DCRs can be accessed only by privileged software. See the **PPC405 Processor Block Manual** for more information on implementing DCRs.

PPC405 Organization

As shown in [Figure 1-3](#), the PPC405 processor contains the following elements:

- A 5-stage pipeline consisting of fetch, decode, execute, write-back, and load write-back stages
- A virtual-memory-management unit that supports multiple page sizes and a variety of storage-protection attributes and access-control options
- Separate instruction-cache and data-cache units
- Debug support, including a JTAG interface
- Three programmable timers

The following sections provide an overview of each element.



UG011_29_033101

Figure 1-3: PPC405 Organization

Central-Processing Unit

The PPC405 central-processing unit (CPU) implements a 5-stage instruction pipeline consisting of fetch, decode, execute, write-back, and load write-back stages.

The fetch and decode logic sends a steady flow of instructions to the execute unit. All instructions are decoded before they are forwarded to the execute unit. Instructions are queued in the fetch queue if execution stalls. The fetch queue consists of three elements: two prefetch buffers and a decode buffer. If the prefetch buffers are empty instructions flow directly to the decode buffer.

Up to two branches are processed simultaneously by the fetch and decode logic. If a branch cannot be resolved prior to execution, the fetch and decode logic predicts how that branch is resolved, causing the processor to speculatively fetch instructions from the predicted path. Branches with negative-address displacements are predicted as taken, as are branches that do not test the condition register or count register. The default prediction can be overridden by software at assembly or compile time. This capability is described further in **Branch Prediction**, page 406.

The PPC405 has a single-issue execute unit containing the general-purpose register file (GPR), arithmetic-logic unit (ALU), and the multiply-accumulate unit (MAC). The GPRs consist of thirty-two 32-bit registers that are accessed by the execute unit using three read ports and two write ports. During the decode stage, data is read out of the GPRs for use by the execute unit. During the write-back stage, results are written to the GPR. The use of five read/write ports on the GPRs allows the processor to execute load/store operations in parallel with ALU and MAC operations.

The execute unit supports all 32-bit PowerPC UISA integer instructions in hardware, and is compliant with the PowerPC embedded-environment architecture specification. Floating-point operations are not supported.

The MAC unit supports implementation-specific multiply-accumulate instructions and multiply-halfword instructions. MAC instructions operate on either signed or unsigned 16-bit operands, and they store their results in a 32-bit GPR. These instructions can produce results using either modulo arithmetic or saturating arithmetic. All MAC instructions have a single cycle throughput. See **Multiply-Accumulate Instruction-Set Extensions**, page 441 for more information.

Exception Handling Logic

Exceptions are divided into two classes: critical and noncritical. The PPC405 CPU services exceptions caused by error conditions, the internal timers, debug events, and the external interrupt controller (EIC) interface. Across the two classes, a total of 19 possible exceptions are supported, including the two provided by the EIC interface.

Each exception class has its own pair of save/restore registers. SRR0 and SRR1 are used for noncritical interrupts, and SRR2 and SRR3 are used for critical interrupts. The exception-return address and the machine state are written to these registers when an exception occurs, and they are automatically restored when an interrupt handler exits using the return-from-interrupt (**rfi**) or return-from critical-interrupt (**rfci**) instruction. Use of separate save/restore registers allows the PPC405 to handle critical interrupts independently of noncritical interrupts.

See **Chapter 7, Exceptions and Interrupts**, for information on exception handling in the PPC405.

Memory Management Unit

The PPC405 supports 4 GB of flat (non-segmented) address space. The memory-management unit (MMU) provides address translation, protection functions, and storage-attribute control for this address space. The MMU supports demand-paged virtual memory using multiple page sizes of 1 KB, 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB and 16 MB. Multiple page sizes can improve memory efficiency and minimize the number of TLB misses. When supported by system software, the MMU provides the following functions:

- Translation of the 4 GB logical-address space into a physical-address space.
- Independent enabling of instruction translation and protection from that of data translation and protection.
- Page-level access control using the translation mechanism.
- Software control over the page-replacement strategy.
- Additional protection control using zones.
- Storage attributes for cache policy and speculative memory-access control.

The translation look-aside buffer (TLB) is used to control memory translation and protection. Each one of its 64 entries specifies a page translation. It is fully associative, and can simultaneously hold translations for any combination of page sizes. To prevent TLB contention between data and instruction accesses, a 4-entry instruction and an 8-entry data shadow-TLB are maintained by the processor transparently to software.

Software manages the initialization and replacement of TLB entries. The PPC405 includes instructions for managing TLB entries by software running in privileged mode. This capability gives significant control to system software over the implementation of a page replacement strategy. For example, software can reduce the potential for TLB thrashing or delays associated with TLB-entry replacement by reserving a subset of TLB entries for globally accessible pages or critical pages.

Storage attributes are provided to control access of memory regions. When memory translation is enabled, storage attributes are maintained on a page basis and read from the

TLB when a memory access occurs. When memory translation is disabled, storage attributes are maintained in storage-attribute control registers. A zone-protection register (ZPR) is provided to allow system software to override the TLB access controls without requiring the manipulation of individual TLB entries. For example, the ZPR can provide a simple method for denying read access to certain application programs.

Chapter 6, Virtual-Memory Management, describes these memory-management resources in detail.

Instruction and Data Caches

The PPC405 accesses memory through the instruction-cache unit (ICU) and data-cache unit (DCU). Each cache unit includes a PLB-master interface, cache arrays, and a cache controller. Hits into the instruction cache and data cache appear to the CPU as single-cycle memory accesses. Cache misses are handled as requests over the PLB bus to another PLB device, such as an external-memory controller.

The PPC405 implements separate instruction-cache and data-cache arrays. Each is 16 KB in size, is two-way set-associative, and operates using 8-word (32 byte) cachelines. The caches are non-blocking, allowing the PPC405 to overlap instruction execution with reads over the PLB (when cache misses occur).

The cache controllers replace cachelines according to a least-recently used (LRU) replacement policy. When a cacheline fill occurs, the most-recently accessed line in the cache set is retained and the other line is replaced. The cache controller updates the LRU during a cacheline fill.

The ICU supplies up to two instructions every cycle to the fetch and decode unit. The ICU can also forward instructions to the fetch and decode unit during a cacheline fill, minimizing execution stalls caused by instruction-cache misses. When the ICU is accessed, four instructions are read from the appropriate cacheline and placed temporarily in a line buffer. Subsequent ICU accesses check this line buffer for the requested instruction prior to accessing the cache array. This allows the ICU cache array to be accessed as little as once every four instructions, significantly reducing ICU power consumption.

The DCU can independently process load/store operations and cache-control instructions. The DCU can also dynamically reprioritize PLB requests to reduce the length of an execution stall. For example, if the DCU is busy with a low-priority request and a subsequent storage operation requested by the CPU is stalled, the DCU automatically increases the priority of the current (low-priority) request. The current request is thus finished sooner, allowing the DCU to process the stalled request sooner. The DCU can forward data to the execute unit during a cacheline fill, further minimizing execution stalls caused by data-cache misses.

Additional features allow programmers to tailor data-cache performance to a specific application. The DCU can function in write-back or write-through mode, as determined by the storage-control attributes. Loads and stores that do not allocate cachelines can also be specified. Inhibiting certain cacheline fills can reduce potential pipeline stalls and unwanted external-bus traffic.

See **Chapter 5, Memory-System Management**, for details on the operation and control of the PPC405 caches.

Timer Resources

The PPC405 contains a 64-bit time base and three timers. The time base is incremented synchronously using the CPU clock or an external clock source. The three timers are incremented synchronously with the time base. (See **Chapter 8, Timer Resources**, for more information on these features.) The three timers supported by the PPC405 are:

- Programmable Interval Timer
- Fixed Interval Timer
- Watchdog Timer

Programmable Interval Timer

The *programmable interval timer* (PIT) is a 32-bit register that is decremented at the time-base increment frequency. The PIT register is loaded with a delay value. When the PIT count reaches 0, a PIT interrupt occurs. Optionally, the PIT can be programmed to automatically reload the last delay value and begin decrementing again.

Fixed Interval Timer

The *fixed interval timer* (FIT) causes an interrupt when a selected bit in the time-base register changes from 0 to 1. Programmers can select one of four predefined bits in the time-base for triggering a FIT interrupt.

Watchdog Timer

The *watchdog timer* causes a hardware reset when a selected bit in the time-base register changes from 0 to 1. Programmers can select one of four predefined bits in the time-base for triggering a reset, and the type of reset can be defined by the programmer.

Note: The time-base register alone does not cause interrupts to occur.

Debug

The PPC405 debug resources include special debug modes that support the various types of debugging used during hardware and software development. These are:

- *Internal-debug mode* for use by ROM monitors and software debuggers
- External-debug mode for use by JTAG debuggers
- *Debug-wait mode*, which allows the servicing of interrupts while the processor appears to be stopped
- *Real-time trace mode*, which supports event triggering for real-time tracing

Debug events are **supported** that allow developers to manage the debug process. Debug modes and debug events are controlled using debug registers in the processor. The debug registers are accessed either through software running on the processor or through the JTAG port. The JTAG port can also be used for board tests.

The debug modes, events, controls, and interfaces provide a powerful combination of debug resources for hardware and software development tools. **Chapter 9, Debugging**, describes these resources in detail.

PPC405 Interfaces

The PPC405 provides a set of interfaces that supports the attachment of cores and user logic. The software resources used to manage the PPC405 interfaces are described in the **Core-Configuration Register, page 495**. For information on the hardware operation, use, and electrical characteristics of these interfaces, refer to the **PPC405 Processor Block Manual**. The following interfaces are provided:

- Processor local bus interface
- Device control register interface
- Clock and power management interface
- JTAG port interface
- On-chip interrupt controller interface
- On-chip memory controller interface

Processor Local Bus

The *processor local bus* (PLB) interface provides a 32-bit address and three 64-bit data buses attached to the instruction-cache and data-cache units. Two of the 64-bit buses are attached to the data-cache unit, one supporting read operations and the other supporting write operations. The third 64-bit bus is attached to the instruction-cache unit to support instruction fetching.

Device Control Register

The *device control register (DCR) bus interface* supports the attachment of on-chip registers for device control. Software can access these registers using the **mfdcr** and **mtdcr** instructions.

Clock and Power Management

The *clock and power-management interface* supports several methods of clock distribution and power management.

JTAG Port

The *JTAG port interface* supports the attachment of external debug tools. Using the JTAG test-access port, a debug tool can single-step the processor and examine internal-processor state to facilitate software debugging. This capability complies with the IEEE 1149.1 specification for vendor-specific extensions, and is therefore compatible with standard JTAG hardware for boundary-scan system testing.

On-Chip Interrupt Controller

The *on-chip interrupt controller interface* is an external interrupt controller that combines asynchronous interrupt inputs from on-chip and off-chip sources and presents them to the core using a pair of interrupt signals (critical and noncritical). Asynchronous interrupt sources can include external signals, the JTAG and debug units, and any other on-chip peripherals.

On-Chip Memory Controller

An *on-chip memory (OCM) interface* supports the attachment of additional memory to the instruction and data caches that can be accessed at performance levels matching the cache arrays.

Operational Concepts

This chapter describes the operational concepts governing the PPC405 programming model. These concepts include the execution and memory-access models, processor operating modes, memory organization and management, and instruction conventions.

Execution Model

From a software viewpoint, PowerPC® processors implement a *sequential-execution model*. That is, the processors appear to execute instructions in program order. Internally and invisible to software, PowerPC processors can execute instructions out-of-order and can speculatively execute instructions. The processor is responsible for maintaining an in-order execution state visible to software. The execution of an instruction sequence can be interrupted by an exception caused by one of the executing instructions or by an asynchronous event. The PPC405 *does not support* out-of-order instruction execution. However, the processor does support speculative instruction execution, typically by predicting the outcome of branch instructions.

As described in **Ordering Memory Accesses**, [page 484](#), the PowerPC architecture specifies a weakly consistent memory model for shared-memory multiprocessor systems. The weakly consistent memory model allows system bus operations to be reordered dynamically. The goal of reordering bus operations is to reduce the effect of memory latency and improving overall performance. In single-processor systems, loads and stores can be reordered dynamically to allow efficient utilization of the processor bus. Loads can be performed speculatively to enhance the speculative-execution capabilities. This model provides an opportunity for significantly improved performance over a model that has stronger memory-consistency rules, but places the responsibility for access ordering on the programmer.

When a program requires strict instruction-execution ordering or memory-access ordering for proper execution, the programmer must insert the appropriate ordering or synchronization instructions into the program. These instructions are described in **Synchronizing Instructions**, [page 460](#). The concept of synchronization is described in the **Synchronization Operations** section that follows.

The PPC405 supports many aspects of the weakly consistent model but not all of them. Specifically, the PPC405 *does not provide* hardware support for multiprocessor memory coherency and *does not support* speculative loads. If the order of memory accesses is important to the correct operation of a program, care must be taken in porting such a program from the PPC405 to a processor that supports multiprocessor memory coherency and speculative loads.

Synchronization Operations

Various forms of synchronizing operations can be used by programs executing on the PPC405 processor to control the behavior of instruction execution and memory accesses. Synchronizing operations fall into the following three categories:

- Context synchronization
- Execution synchronization
- Storage synchronization

Each synchronization category is described in the following sections. Instructions provided by the PowerPC architecture for synchronization purposes are described on [page 460](#).

Context Synchronization

The state of the execution environment (privilege level, translation mode, and memory protection) defines a program's context. An instruction or event is *context synchronizing* if the operation satisfies all of the following conditions:

- Instruction dispatch is halted when the operation is recognized by the processor. This means the instruction-fetch mechanism stops issuing (sending) instructions to the execution units.
- The operation is not initiated (for instructions, this means dispatched) until all prior instructions complete execution to a point where they report any exceptions they cause to occur. In the case of an instruction-synchronize (**isync**) instruction, the **isync** does not complete execution until all prior instructions complete execution to a point where they report any exceptions they cause to occur.
- All instructions that precede the operation complete execution in the context they were initiated. This includes privilege level, translation mode, and memory protection.
- All instructions following the operation complete execution in the new context established by the operation.
- If the operation is an exception, or directly causes an exception to occur (for example, the **sc** instruction causes a system-call exception), the operation is not initiated until all higher-priority exceptions are recognized by the exception mechanism.

The system-call instruction (**sc**), return-from-interrupt instructions (**rfi** and **rfci**), and most exceptions are examples of context-synchronizing operations.

Context-synchronizing operations do not guarantee that subsequent memory accesses are performed using the memory context established by previous instructions. When memory-access ordering must be enforced, storage-synchronizing instructions are required.

Execution Synchronization

An instruction is *execution synchronizing* if it satisfies the conditions of the first two items (as described above) for context synchronization:

- Instruction dispatch is halted when the operation is recognized by the processor. This means the instruction-fetch mechanism stops issuing (sending) instructions to the execution units.
- The operation is not initiated until all instructions in execution complete to a point where they report any exceptions they cause to occur. In the case of a synchronize (**sync**) instruction, the **sync** does not complete execution until all prior instructions complete execution to a point where they report any exceptions they cause to occur.

The **sync** and *move-to machine-state register* (**mtmsr**) instructions are examples of execution-synchronizing instructions.

All context-synchronizing instructions are execution synchronizing. However, unlike a context-synchronizing operation, there is no guarantee that subsequent instructions execute in the context established by an execution-synchronizing instruction. The new context becomes effective sometime after the execution-synchronizing instruction completes and before or during a subsequent context-synchronizing operation.

Storage Synchronization

The PowerPC architecture specifies a weakly consistent memory model for shared-memory multiprocessor systems. With this model, the order that the processor performs memory accesses, the order that those accesses complete in memory, and the order that those accesses are viewed as occurring by another processor can all differ. The PowerPC architecture supports storage-synchronizing operations that provide a capability for enforcing memory-access ordering, allowing programs to share memory. Support is also provided to allow programs executing on a processor to share memory with some other mechanism that can access memory, such as an I/O device.

Device control registers (DCRs) are treated as memory-mapped registers from a synchronization standpoint. Storage-synchronization operations must be used to enforce synchronization of DCR reads and writes.

Processor Operating Modes

The PowerPC architecture defines two levels of privilege, each with an associated processor operating mode:

- Privileged mode
- User mode

The processor operating mode is controlled by the privilege-level field in the machine-state register (MSR[PR]). When MSR[PR] = 0, the processor operates in privileged mode. When MSR[PR] = 1, the processor operates in user mode. MSR[PR] = 0 following reset, placing the processor in privileged mode. See **Machine-State Register**, [page 467](#) for more information on this register.

Attempting to execute a privileged instruction when in user mode causes a privileged-instruction program exception (see **Program Interrupt (0x0700)**, [page 547](#)).

Throughout this book, the terms *privileged* and *system* are used interchangeably to refer to software that operates under the privileged-programming model. Likewise, the terms *user* and *application* are used to refer to software that operates under the user-programming model. Registers and instructions are defined as either privileged or user, indicating which of the two programming models they belong to. User registers and user instructions belong to both the user-programming and privileged-programming models.

Privileged Mode

Privileged mode allows programs to access all registers and execute all instructions supported by the processor. The *privileged-programming model* comprises the entire register set and instruction set supported by the PPC405. Operating systems are typically the only software that runs in privileged mode.

The registers available only in privileged mode are shown in [Figure 4-1, page 466](#). Refer to the corresponding section describing each register for more information. The instructions available only in privileged mode are shown in [Table 4-3, page 470](#). The operation of each instruction is described in [Chapter 11, Instruction Set](#).

Privileged mode is sometimes referred to as *supervisor state*.

User Mode

User mode restricts access to some registers and instructions. The *user-programming model* comprises the register set and instruction set supported by the processor running in user mode, and is a subset of the privileged-programming model. Operating systems typically confine the execution of application programs to user mode, thereby protecting system resources and other software from the effects of errant applications.

The registers available in user mode are shown in [Figure 3-1, page 396](#). Refer to the corresponding section in [Chapter 3](#) for a description of each register. All instructions are available in user mode except as shown in [Table 4-3, page 470](#).

User mode is sometimes referred to as *problem state*.

Memory Organization

PowerPC programs reference memory using an effective address computed by the processor when executing a load, store, branch, or cache-control instruction, and when fetching the next-sequential instruction. Depending on the address-relocation mode, this effective address is either used to directly access physical memory or is treated as a virtual address that is translated into physical memory.

Effective-Address Calculation

Programs reference memory using an *effective address* (also called a *logical address*). An effective address (EA) is the 32-bit unsigned sum computed by the processor when accessing memory, executing a branch instruction, or fetching the next-sequential instruction. An EA is often referred to as the *next-instruction address* (NIA) when it is used to fetch an instruction (sequentially or as the result of a branch). The input values and method used by the processor to calculate an EA depend on the instruction that is executed.

When accessing data in memory, effective addresses are calculated in one of the following ways:

- $EA = (rA \mid 0)$ —this is referred to as *register-indirect* addressing.
- $EA = (rA \mid 0) + \text{offset}$ —this is referred to as *register-indirect with immediate-index* addressing.
- $EA = (rA \mid 0) + (rB)$ —this is referred to as *register-indirect with index* addressing.

Note: In the above, the notation $(rA \mid 0)$ specifies the following:

If the rA instruction field is 0, the base address is 0.

If the rA instruction field is not 0, the contents of register rA are used as the base address.

When instructions execute sequentially, the next-instruction effective address is the current-instruction address (CIA) + 4. This is because all instructions are four bytes long. When branching to a new address, the next-instruction effective address is calculated in one of the following ways:

- $NIA = CIA + \text{displacement}$ —this is referred to as *branch-to-relative* addressing.
- $NIA = \text{displacement}$ —this is referred to as *branch-to-absolute* addressing.
- $NIA = (LR)$ —this is referred to as *branch to link-register* addressing.
- $NIA = (CTR)$ —this is referred to as *branch to count-register* addressing.

When the NIA is calculated for a branch instruction, the two low-order bits (30:31) are always cleared to 0, forcing word-alignment of the address. This is true even when the address is contained in the LR or CR, and the register contents are not word-aligned.

All effective-address computations are performed by the processor using unsigned binary arithmetic. Carries from bit 0 are ignored and the effective address wraps from the maximum address ($2^{32}-1$) to address 0 when the calculation overflows.

Physical Memory

Physical memory represents the address space of memory installed in a computer system, including memory-mapped I/O devices. Generally, the amount of physical memory actually available in a system is smaller than that supported by the processor. When address translation is supported by the operating system—as it is in virtual-memory systems—the very-large virtual-address space is translated into the smaller physical-address space using the memory-management resources supported by the processor.

The PPC405 supports up to four gigabytes of physical memory using a 32-bit physical address. A hierarchical-memory system involving external (system) memory and the caches internal to the processor are employed to support that address space. The PPC405 supports separate level-1 (L1) caches for instructions and data. The operation and control of these caches is described in **Chapter 5, Memory-System Management**.

Virtual Memory

Virtual memory is a relocatable address space that is generally larger than the physical-memory space installed in a computer system. Operating systems relocate (map) applications and data in virtual memory so it appears that more memory is available than actually exists. Virtual memory software moves unused instructions and data between physical memory and external storage devices (such as a hard drive) when insufficient physical memory is available. The PPC405 supports a 40-bit virtual address that allows privileged software to manage a one-terabyte virtual-memory space.

Memory Management

Memory management describes the collection of mechanisms used to translate the addresses generated by programs into physical-memory addresses. Memory management also consists of the mechanisms used to characterize memory-region behavior, also referred to as *storage control*. Memory management is performed by privileged-mode software and is completely transparent to user-mode programs running in virtual mode.

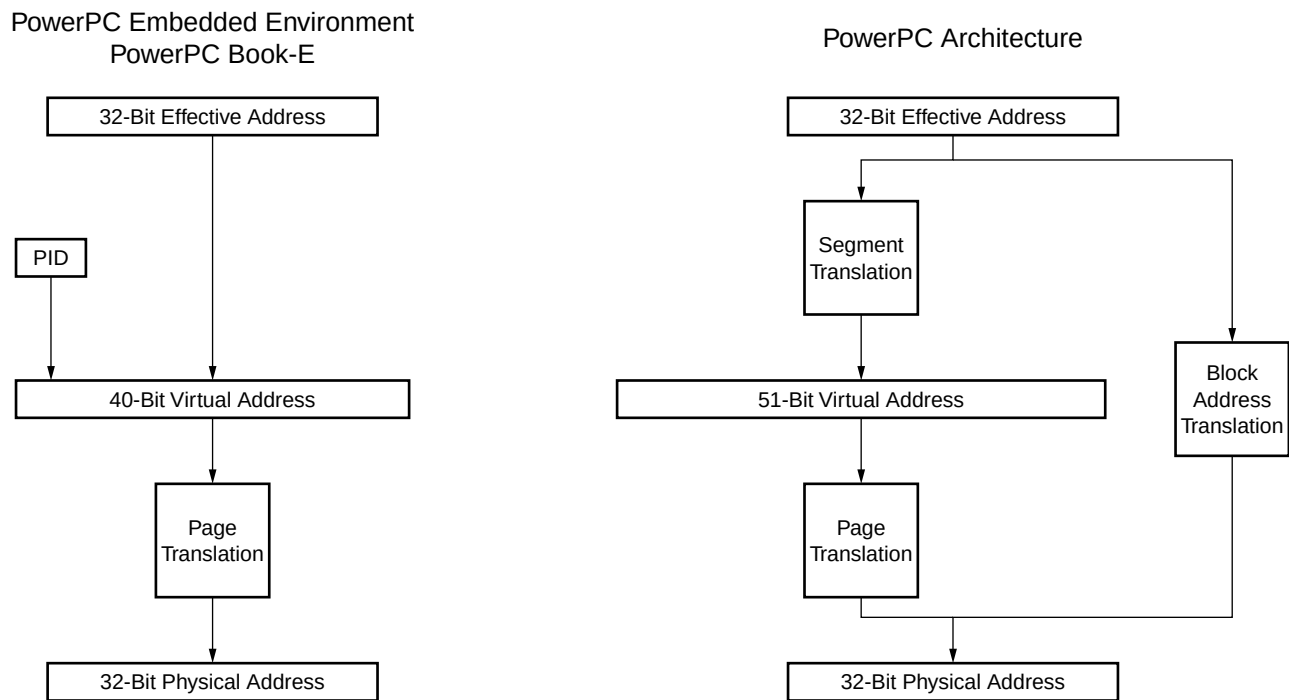
The PPC405 is a PowerPC embedded-environment implementation. The memory-management resources defined by the PowerPC embedded-environment architecture (and its successor, the PowerPC Book-E architecture) differ significantly from the resources defined by the PowerPC architecture. The resources defined by the PowerPC embedded environment architecture are well-suited for the special requirements of embedded-system applications. The resources defined by the PowerPC architecture better meet the requirements of desktop and commercial-workstation systems.

Generally, the differences between the two memory-management mechanisms are as follows:

- The PPC405 supports *software page translation* and provides special instructions for managing the page tables and the translation look-aside buffer (TLB) internal to the processor. The page-translation table format, organization, and search algorithms are software-dependent and transparent to the PPC405 processor. The PowerPC architecture, on the other hand, defines the page-translation table organization, format, and search algorithms. It does not define support for the special page table and TLB instructions but instead assumes the processor hardware is responsible for searching page tables and updating the TLB.
- The PPC405 supports *variable-sized pages*. The PowerPC architecture defines fixed-size pages of 4 KB.
- The PPC405 *does not* support the segment-translation mechanism defined by the PowerPC architecture.
- The PPC405 *does not* support the block-address-translation (BAT) mechanism defined by the PowerPC architecture.

- Additional storage-control attributes not defined by the PowerPC architecture are supported by the PPC405. The methods for using these attributes to characterize memory regions also differ.

At a high level, [Figure 2-1](#) shows the differences between 32-bit memory management in the PowerPC embedded-environment architecture (and PowerPC Book-E architecture) and in the PowerPC architecture. See [Chapter 6, Virtual-Memory Management](#) for more information on the resources supported by the PPC405. Additional information on the differences with the PowerPC architecture is described in [Appendix E, PowerPC® 6xx/7xx Compatibility](#). PowerPC Book-E architecture extends the resources first defined by the PowerPC embedded-environment architecture. A description of those extensions is in [Appendix F, PowerPC® Book-E Compatibility](#).



UG011_13_033101

Figure 2-1: PowerPC 32-Bit Memory Management

Addressing Modes

Programs can use 32-bit effective addresses to reference the 4 GB physical-address space using one of two addressing modes:

- Real mode
- Virtual mode

Real mode and virtual mode are enabled and disabled independently for instruction fetches and data accesses. The instruction-fetch address mode is controlled using the instruction-relocate (IR) field in the machine-state register (MSR). When MSR[IR] = 0, instruction fetches are performed in real mode. When MSR[IR] = 1, instruction fetches are performed in virtual mode. Similarly, the data-access address mode is controlled using the data-relocate (DR) field in the MSR. When MSR[DR] = 0, data accesses are performed in real mode. Setting MSR[DR] = 1 enables virtual mode for data accesses. See [Virtual Mode, page 508](#) for more information on these fields.

Real Mode

In *real mode*, an effective address is used directly as the physical address into the 4 GB address space. Here, the logical-address space is mapped directly onto the physical-address space.

Virtual Mode

In *virtual mode*, address translation is enabled. Effective addresses are translated into physical addresses using the memory-management unit, as shown in [Figure 2-1, page 382](#). In this mode, pages within the logical-address space are mapped onto pages in the physical-address space. An overview of memory management is provided in the following section.

Operand Conventions

Bit positions within registers and memory operands (bytes, halfwords, and words) are numbered consecutively from left to right, starting with zero. The most-significant bit is always numbered 0. The number assigned to the least-significant bit depends on the size of the register or memory operand, as follows:

- Byte—the least-significant bit is numbered 7.
- Halfword—the least-significant bit is numbered 15.
- Word—the least-significant bit is numbered 31.

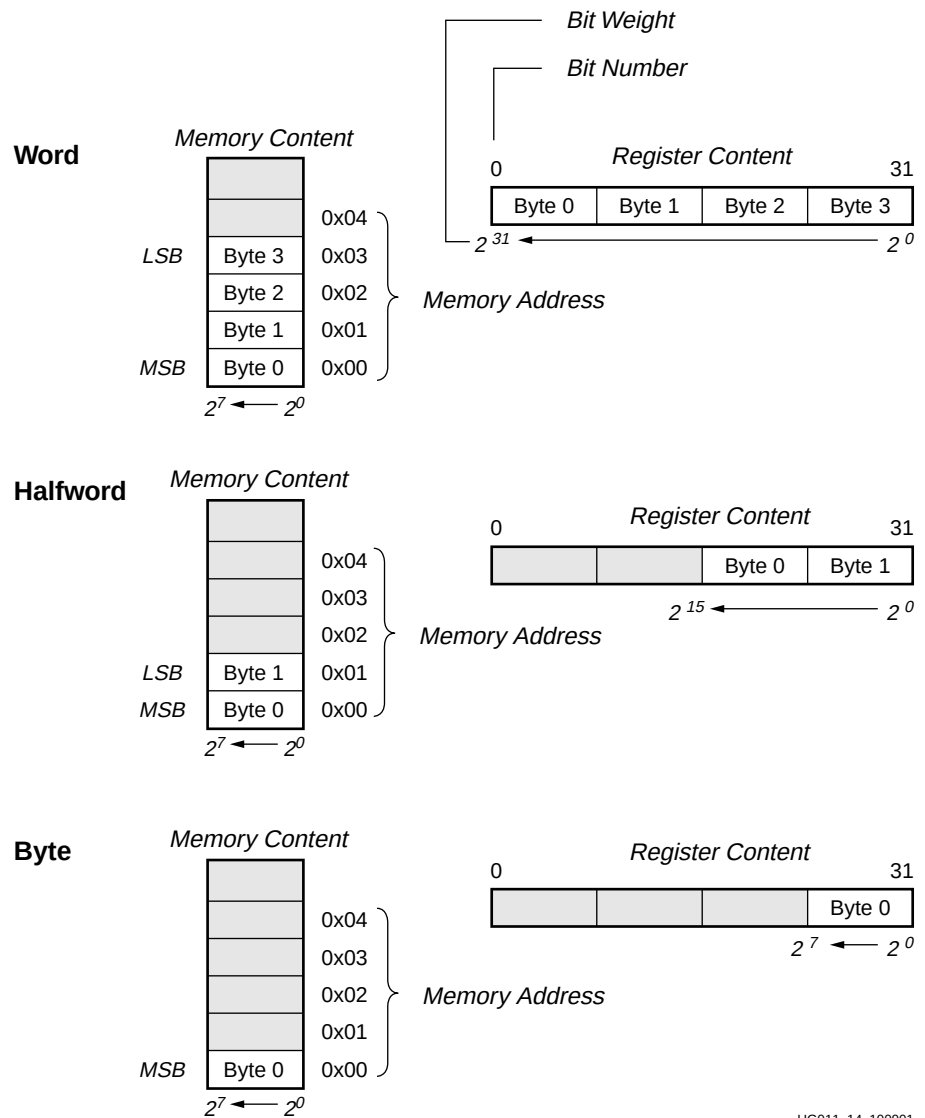
A bit set to 1 has a numerical value associated with its position (b) relative to the least-significant bit (lsb). This value is equal to $2^{(lsb-b)}$. For example, if bit 5 is set to 1 in a byte, halfword, or word memory operand, its value is determined as follows:

- Byte—the value is $2^{(7-5)}$, or 4 .
- Halfword—the value is $2^{(15-5)}$, or 1024 .
- Word—the value is $2^{(31-5)}$, or 67108864 .

Bytes in memory are addressed consecutively starting with zero. The PPC405 supports both big-endian and little-endian byte ordering, with big-endian being the default byte ordering. Bit ordering within bytes and registers is always big endian.

The operand length is implicit for each instruction. Memory operands can be bytes (eight bits), halfwords (two bytes), words (four bytes), or strings (one to 128 bytes). For the load/store multiple instructions, memory operands are a sequence of words. The address of any memory operand is the address of its first byte (that is, of its lowest-numbered byte). [Figure 2-2](#) shows how word, halfword, and byte operands appear in memory (using big-endian ordering) and in a register. The memory operand appears on the left in this diagram and the equivalent register representation appears on the right.

The following sections describe the concepts of byte ordering and data alignment, and their significance to the PowerPC PPC405.



UG011_14_100901

Figure 2-2: Operand Data Types

Byte Ordering

The order that addresses are assigned to individual bytes within a scalar (a single data object or instruction) is referred to as *endianness*. Halfwords, words, and doublewords all consist of more than one byte, so it is important to understand the relationship between the bytes in a scalar and the addresses of those bytes. For example, when the processor loads a register with a value from memory, it needs to know which byte in memory holds the high-order byte, which byte holds the next-highest-order byte, and so on.

Computer systems generally use one of the following two byte orders to address data:

- *Big-endian* ordering assigns the lowest-byte address to the highest-order (“left-most”) byte in the scalar. The next sequential-byte address is assigned to the next-highest byte, and so on. The term “big endian” is used because the “big end” of the scalar (when considered as a binary number) comes first in memory.
- *Little-endian* ordering assigns the lowest-byte address to the lowest-order (“right-most”) byte in the scalar. The next sequential-byte address is assigned to the next-lowest byte, and so on. The term “little endian” is used because the “little end” of the scalar (when considered as a binary number) comes first in memory.

The following sections further describe the differences between big-endian and little-endian byte ordering. The default byte ordering assumed by the PPC405 is big-endian. However, the PPC405 also fully supports little-endian peripherals and memory.

Structure-Mapping Examples

The following C language structure, *s*, contains an assortment of scalars and a character string. The comments show the values assumed in each structure element. These values show how the bytes comprising each structure element are mapped into memory.

```
struct {
    int a;           /* 0x1112_1314 word */
    long long b;     /* 0x2122_2324_2526_2728 doubleword */
    char *c;         /* 0x3132_3334 word */
    char d[7];       /* 'A', 'B', 'C', 'D', 'E', 'F', 'G' array of bytes */
    short e;         /* 0x5152 halfword */
    int f;           /* 0x6162_6364 word */
} s;
```

C structure-mapping rules permit the use of padding (skipped bytes) to align scalars on desirable boundaries. The structure-mapping examples show how each scalar aligns on its natural boundary (the alignment boundary is equal to the scalar size). This alignment introduces padding of four bytes between *a* and *b*, one byte between *d* and *e*, and two bytes between *e* and *f*. The same amount of padding is present in both big-endian and little-endian mappings.

Big-Endian Mapping

The big-endian mapping of structure *s* follows. The contents of each byte, as defined in structure *s*, is shown as a (hexadecimal) number or character (for the string elements). Data addresses (in hexadecimal) are shown below the corresponding data value.

11 0x00	12 0x01	13 0x02	14 0x03	0x04	0x05	0x06	0x07
21 0x08	22 0x09	23 0x0A	24 0x0B	25 0x0C	26 0x0D	27 0x0E	28 0x0F
31 0x10	32 0x11	33 0x12	34 0x13	'A' 0x14	'B' 0x15	'C' 0x16	'D' 0x17
'E' 0x18	'F' 0x19	'G' 0x1A	0x1B	51 0x1C	52 0x1D	0x1E	0x1F
61 0x20	62 0x21	63 0x22	64 0x23	0x24	0x25	0x26	0x27

Little-Endian Mapping

The little-endian mapping of structure *s* follows.

14 0x00	13 0x01	12 0x02	11 0x03	0x04	0x05	0x06	0x07
28 0x08	27 0x09	26 0x0A	25 0x0B	24 0x0C	23 0x0D	22 0x0E	21 0x0F
34 0x10	33 0x11	32 0x12	31 0x13	'A' 0x14	'B' 0x15	'C' 0x16	'D' 0x17
'E' 0x18	'F' 0x19	'G' 0x1A	0x1B	52 0x1C	51 0x1D	0x1E	0x1F
64 0x20	63 0x21	62 0x22	61 0x23	0x24	0x25	0x26	0x27

Little-Endian Byte Ordering Support

Except as noted, this book describes the processor from the perspective of big-endian operations. However, the PPC405 processor also fully supports little-endian operations. This support is provided by the endian (E) storage attribute described in the following sections. The endian-storage attribute is defined by both the PowerPC embedded-environment architecture and PowerPC Book-E architecture.

Little-endian *mode*, defined by the PowerPC architecture, is *not implemented* by the PPC405. Little-endian mode does not support *true* little-endian memory accesses. This is because little-endian mode modifies memory addresses rather than reordering bytes as they are accessed. Memory-address modification restricts how the processor can access misaligned data and I/O. The PPC405 little-endian support does not have these restrictions.

Endian (E) Storage Attribute

The endian (E) storage attribute allows the PPC405 to support direct connection of little-endian peripherals and memory containing little-endian instructions and data. An E storage attribute is associated with every memory reference—instruction fetch, data load, and data store. The E attribute specifies whether the memory region being accessed should be interpreted as big endian (E = 0) or little endian (E = 1).

If virtual mode is enabled (MSR[IR] = 1 or MSR[DR] = 1), the E field in the corresponding TLB entry defines the endianness of a memory region. When virtual mode is disabled (MSR[IR] = 0 and MSR[DR] = 0), the SLER defines the endianness of a memory region. See [Chapter 6, Virtual-Memory Management](#) for more information on virtual memory, and [Storage Little-Endian Register \(SLER\), page 491](#) for more information on the SLER.

When a memory region is defined as little endian, the processor accesses those bytes as if they are arranged in true little-endian order. Unlike the little-endian mode defined by the PowerPC architecture, no address modification is performed when accessing memory regions designated as little endian. Instead, the PPC405 reorders the bytes as they are transferred between the processor and memory.

On-the-fly reversal of bytes in little-endian memory regions is handled in one of two ways, depending on whether the memory access is an instruction fetch or a data access (load or store). The following sections describe byte reordering for both types of memory accesses.

Little-Endian Instruction Fetching

Instructions are word (four-byte) data types that are always aligned on word boundaries in memory. Instructions stored in a big-endian memory region are arranged with the most-significant byte (MSB) of the instruction word at the lowest byte address.

Consider the big-endian mapping of instruction *p* at address 0x00, where, for example, *p* is an **add r7,r7,r4** instruction (instruction opcode bytes are shown in hexadecimal on top, with the corresponding byte address shown below):

MSB			LSB
7C	E7	22	14
0x00	0x01	0x02	0x03

In the little-endian mapping, instruction *p* is arranged with the least-significant byte (LSB) of the instruction word at the lowest byte address:

LSB			MSB
14	22	E7	7C
0x00	0x01	0x02	0x03

The instruction decoder on the PPC405 assumes the instructions it receives are in big-endian order. When an instruction is fetched from memory, the instruction must be placed in the instruction queue in big-endian order so that the instruction is properly decoded. When instructions are fetched from little-endian memory regions, the four bytes of an instruction word are reversed by the processor before the instruction is decoded. This byte reversal occurs between memory and the instruction-cache unit (ICU) and is transparent to software. The ICU always stores instructions in big-endian order regardless of whether the instruction-memory region is defined as big endian or little endian. This means the bytes are already in the proper order when an instruction is transferred from the ICU to the instruction decoder.

If the endian-storage attribute is changed, the affected memory region must be reloaded with program and data structures using the new endian ordering. If the endian ordering of

instruction memory changes, the ICU must be made coherent with the updates. This is accomplished by invalidating the ICU and updating the instruction memory with instructions using the new endian ordering. Subsequent fetches from the updated memory region are interpreted correctly before they are cached and decoded. See **Instruction-Cache Control Instructions**, page 492 for information on instruction-cache invalidation.

Little-Endian Data Accesses

Unlike instruction fetches, data accesses from little-endian memory regions are *not* byte-reversed between memory and the data-cache unit (DCU). The data-byte ordering stored in memory depends on the data size (byte, halfword, or word). The data size is not known until the data item is moved between memory and a general-purpose register. In the PPC405, byte reversal of load and store accesses is performed between the DCU and the GPRs.

When accessing data in a little-endian memory region, the processor automatically does the following regardless of data alignment:

- For byte loads/stores, no reordering occurs
- For halfword loads/stores, bytes are reversed within the halfword
- For word loads/stores, bytes are reversed within the word

The big-endian and little-endian mappings of the structure *s*, shown in **Structure-Mapping Examples**, page 385, demonstrate how the size of a data item determines its byte ordering. For example:

- The word *a* has its four bytes reversed within the word spanning addresses 0x00–0x03
- The halfword *e* has its two bytes reversed within the halfword spanning addresses 0x1C–0x1D
- The array of bytes *d* (where each data item is a byte) is not reversed when the big-endian and little-endian mappings are compared (For example, the character 'A' is located at address 14 in both the big-endian and little-endian mappings)

In little-endian memory regions, data alignment is treated as it is in big-endian memory regions. Unlike little-endian mode in the PowerPC architecture, no special alignment exceptions occur when accessing data in little-endian memory regions versus big-endian regions.

Load and Store Byte-Reverse Instructions

When accessing big-endian memory regions, load/store instructions move the more-significant register bytes to and from the lower-numbered memory addresses and the less-significant register bytes are moved to and from the higher-numbered memory addresses. The *load/store with byte-reverse* instructions, as described in **Load and Store with Byte-Reverse Instructions**, page 421, do the opposite. The more-significant register bytes are moved to and from the higher-numbered memory addresses, and the less-significant register bytes are moved to and from the lower-numbered memory addresses.

Even though the load/store with byte-reverse instructions can be used to access little-endian memory, the E storage attribute provides two advantages over using those instructions:

- The load/store with byte-reverse instructions do not solve the problem of fetching instructions from a little-endian memory region. Only the E storage attribute mechanism supports little-endian instruction fetching.
- Typical compilers cannot make general use of the load/store with byte-reverse instructions, so these instructions are normally used only in device drivers written in hand-coded assembler. However, compilers can take full advantage of the E storage-attribute mechanism, allowing application programmers working in a high-level language, such as C, to compile programs and data structures using little-endian ordering.

Operand Alignment

The operand of a memory-access instruction has a natural alignment boundary equal to the operand length. In other words, the *natural* address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned on its natural boundary, otherwise it is misaligned.

All instructions are words and are always aligned on word boundaries.

Table 2-1 shows the value required by the least-significant four address bits (bits 28:31) of each data type for it to be aligned in memory. A value of *x* in a given bit position indicates the address bit can have a value of 0 or 1.

Table 2-1: Memory Operand Alignment Requirements

Data Type	Size	Aligned Address Bits 28:31
Byte	8 Bits	<i>xxxx</i>
Halfword	2 Bytes	<i>xxx0</i>
Word	4 Bytes	<i>xx00</i>
Doubleword	8 Bytes	<i>x000</i>

The concept of alignment can be generally applied to any data in memory. For example, a 12-byte data item is said to be word aligned if its address is a multiple of four.

Some instructions require aligned memory operands. Also, alignment can affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned.

Alignment and Endian Storage Control

The endian storage-control attribute (E) *does not* affect how the processor handles operand alignment. Data alignment is handled identically for accesses to big-endian and little-endian memory regions. No special alignment exceptions occur when accessing data in little-endian memory regions. However, alignment exceptions that apply to big-endian memory accesses also apply to little-endian memory accesses.

Performance Effects of Operand Alignment

The performance of accesses varies depending on the following parameters:

- Operand size
- Operand alignment
- Boundary crossing:
 - None
 - Cache block
 - Page

To obtain the best performance across the widest range of PowerPC embedded-environment implementations and PowerPC Book-E processor implementations, programmers should assume the alignment performance effects described in **Figure 2-2**. This table applies to both big-endian and little-endian accesses. **Figure 2-2** also applies to PowerPC processors running in the default big-endian mode. However, those same processors suffer further performance degradation when running in PowerPC little-endian mode.

Table 2-2: Performance Effects of Operand Alignment

Operand		Boundary Crossing		
Size	Byte Alignment	None	Cache Block	Page
Byte	1	Optimal	Not Applicable	
Halfword	2	Optimal	Not Applicable	
	1	Good	Good	Poor
Word	4	Optimal	Not Applicable	
	<4	Good	Good	Poor
Multiple Word	4	Good	Good	Good ¹
Byte String	1	Good	Good	Poor
Note: Assumes both pages have identical storage-control attributes. Performance is poor otherwise.				

Alignment Exceptions

Misalignment occurs when addresses are not evenly divided by the data-object size. The PPC405 automatically handles misalignments within word boundaries and across word boundaries, generally at a cost in performance. Some instructions cause an alignment exception if their operand is not properly aligned, as shown in [Table 2-3](#).

Table 2-3: Instructions Causing Alignment Exceptions

Mnemonic	Condition
dcbz	EA is in non-cacheable or write-through memory.
dcread, lwarx, stwcx	EA is not word aligned.

Cache-control instructions ignore the four least-significant bits of the EA. No alignment restrictions are placed on an EA when executing a cache-control instruction. However, certain storage-control attributes can cause an alignment exception to occur when a cache-control instruction is executed. If data-address translation is disabled (MSR[DR]=0) and a **dcbz** instruction references a non-cacheable memory region, or the memory region uses a write-through caching policy, an alignment exception occurs. The alignment exception allows the operating system to emulate the write-through caching policy. See **Alignment Interrupt (0x0600)**, [page 546](#) for more information.

Instruction Conventions

Instruction Forms

Opcode tables and instruction listings often contain information regarding the instruction *form*. This information refers to the type of format used to encode the instruction. Grouping instructions by format is useful for programmers that must deal directly with machine-level code, particularly programmers that write assemblers and disassemblers.

The formats used for the instructions of the PowerPC embedded-environment architecture are shown in **Instructions Grouped by Form**, [page 828](#). The **Instruction Set Information**, [page 833](#) also shows the form used by each instruction, listed alphabetically by mnemonic.

Instruction Classes

PowerPC instructions belong to one of the following three classes:

- Defined
- Illegal
- Reserved

An instruction class is determined by examining the primary opcode, and the extended opcode if one exists. If the opcode and extended opcode combination does not specify a defined instruction or reserved instruction, the instruction is illegal. Although the definitions of these terms are consistent among PowerPC processor implementations, the assignment of these classifications is not. For example, an instruction specific to 64-bit implementations is considered defined for 64-bit implementations but illegal for 32-bit implementations.

In future versions of the PowerPC architecture, instruction encodings that are now illegal or reserved can become defined (by being added to the architecture) or reserved (by being assigned a special purpose in an implementation).

Boundedly Undefined

The results of executing an instruction are said to be *boundedly undefined* if those results could be achieved by executing an arbitrary sequence of instructions, starting in the machine state prior to executing the given instruction. Boundedly-undefined results for an instruction can vary between implementations and between different executions on the same implementation.

Defined Instruction Class

Defined instructions contain all the instructions defined by the PowerPC architecture. Defined instructions are guaranteed to be supported by all implementations of the PowerPC architecture. The only exceptions are the instructions defined only for 64-bit implementations, instructions defined only for 32-bit implementations, and instructions defined only for embedded implementations. A PowerPC processor can invoke the illegal-instruction error handler (through the program-interrupt handler) when an unimplemented instruction is encountered, allowing emulation of the instruction in software.

A defined instruction can have preferred forms and invalid forms as described in the following sections.

Preferred Instruction Forms

A *preferred form* of a defined instruction is one in which the instruction executes in an efficient manner. Any form other than the preferred form can take significantly longer to execute. The following instructions have preferred forms:

- Load-multiple and store-multiple instructions
- Load-string and store-string instructions
- OR-immediate instruction (preferred form of no-operation)

Invalid Instruction Forms

An *invalid form* of a defined instruction is one in which one or more operands are coded incorrectly and in a manner that can be deduced only by examining the instruction encoding (primary and extended opcodes). For example, coding a value of 1 in a reserved bit (normally cleared to 0) produces an invalid instruction form.

The following instructions have invalid forms:

- Branch-conditional instructions
- Load with update and store with update instructions
- Load multiple instructions

- Load string instructions
- Integer compare instructions

On the PPC405, attempting to execute an invalid instruction form generally yields a boundedly-undefined result, although in some cases a program exception (illegal-instruction error) can occur.

Optional Instructions

The PowerPC architecture allows implementations to optionally support some defined instructions. The PPC405 does not implement the following instructions:

- Floating-point instructions
- External-control instructions (**eciwx**, **ecowx**)
- Invalidate TLB entry (**tlbie**)

Illegal Instruction Class

Illegal instructions are grouped into the following categories:

- Unused primary opcodes. The following primary opcodes are defined as illegal but can be defined by future extensions to the architecture:
1, 5, 6, 56, 57, 60, 61
- Unused extended opcodes. Unused extended opcodes can be derived from information in **Instructions Sorted by Opcode**, [page 817](#). The following primary opcodes have unused extended opcodes:
19, 31, 59, 63
- An instruction consisting entirely of zeros is guaranteed to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized memory causes an illegal-instruction error. If only the primary opcode consists of all zeros, the instruction is considered a reserved instruction, as described in the following section.

An attempt to execute an illegal instruction causes an illegal-instruction error (program exception). With the exception of an instruction consisting entirely of zeros, illegal instructions are available for future addition to the PowerPC architecture.

Reserved Instruction Class

Reserved instructions are allocated to specific implementation-dependent purposes not defined by the PowerPC architecture. An attempt to execute an unimplemented reserved instruction causes an illegal-instruction error (program exception). The following types of instructions are included in this class:

- Instructions for the POWER architecture that have not been included in the PowerPC architecture.
- Implementation-specific instructions used to conform to the PowerPC architecture specification. For example, *load data-TLB entry* (**tlbld**) and *load instruction-TLB entry* (**tlbli**) instructions in the PowerPC 603™.
- The instruction with primary opcode 0, when the instruction does not consist entirely of binary zeros.
- Any other implementation-specific instruction not defined by the PowerPC architecture.

PowerPC Embedded-Environment Instructions

To support functions required in embedded-system applications, the PowerPC embedded-environment architecture defines instructions that are not part of the PowerPC architecture. [Table 2-4](#) lists the instructions specific to the PPC405 and other PowerPC embedded-environment family implementations. From the standpoint of the PowerPC architecture, these instructions are part of the reserved class and are implementation

dependent. Programs using these instructions are not portable to implementations that do not support the PowerPC embedded-environment architecture.

In the table, the syntax “[o]” indicates the instruction has an overflow-enabled form that updates XER[OV,SO] as well as a non-overflow-enabled form. The syntax “[.]” indicates the instruction has a record form that updates CR[CR0] as well as a non-record form. The headings “defined” and “allocated”, as they are used in [Table 2-4](#), are described in the following section, **PowerPC Book-E Instruction Classes**.

Table 2-4: PowerPC Embedded-Environment Instructions

Defined (Book-E)		Allocated (Book-E)		
mfdr	tlbre	dccci	macchw[o][.]	nmacchw[o][.]
mtdr	tlbsx[.]	dcread	macchws[o][.]	nmacchws[o][.]
rfci	tlbwe	iccci	macchwsu[o][.]	nmachhw[o][.]
wrtee		icread	macchwu[o][.]	nmachhws[o][.]
wrteei			machhw[o][.]	nmaclhw[o][.]
			machhws[o][.]	nmaclhws[o][.]
			machhwsu[o][.]	mulchw[.]
			machhwu[o][.]	mulchwu[.]
			maclhw[o][.]	mulhhw[.]
			maclhws[o][.]	mulhhwu[.]
			maclhwsu[o][.]	mullhw[.]
			maclhwu[o][.]	mullhwu[.]

PowerPC Book-E Instruction Classes

The PowerPC Book-E architecture defines *four* instruction classes:

- Defined
- Allocated
- Reserved
- Preserved

Referring to [Table 2-4](#), the first two columns indicate which PPC405 instructions are part of the defined instruction class and are guaranteed support in PowerPC Book-E processor implementations. The last three columns indicate which PPC405 instructions are part of the allocated instruction class. Support of these instructions by PowerPC Book-E processors is implementation-dependent.

Defined Book-E Instruction Class

The *defined instruction class* consists of all instructions defined by the PowerPC Book E architecture. In general, defined instructions are guaranteed to be supported by a PowerPC Book E processor as specified by the architecture, either within the processor implementation itself or within emulation software supported by the operating system.

Allocated Book-E Instruction Class

The *allocated instruction class* contains the set of instructions used for implementation-dependent and application-specific use, outside the scope of the PowerPC Book E architecture.

Reserved Book-E Instruction Class

The *reserved instruction class* consists of all instruction primary opcodes (and associated extended opcodes, if applicable) that do not belong to either the defined class or the allocated class.

Preserved Book-E Instruction Class

The *preserved instruction class* is provided to support backward compatibility with previous generations of this architecture.

User Programming Model

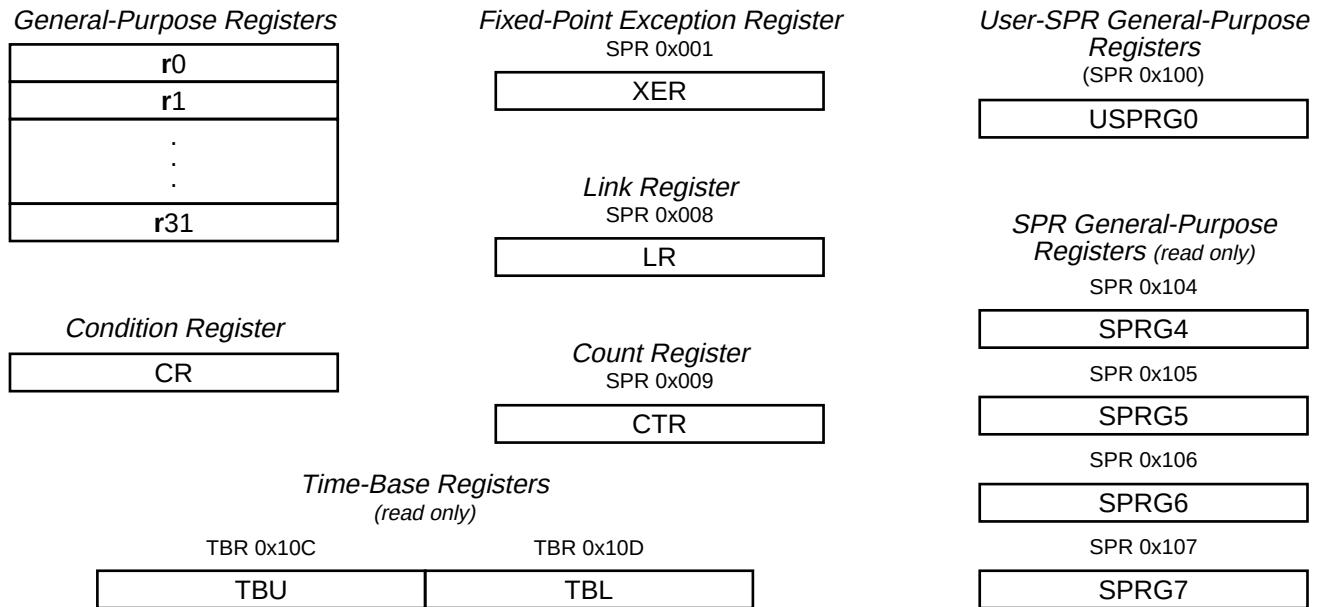
This chapter describes the processor resources and instructions available to all programs running on the PPC405, whether they are running in user mode or privileged mode. These resources and instructions are referred to as the *user-programming model*, which is a subset of the privileged-programming model. Applications are typically restricted to running in user mode. System software runs in privileged mode and has access to all register processor resources, and can execute all instructions supported by the PPC405. System software typically creates a context (execution environment) that protects itself and other applications from the effects of an errant application program.

The remaining chapters in this book generally describe aspects of the privileged-programming model and are not relevant to application programmers. There are two exceptions:

- **Chapter 5, Memory-System Management**, describes cache management features available to both system and application programs.
- **Chapter 8, Timer Resources**, describes the time base, which can be read by application programs.

User Registers

Figure 3-1 shows the user registers supported by the PPC405, all of which are available to software running in user mode and privileged mode. In the PPC405, all user registers are 32-bits wide, except for the time base as described in **Time Base**, **page 560**. Floating-point registers are not supported by the PPC405.



UG011_30_033101

Figure 3-1: PPC405 User Registers

Special-Purpose Registers (SPRs)

Most registers in the PPC405 are *special-purpose registers*, or SPRs. SPRs control the operation of debug facilities, timers, interrupts, storage control attributes, and other processor resources. All SPRs can be accessed explicitly using the *move to special-purpose register* (**mtspr**) and *move from special-purpose register* (**mfspr**) instructions. See **Special-Purpose Register Instructions**, [page 460](#) for more information on these instructions. A few registers are accessed as a by-product of executing certain instructions. For example, some branch instructions access and update the link register.

The PPC405 SPRs in the user-programming model are shown in [Figure 3-1](#). The SPR number (SPRN) for each SPR is shown above the corresponding register. See [Appendix A, Special-Purpose Registers](#), [page 806](#) for a complete list of all SPRs (user and privileged) supported by the PPC405.

Simplified instruction mnemonics are available for the **mtspr** and **mfspr** instructions for some SPRs. See **Special-Purpose Registers**, [page 866](#) for more information.

General-Purpose Registers (GPRs)

The PPC405 contains thirty-two 32-bit general-purpose registers (GPRs), numbered r0 through r31, as shown in [Figure 3-2](#). Data from memory are read into GPRs using load instructions and the contents of GPRs are written to memory using store instructions. Most integer instructions use the GPRs for source and destination operands.

0

31

Figure 3-2: General Purpose Registers (R0-R31)

Condition Register (CR)

The condition register (CR) is a 32-bit register that reflects the result of certain instructions and provides a mechanism for testing and conditional branching. The bits in the CR are grouped into eight 4-bit fields, CR0–CR7, as shown in [Figure 3-3](#). The bits within an arbitrary CR n field are shown in [Figure 3-4](#). In this figure, the bit positions shown are relative positions within the field rather than absolute positions within the CR register.

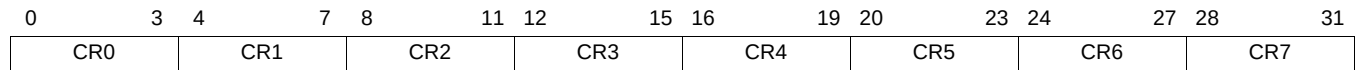


Figure 3-3: Condition Register (CR)

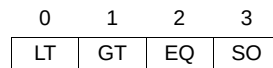


Figure 3-4: CR n Field

In the PPC405, the CR fields are modified in the following ways:

- The **mtcrf** instruction can update specific fields in the CR from a GPR.
- The **mcrxr** instruction can update a CR field with the contents of XER[0:3].
- The **mcrf** instruction can copy one CR field into another CR field.
- The condition-register logical instructions can update specific bits in the CR.
- The integer-arithmetic instructions can update CR0 to reflect their result.
- The integer-compare instructions can update a specific CR field to reflect their result.

Conditional-branch instructions can test bits in the CR and use the results of such a test as the branch condition.

CR0 Field

The CR0 field is updated to reflect the result of an integer instruction if the Rc opcode field (record bit) is set to 1. The **addic.**, **andi.**, and **andis.** instructions also update CR0 to reflect the result they produce. For all of these instructions, CR0 is updated as follows:

- The instruction result is interpreted as a signed integer and algebraically compared to 0. The first three bits of CR0 (CR0[0:2]) are updated to reflect the result of the algebraic comparison.
- The fourth bit of CR0 (CR0[3]) is copied from XER[SO].

The CR0 bits are interpreted as described in [Table 3-1](#). If any portion of the result is undefined, the value written into CR0[0:2] is undefined.

Table 3-1: CR0-Field Bit Settings

Bit	Name	Function	Description
0	LT	Negative 0—Result is not negative. 1—Result is negative.	This bit is set when the result is negative, otherwise it is cleared.
1	GT	Positive 0—Result is not positive. 1—Result is positive.	This bit is set when the result is positive (and not zero), otherwise it is cleared.
2	EQ	Zero 0—Result is not equal to zero. 1—Result is equal to zero.	This bit is set when the result is zero, otherwise it is cleared.
3	SO	Summary overflow 0—No overflow occurred. 1—Overflow occurred.	This is a copy of the final state of XER[SO] at the completion of the instruction.

CR1 Field

In PowerPC® implementations that support floating-point operations, the CR1 field can be updated by the processor to reflect the result of those operations. Because the PPC405 does not support floating-point operations in hardware, CR1 is not updated in this manner.

CR_n Fields (Compare Instructions)

Any one of the eight CR_n fields (including CR0 and CR1) can be updated to reflect the result of a compare instruction. The CR_n-field bits are interpreted as described in Table 3-2.

Table 3-2: CR_n-Field Bit Settings

Bit	Name	Function	Description
0	LT	Less than 0—rA is not less than. 1—rA is less than.	This bit is set when rA < SIMM or rB (signed comparison), or rA < UIMM or rB (unsigned comparison), otherwise it is cleared.
1	GT	Greater than 0—rA is not greater than. 1—rA is greater than.	This bit is set when rA > SIMM or rB (signed comparison), or rA > UIMM or rB (unsigned comparison), otherwise it is cleared.
2	EQ	Equal to 0—rA is not equal. 1—rA is equal.	This bit is set when rA = SIMM or rB (signed comparison), or rA = UIMM or rB (unsigned comparison), otherwise it is cleared.
3	SO	Summary overflow 0—No overflow occurred. 1—Overflow occurred.	This is a copy of the final state of XER[SO] at the completion of the instruction.

Fixed-Point Exception Register (XER)

The fixed-point exception register (XER) is a 32-bit register that reflects the result of arithmetic operations that have resulted in an overflow or carry. This register is also used to indicate the number of bytes to be transferred by load/store string indexed instructions. [Figure 3-5](#) shows the format of the XER. The bits in the XER are defined as shown in [Table 3-3](#).



Figure 3-5: Fixed Point Exception Register (XER)

Table 3-3: Fixed Point Exception Register (XER) Bit Definitions

Bit	Name	Function	Description
0	SO	Summary overflow 0—No overflow occurred. 1—Overflow occurred.	SO is set to 1 whenever an instruction (except mtspr) sets the overflow bit (XER[OV]). Once set, the SO bit remains set until it is cleared to 0 by an mtspr instruction (specifying the XER) or an mcrxr instruction. SO can be cleared to 0 and OV set to 1 using an mtspr instruction.
1	OV	Overflow 0—No overflow occurred. 1—Overflow occurred.	OV can be modified by instructions when the overflow-enable bit in the instruction encoding is set (OE=1). Add, subtract, and negate instructions set OV=1 if the carry out from the result msb is not equal to the carry out from the result msb + 1. Otherwise, they clear OV=0. Multiply and divide set OV=1 if the result cannot be represented in 32 bits. mtspr can be used to set OV=1, and mtspr and mcrxr can be used to clear OV=0.
2	CA	Carry 0—Carry did not occur. 1—Carry occurred.	CA can be modified by <i>add-carrying</i> , <i>subtract-from-carrying</i> , <i>add-extended</i> , and <i>subtract-from-extended</i> instructions. These instructions set CA=1 when there is a carry out from the result msb. Otherwise, they clear CA=0. Shift-right algebraic instructions set CA=1 if any 1 bits are shifted out of a negative operand. Otherwise, they clear CA=0. mtspr can be used to set CA=1, and mtspr and mcrxr can be used to clear CA=0.
3:24		Reserved	
25:31	TBC	Transfer-byte count	TBC is modified using the mtspr instruction. It specifies the number of bytes to be transferred by a <i>load-string word indexed (lswx)</i> or <i>store-string word indexed (stswx)</i> instruction.

The XER is an SPR with an address of 1 (0x001) and can be read and written using the **mfspir** and **mtspir** instructions. The **mcrxr** instruction can be used to move XER[0:3] into one of the seven CR fields.

Link Register (LR)

The link register (LR) is a 32-bit register that is used by branch instructions, generally for the purpose of subroutine linkage. Two types of branch instructions use the link register:

- *Branch-conditional to link-register (bclrx)* instructions read the branch-target address from the LR.
- Branch instructions with the link-register update-option enabled load the LR with the effective address of the instruction following the branch instruction. The link-register update-option is enabled when the branch-instruction LK opcode field (bit 31) is set to 1.

The format of LR is shown in [Figure 3-6](#).

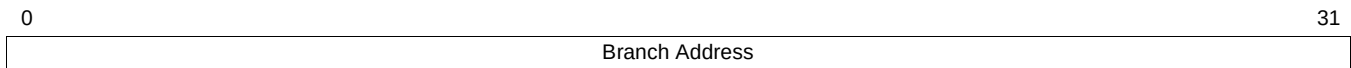


Figure 3-6: Link Register (LR)

The LR is an SPR with an address of 8 (0x008) and can be read and written using the **mf spr** and **mt spr** instructions. It is possible for the processor to prefetch instructions along the target path specified by the LR provided the LR is loaded sufficiently ahead of the branch to link-register instruction, giving branch-prediction hardware time to calculate the branch address.

The two least-significant bits (LR[30:31]) can be written with any value. However, those bits are ignored and assumed to have a value of 0 when the LR is used as a branch-target address.

Some PowerPC processors implement a software-invisible *link-register stack* for performance reasons. Although the PPC405 processor does not implement such a stack, certain programming conventions should be followed so that software running on multiple PowerPC processors can benefit from this stack. See **Link-Register Stack**, [page 407](#) for more information.

Count Register (CTR)

The count register (CTR) is a 32-bit register that can be used by branch instructions in the following two ways:

- The CTR can hold a loop count that is decremented by a conditional-branch instruction with an appropriately coded BO opcode field. The value in the CTR wraps to 0xFFFF_FFFF if the value in the register is 0 prior to the decrement. See **Conditional Branch Control**, [page 403](#) for information on encoding the BO opcode field.
- The CTR can hold the branch-target address used by *branch-conditional to count-register* (**bcctrx**) instructions.

The format of CTR is shown in [Figure 3-7](#).

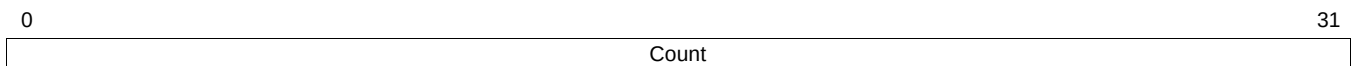


Figure 3-7: Count Register (CTR)

The CTR is an SPR with an address of 9 (0x009) and can be read and written using the **mf spr** and **mt spr** instructions. It is possible for the processor to prefetch instructions along the target path specified by the CTR provided the CTR is loaded sufficiently ahead of the branch to count-register instruction, giving branch-prediction hardware time to calculate the branch address.

The two least-significant bits (CTR[30:31]) can be written with any value. However, those bits are ignored and assumed to have a value of 0 when the CTR is used as a branch-target address.

User-SPR General-Purpose Register

The user-SPR general-purpose register (USPRG0) is a 32-bit register that can be used by application software for any purpose. The value stored in this register does not have an effect on the operation of the PPC405 processor.

The format of USPRG0 is shown in [Figure 3-8](#).

0	31
General-Purpose Application-Software Data	

Figure 3-8: User SPR General-Purpose Register (USPRG0)

The USPRG0 is an SPR with an address of 256 (0x100) and can be read and written using the **mf spr** and **mt spr** instructions.

SPR General-Purpose Registers

The SPR general-purpose registers (SPRG0–SPRG7) are 32-bit registers that can be used by system software for any purpose. Four of the registers (SPRG4–SPRG7) are available from user mode with *read-only access*. Application software can read the contents of SPRG4–SPRG7, but cannot modify them. The values stored in these registers do not affect the operation of the PPC405 processor.

The format of all SPRG n registers is shown in [Figure 3-9](#).

0	31
General-Purpose System-Software Data	

Figure 3-9: SPR General-Purpose Registers (SPRG4–SPRG7)

The SPRG n registers are SPRs with the following addresses:

- SPRG4—260 (0x104).
- SPRG5—261 (0x105).
- SPRG6—262 (0x106).
- SPRG7—263 (0x107).

These registers can be read using the **mf spr** instruction. In privileged mode, system software accesses these registers using different SPR numbers (see [page 468](#)).

Time-Base Registers

The time base is a 64-bit incrementing counter implemented as two 32-bit registers. The time-base upper register (TBU) holds time-base bits 0:31, and the time-base lower register (TBL) holds time-base bits 32:63. [Figure 3-10](#) shows the format of the time base.

0	31
TBU (Time Base [0:31])	
0	31
TBL (Time Base [32:63])	

Figure 3-10: Time-Base Register

The TBU and TBL registers are SPRs with user-mode read access and privileged-mode write access. Reading the time-base registers requires use of the **mftb** instruction with the following addresses:

- TBU—269 (0x10D).
- TBL—268 (0x10C).

See [Time Base](#), [page 560](#), for information on using the time base.

Exception Summary

An exception is an event that can be caused by a number of sources, including:

- Error conditions arising from instruction execution.
- Internal timer resources.
- Internal debug resources.
- External peripherals.

When an exception occurs, the processor can interrupt the currently executing program so that system software can deal with the exception condition. The action taken by an interrupt includes saving the processor context and transferring control to a predetermined exception-handler address operating under a new context. When the interrupt handler completes execution, it can return to the interrupted program by executing a *return-from-interrupt* instruction.

Exceptions are handled by privileged software. The exception mechanism is described in **Chapter 7, Exceptions and Interrupts**. Following is a list of exceptions that can be caused by the execution of an instruction in user mode.

- Data-Storage Exception.

An attempt to access data in memory that results in a memory-protection violation causes the data-storage interrupt handler to be invoked.

- Instruction-Storage Exception.

An attempt to access instructions in memory that result in a memory-protection violation causes the instruction-storage interrupt handler to be invoked.

- Alignment Exception.

An attempt to access memory with an invalid effective-address alignment (for the specific instruction) causes the alignment-interrupt handler to be invoked.

- Program Exception.

Three different types of interrupt handlers can be invoked when a program exception occurs: illegal instruction, privileged instruction, and system trap. The conditions causing a program interrupt include:

- An attempt to execute an illegal instruction causes the illegal-instruction interrupt handler to be invoked.
- An attempt to execute an optional instruction not implemented by the PPC405 causes the illegal-instruction interrupt handler to be invoked.
- An attempt by a user-level program to execute a supervisor-level instruction causes the privileged-instruction interrupt handler to be invoked.
- An attempt to execute a defined instruction with an invalid form causes either the illegal-instruction interrupt handler or the privileged-instruction interrupt handler to be invoked.
- Executing a trap instruction can cause the system-trap interrupt handler to be invoked.

- Floating-Point Unavailable Exception.

On processors that support floating-point instructions, executing such instructions when the floating-point unit is disabled (MSR[FP]=0) invokes the floating-point-unavailable interrupt handler.

- System-Call Exception.

The execution of an *sc* instruction causes the system-call interrupt handler to be invoked. The interrupt handler can be used to call a system-service routine.

- Data TLB-Miss Exception.

If data translation is enabled, an attempt to access data in memory when a valid TLB entry is not present causes the data TLB-miss interrupt handler to be invoked.

- Instruction TLB-Miss Exception.

If instruction translation is enabled, an attempt to access instructions in memory when a valid TLB entry is not present causes the instruction TLB-miss interrupt handler to be invoked.

Other exceptions can occur during user-mode program execution that are not directly caused by instruction execution. These are also described in [Chapter 7](#):

- Machine-check exceptions.
- Exceptions caused by external devices.
- Exceptions caused by a timer.
- Debug exceptions.

Branch and Flow-Control Instructions

Branch instructions redirect program flow by altering the next-instruction address non-sequentially. Branches unconditionally or conditionally alter program flow forward or backward using either an absolute address or an address relative to the branch-instruction address. Branches calculate the target address using the contents of the CTR, LR, or fields within the branch instruction. Optionally, a branch-return address can be automatically loaded into the LR by setting the LK instruction-opcode bit to 1. This option is useful for specifying the return address for subroutine calls and causes the address of the instruction following the branch to be loaded in the LR. Branches are used for all non-sequential program flow including jumps, loops, calls and returns.

Branch-conditional instructions redirect program flow if a tested condition is true. These instructions can test a bit value within the CR, the value of the CTR, or both. Condition-register logical instructions are provided to set up the tests for branch-conditional instructions.

Conditional Branch Control

With branch-conditional instructions, the BO opcode field specifies the branch-control conditions and how the branch affects the CTR. The BO field can specify a test of the CR and it can specify that the CTR be decremented and tested. The BO field can also be initialized to reverse the default prediction performed by the processor. The bits within the BO field are defined as shown in [Table 3-4](#).

Table 3-4: BO Field Bit Definitions

BO Bit	Description
BO[0]	CR Test Control 0—Test the CR bit specified by the BI opcode field for the value indicated by BO[1]. 1—Do not test the CR.
BO[1]	CR Test Value 0—Test for CR[BI]=0. 1—Test for CR[BI]=1.

Table 3-4: BO Field Bit Definitions (Continued)

BO Bit	Description
BO[2]	CTR Test Control 0—Decrement CTR by one, and test whether CTR satisfies the condition specified by BO[3]. 1—Do not change or test CTR.
BO[3]	CTR Test Value 0—Test for CTR ≠ 0. 1—Test for CTR=0.
BO[4]	Branch Prediction Reversal 0—Apply standard branch prediction. 1—Reverse the standard branch prediction.

The 5-bit BI opcode field in branch-conditional instructions specifies which of the 32 bits in the CR are used in the branch-condition test. For example, if BI=0b01010, CR₁₀ is used in the test.

In some encodings of the BO field, certain BO bits are ignored. Ignored bits can be assigned a meaning in future extensions of the PowerPC architecture and should be cleared to 0. Valid BO field encodings are shown in Table 3-5. In this table, z indicates the ignored bits that should be cleared to 0. The y bit (BO[4]) specifies the branch-prediction behavior for the instruction as described in **Specifying Branch-Prediction Behavior**, page 406.

Table 3-5: Valid BO Opcode-Field Encoding

BO[0:4]	Description
0000y	Decrement the CTR. Branch if the decremented CTR ≠ 0 and CR[BI]=0.
0001y	Decrement the CTR. Branch if the decremented CTR = 0 and CR[BI]=0.
001zy	Branch if CR[BI]=0.
0100y	Decrement the CTR. Branch if the decremented CTR ≠ 0 and CR[BI]=1.
0101y	Decrement the CTR. Branch if the decremented CTR=0 and CR[BI]=1.
011zy	Branch if CR[BI]=1.
1z00y	Decrement the CTR. Branch if the decremented CTR ≠ 0.
1z01y	Decrement the CTR. Branch if the decremented CTR = 0.
1z1zz	Branch always.

Branch Instructions

The following sections describe the branch instructions defined by the PowerPC architecture. A number of simplified mnemonics are defined for the branch instructions. See **Branch Instructions**, page 857 for more information.

Branch Unconditional

Table 3-6 lists the PowerPC *unconditional branch* instructions. These branches specify a 26-bit signed displacement to the branch-target address by appending the 24-bit LI instruction field with 0b00. The displacement value gives unconditional branches the ability to cover an address range of ±32 MB.

Table 3-6: Branch-Unconditional Instructions

Mnemonic	Name	Operation	Operand Syntax
b	Branch	Branch to relative address..	tgt_addr
ba	Branch Absolute	Branch to absolute address.	
bl	Branch and Link	Branch to relative address. LR is updated with the address of the instruction following the branch.	
bla	Branch Absolute and Link	Branch to absolute address. LR is updated with the address of the instruction following the branch.	

Branch Conditional

Table 3-7 lists the PowerPC *branch-conditional* instructions. The BO field specifies the condition tested by the branch, as shown in **Table 3-5, page 404**. The BI field specifies the CR bit used in the test. These branches specify a 16-bit signed displacement to the branch-target address by appending the 14-bit BD instruction field with 0b00. The displacement value gives conditional branches the ability to cover an address range of ± 32 KB.

Table 3-7: Branch-Conditional Instructions

Mnemonic	Name	Operation	Operand Syntax
bc	Branch Conditional	Branch-conditional to relative address..	BO,BI,tgt_addr
bca	Branch Conditional Absolute	Branch-conditional to absolute address.	
bcl	Branch Conditional and Link	Branch-conditional to relative address. LR is updated with the address of the instruction following the branch.	
bcla	Branch Conditional Absolute and Link	Branch-conditional to absolute address. LR is updated with the address of the instruction following the branch.	

Branch Conditional to Link Register

Table 3-8 lists the PowerPC *branch-conditional to link-register* instructions. The BO field specifies the condition tested by the branch, as shown in **Table 3-5, page 404**. The BI field specifies the CR bit used in the test. The branch-target address is read from the LR, with LR[30:31] cleared to zero to form a word-aligned address. Using the 32-bit LR as a branch target gives these branches the ability to cover the full 4 GB address range.

Table 3-8: Branch-Conditional to Link-Register Instructions

Mnemonic	Name	Operation	Operand Syntax
bclr	Branch Conditional to Link Register	Branch-conditional to address in LR.	BO,BI
bclrl	Branch Conditional to Link Register and Link	Branch-conditional to address in LR. LR is updated with the address of the instruction following the branch.	

Branch Conditional to Count Register

Table 3-9 lists the PowerPC *branch-conditional to count-register* instructions. The BO field specifies the condition tested by the branch, as shown in **Table 3-5, page 404**. The BI field specifies the CR bit used in the test. The branch-target address is read from the CTR, with CTR[30:31] cleared to zero to form a word-aligned address. Using the 32-bit CTR as a branch target gives these branches the ability to cover the full 4 GB address range.

Table 3-9: Branch-Conditional to Count-Register Instructions

Mnemonic	Name	Operation	Operand Syntax
bcctr	Branch Conditional to Count Register	Branch-conditional to address in CTR.	BO, BI
bcctrl	Branch Conditional to Count Register and Link	Branch-conditional to address in CTR. LR is updated with the address of the instruction following the branch.	

Branch Prediction

Conditional branches alter program flow based on the value of bits in the CR. If a condition is met by the CR bits, the branch instruction alters the next-instruction address non-sequentially. Otherwise, the next-sequential instruction following the branch is executed. When the processor encounters a conditional branch, it scans the execution pipelines to determine whether an instruction in progress can affect the CR bit tested by the branch. If no such instruction is found, the branch can be resolved immediately by checking the bit in the CR and taking the action defined by the branch instruction.

However, if a CR-altering instruction is detected, the branch is considered unresolved until the CR-altering instruction completes execution and writes its result to the CR. Prior to that time, the processor can *predict* how the branch is resolved. First, the processor uses special *dynamic prediction* hardware to analyze instruction flow and branch history to predict resolution of the current branch. If branches are predicted correctly, performance improvements can be realized because instruction execution does not stall waiting for the branch to be resolved. The PowerPC architecture provides software with the ability to override (reverse) the dynamic prediction using a *static prediction* hint encoded in the instruction opcode. This can be useful when it is known at compile time that a branch is likely to behave contrary to what the processor expects. The use of static prediction is described in the next section, **Specifying Branch-Prediction Behavior**.

When a prediction is made, instructions are fetched from the predicted execution path. If the processor determines the prediction was incorrect after the CR-altering instruction completes execution, all instructions fetched as a result of the prediction are discarded by the processor. Instruction fetch is restarted along the correct path. If the prediction was correct, instruction fetch and execution proceed normally along the predicted (and now resolved) path.

Branch prediction is most effective when the branch-target address is computed well in advance of resolving the branch. If a branch instruction contains immediate addressing operands, the processor can compute the branch-target address ahead of branch resolution. If the branch instruction uses the LR or CTR for addressing, it is important that the register is loaded by software sufficiently ahead of the branch instruction.

Specifying Branch-Prediction Behavior

All PowerPC processors predict a conditional branch as taken using the following rules:

- For the **bcx** instruction with a negative value in the displacement operand, the branch is predicted taken.
- For all other branch-conditional instructions (**bcx** with a non-negative value in the displacement operand, **bclrx**, or **bcctrx**), the branch is predicted not taken.

Algorithmically, a branch is predicted taken if:

$$((BO[0] \wedge BO[2]) \vee s) = 1$$

where s is the sign bit of the displacement operand, if the instruction has a displacement operand (bit 16 of the branch-conditional instruction encoding).

When the result of the above equation is 0, the branch is predicted not-taken and the processor speculatively fetches instructions that sequentially follow the branch instruction.

Examining the above equation, $BO[0] \wedge BO[2]=1$ only when the conditional branch tests nothing, meaning the branch is always taken. In this case, the processor predicts the branch as taken.

If the conditional branch tests anything ($BO[0] \wedge BO[2]=0$), s controls the prediction. In the **bclrx** and **bcctrx** instructions, bit 16 (s) is reserved and always 0. In this case those instructions are predicted not-taken.

Only the **bcx** instructions can specify a displacement value. The **bcx** instructions are commonly used at the end of loops to control the number of times a loop is executed. Here, the branch is taken every time the loop is executed except the last time, so a branch should normally be predicted as taken. Because the branch target is at the beginning of the loop, the branch displacement is negative and $s=1$, so the processor predicts the branch as taken. Forward branches have a positive displacement and are predicted not-taken.

When the y bit ($BO[4]$) is cleared to 0, the default branch prediction behavior described above is followed by the processor. Setting the y bit to 1 *reverses* the above behavior. For *branch always* encoding ($BO[0]$, $BO[2]$), branch prediction cannot be reversed (no y bit is recognized).

The sign of the displacement operand (s) is used as described above even when the target is an absolute address. The default value for the y bit should be 0. Compilers can set this bit if it they determine that the prediction corresponding to $y=1$ is more likely to be correct than the prediction corresponding to $y=0$. Compilers that do not statically predict branches should always clear the y bit.

Link-Register Stack

Some processor implementations keep a stack (history) of the LR values most recently used by branch-and-link instructions. Those processors use this software-invisible stack to predict the target address of nested-subroutine returns. Although the PPC405 processor does not implement such a stack, the following programming conventions should be followed so that software running on multiple PowerPC processors can benefit from this stack.

In the following examples, let A , B , and $Glue$ represent subroutine labels:

- When obtaining the address of the next instruction, use the following form of branch-and-link:
bcl 20,31,\$+4
- Loop counts:
Keep loop counts in the CTR, and use one of the branch-conditional instructions to decrement the count and to control branching (for example, branching back to the start of a loop if the decremented CTR value is nonzero).
- Computed “go to”, case statements, etc.:
Use the CTR to hold the branch-target address, and use the **bcctr** instruction with the link register option disabled ($LK=0$) to branch to the selected address.
- Direct subroutine linkage, where A calls B and B returns to A :
 - A calls B —use a branch instruction that enables the LR ($LK=1$).

- *B* returns to *A*—use the **bclr** instruction with the link-register option disabled (LK=0). The return address is in, or can be restored to, the LR.
- Indirect subroutine linkage, where *A* calls *Glue*, *Glue* calls *B*, and *B* returns to *A* rather than to *Glue*.

Such a calling sequence is common in linkage code where the subroutine that the programmer wants to call, *B*, is in a different module than the caller, *A*. The binder inserts “glue” code to mediate the branch:

- *A* calls *Glue*—use a branch instruction that sets the LR with the link-register option enabled (LK=1).
- *Glue* calls *B*—write the address of *B* in the CTR, and use the **bcctr** instruction with the link-register option disabled (LK=0).
- *B* returns to *A*—use the **bclr** instruction with the link-register option disabled (LK=0). The return address is in, or can be restored to, the LR.

Branch-Target Address Calculation

Branch instructions compute the effective address (EA) of the next instruction using the following addressing modes:

- Branch to relative (conditional and unconditional).
- Branch to absolute (conditional and unconditional).
- Branch to link register (conditional only).
- Branch to count register (conditional only).

Instruction addresses are always assumed to be word aligned. PowerPC processors ignore the two low-order bits of the generated branch-target address.

Branch to Relative

Instructions that use *branch-to-relative* addressing generate the next-instruction address by right-extending 0b00 to the immediate-displacement operand (LI), and then sign-extending the result. That result is added to the current-instruction address to produce the next-instruction address. Branches using this addressing mode must have the absolute-addressing option disabled by clearing the AA instruction field (bit 30) to 0. The link-register update option is enabled by setting the LK instruction field (bit 31) to 1. This option causes the effective address of the instruction following the branch instruction to be loaded into the LR.

Figure 3-11 shows how the branch-target address is generated when using the branch-to-relative addressing mode.

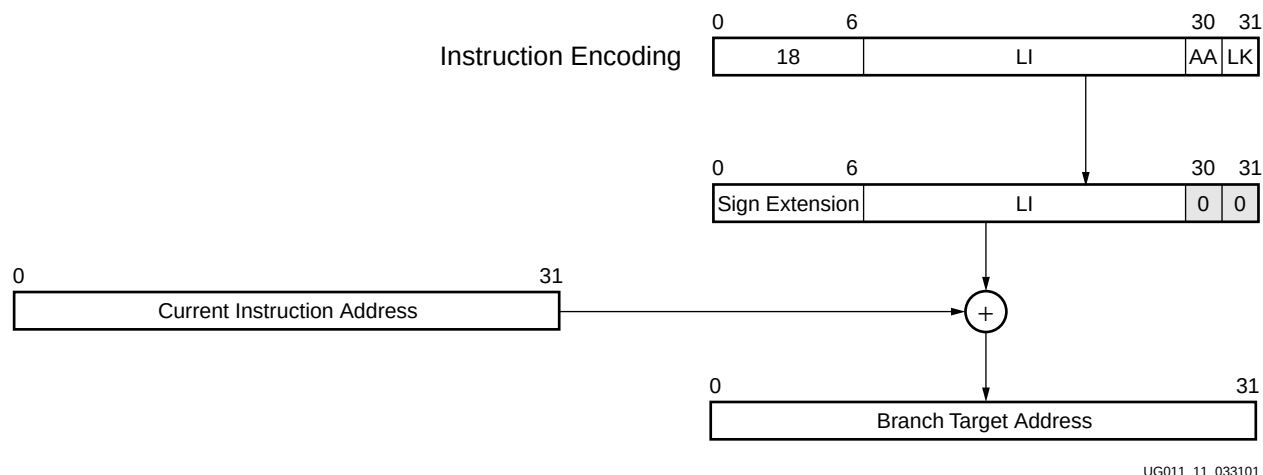


Figure 3-11: Branch-to-Relative Addressing

Branch-Conditional to Relative

If the branch conditions are met, instructions that use *branch-conditional to relative* addressing generate the next-instruction address by appending 0b00 to the immediate-displacement operand (BD) and sign-extending the result. That result is added to the current-instruction address to produce the next-instruction address. Branches using this addressing mode must have the absolute-addressing option disabled by clearing the AA instruction field (bit 30) to 0. The link-register update option is enabled by setting the LK instruction field (bit 31) to 1. This option causes the effective address of the instruction following the branch instruction to be loaded into the LR.

Figure 3-12 shows how the branch-target address is generated when using the branch-conditional to relative addressing mode.

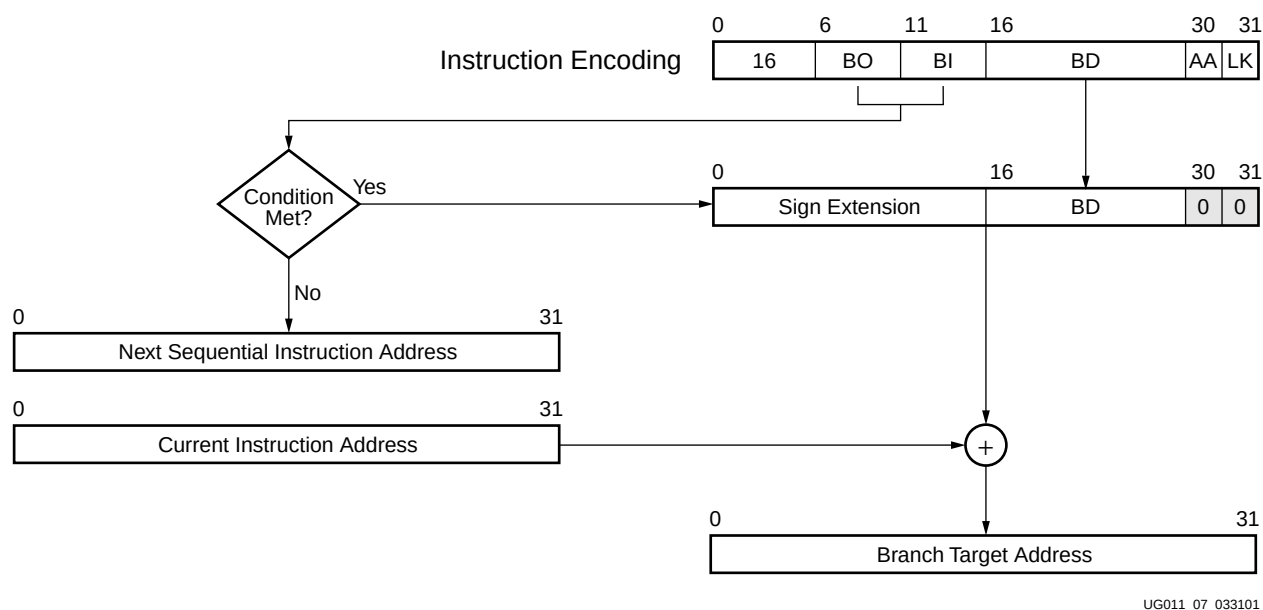


Figure 3-12: Branch-Conditional to Relative Addressing

Branch to Absolute

Instructions that use *branch-to-absolute* addressing generate the next-instruction address by appending 0b00 to the immediate-displacement operand (LI) and sign-extending the result. Branches using this addressing mode must have the absolute-addressing option enabled by setting the AA instruction field (bit 30) to 1. The link-register update option is enabled by setting the LK instruction field (bit 31) to 1. This option causes the effective address of the instruction following the branch instruction to be loaded into the LR.

Figure 3-13 shows how the branch-target address is generated when using the branch-to-absolute addressing mode.

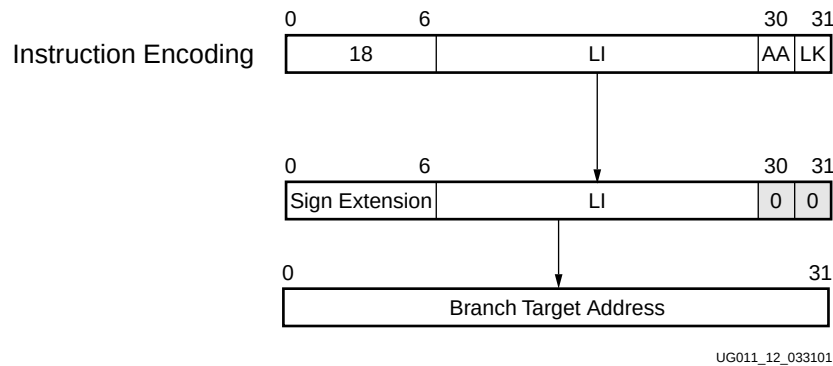


Figure 3-13: Branch-to-Absolute Addressing

Branch-Conditional to Absolute

If the branch conditions are met, instructions that use *branch-conditional to absolute* addressing generate the next-instruction address by appending 0b00 to the immediate-displacement operand (BD) and sign-extending the result. Branches using this addressing mode must have the absolute-addressing option enabled by setting the AA instruction field (bit 30) to 1. The link-register update option is enabled by setting the LK instruction field (bit 31) to 1. This option causes the effective address of the instruction following the branch instruction to be loaded into the LR.

Figure 3-14 shows how the branch-target address is generated when using the branch-conditional to absolute-addressing mode.

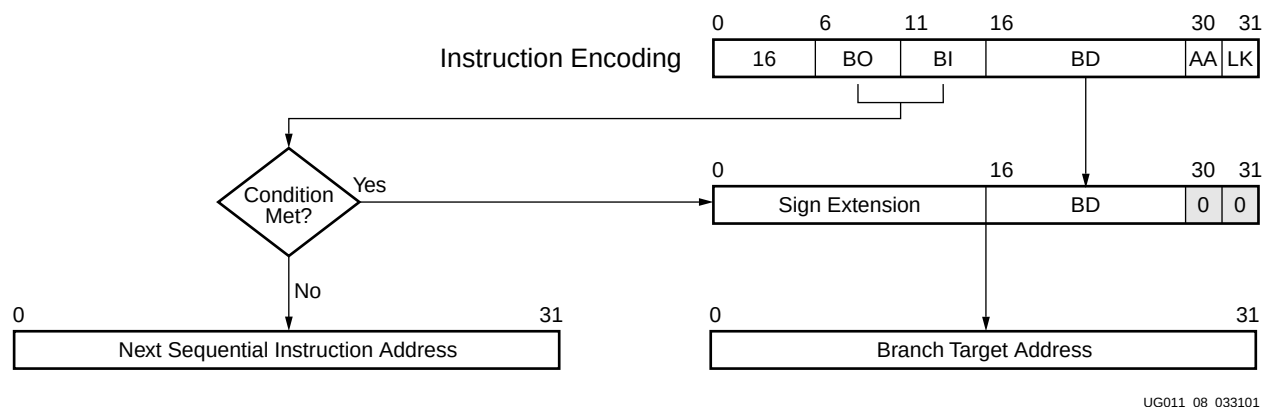
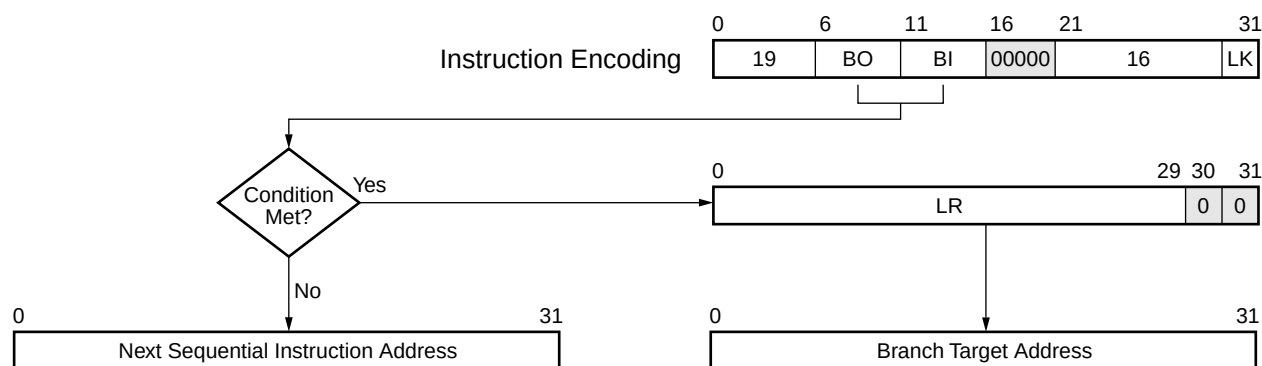


Figure 3-14: Branch-Conditional to Absolute Addressing

Branch-Conditional to Link Register

If the branch conditions are met, the *branch-conditional to link-register* instruction generates the next-instruction address by reading the contents of the LR and clearing the two low-order bits to zero. The link-register update option is enabled by setting the LK instruction field (bit 31) to 1. This option causes the effective address of the instruction following the branch instruction to be loaded into the LR.

Figure 3-15 shows how the branch-target address is generated when using the branch-conditional to link-register addressing mode.



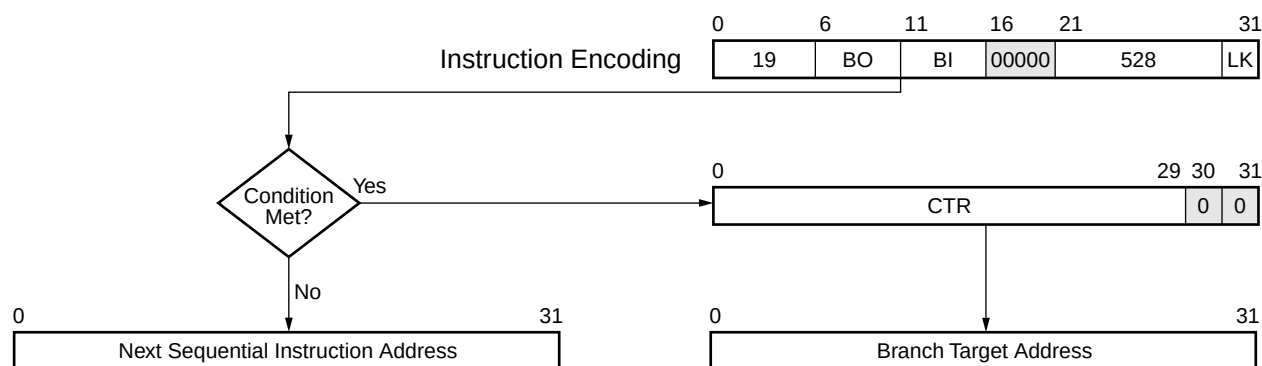
UG011_09_033101

Figure 3-15: Branch-Conditional to Link-Register Addressing

Branch-Conditional to Count Register

If the branch conditions are met, the *branch-conditional to count-register* instruction generates the next-instruction address by reading the contents of the CTR and clearing the two low-order bits to zero. The link-register update option is enabled by setting the LK instruction field (bit 31) to 1. This option causes the effective address of the instruction following the branch instruction to be loaded into the LR.

Figure 3-16 shows how the branch-target address is generated when using the branch-conditional to count-register addressing mode.



UG011_10_033101

Figure 3-16: Branch-Conditional to Count-Register Addressing

Condition-Register Logical Instructions

Table 3-10 lists the PowerPC *condition-register logical* instructions. The condition-register logical instructions perform logical operations on any two bits within the CR and store the result of the operation in any CR bit. The *move condition-register field* instruction is used to move any CR field (each field comprising four bits) to any other CR-field location. All of these instructions are considered flow-control instructions because they are generally used to set up conditions for testing by the branch-conditional instructions and to reduce the number of branches in a code sequence. Simplified mnemonics are defined for the condition-register logical instructions. See **CR-Logical Instructions**, [page 864](#) for more information.

In **Table 3-10**, the instruction-operand fields **crbA**, **crbB**, and **crbD** all specify a single *bit* within the CR. The instruction-operand fields **crfD** and **crfS** specify a 4-bit *field* within the CR.

Table 3-10: Condition-Register Logical Instructions

Mnemonic	Name	Operation	Operand Syntax
crand	Condition Register AND	CR-bit crbA is ANDed with CR-bit crbB and the result is stored in CR-bit crbD .	crbD,crbA,crbB
crandc	Condition Register AND with Complement	CR-bit crbA is ANDed with the <i>complement</i> of CR-bit crbB and the result is stored in CR-bit crbD .	
creqv	Condition Register Equivalent	CR-bit crbA is XORed with CR-bit crbB and the <i>complemented</i> result is stored in CR-bit crbD .	
crnand	Condition Register NAND	CR-bit crbA is ANDed with CR-bit crbB and the <i>complemented</i> result is stored in CR-bit crbD .	
crnor	Condition Register NOR	CR-bit crbA is ORed with CR-bit crbB and the <i>complemented</i> result is stored in CR-bit crbD .	
cror	Condition Register OR	CR-bit crbA is ORed with CR-bit crbB and the result is stored in CR-bit crbD .	
crorc	Condition Register OR with Complement	CR-bit crbA is ORed with the <i>complement</i> of CR-bit crbB and the result is stored in CR-bit crbD .	
crxor	Condition Register XOR	CR-bit crbA is XORed with CR-bit crbB and the result is stored in CR-bit crbD .	
mcrf	Move Condition Register Field	CR-field crfS is copied into CR-field crfD . No other CR fields are modified.	crfD,crfS

System Call

Table 3-11 lists the PowerPC *system-call* instruction. The **sc** instruction is a user-level instruction that can be used by a user-mode program to transfer control to a privileged-mode program (typically a system-service routine). Executing the **sc** instruction causes a system-call exception to occur. See **System-Call Interrupt (0x0C00)**, [page 550](#) for more information on the operation of this instruction.

Table 3-11: System-Call Instruction

Mnemonic	Name	Operation	Operand Syntax
sc	System Call	Causes a system-call exception to occur.	—

System Trap

Table 3-12 lists the PowerPC *system-trap* instructions. System-trap instructions are normally used by software-debug applications to set breakpoints. These instructions test for a specified set of conditions and cause a program exception to occur if any of the conditions are met. If the tested conditions are not met, instruction execution continues normally with the instruction following the system-trap instruction (a program exception does not occur). The system-trap handler can be called from the program-interrupt handler when it is determined that a system-trap instruction caused the exception. See **Program Interrupt (0x0700)**, **page 547** for more information on program exceptions caused by the system-trap instructions.

Trap instructions can also be used to cause a debug exception. See **Trap-Instruction Debug Event**, **page 582** for more information.

Simplified mnemonics are defined for the system-trap instructions. See **Trap Instructions**, **page 868** for more information.

Table 3-12: System-Trap Instructions

Mnemonic	Name	Operation	Operand Syntax
tw	Trap Word	The contents of rA are compared with rB. A program exception occurs if the comparison meets any test condition enabled by the TO operand.	TO,rA,rB
twi	Trap Word Immediate	The contents of rA are compared with the sign-extended SIMM operand. A program exception occurs if the comparison meets any test condition enabled by the TO operand.	TO,rA,SIMM

The TO operand field in the system-trap instructions specifies the test conditions performed on the remaining two operands. Multiple test conditions can be set simultaneously, expanding the number of possible conditions that can cause the trap (program exception). If all bits in the TO operand field are set, the trap always occurs because one of the trap conditions is always met. The bits within the TO field are defined as shown in **Table 3-13**.

Table 3-13: TO Field Bit Definitions

TO Bit	Description
TO[0]	Less-than arithmetic comparison. 0—Ignore trap condition. 1—Trap if first operand is arithmetically less-than second operand.
TO[1]	Greater-than arithmetic comparison. 0—Ignore trap condition. 1—Trap if first operand is arithmetically greater-than second operand.

Table 3-13: TO Field Bit Definitions (Continued)

TO Bit	Description
TO[2]	Equal-to arithmetic comparison. 0—Ignore trap condition. 1—Trap if first operand is arithmetically equal-to second operand.
TO[3]	Less-than unsigned comparison. 0—Ignore trap condition. 1—Trap if first operand is less-than second operand.
TO[4]	Greater-than unsigned comparison. 0—Ignore trap condition. 1—Trap if first operand is greater-than second operand.

Integer Load and Store Instructions

The integer load and store instructions move data between the general-purpose registers and memory. Several types of loads and stores are supported by the PowerPC instruction set:

- Load and zero
- Load algebraic
- Store
- Load with byte reverse and store with byte reverse
- Load multiple and store multiple
- Load string and store string
- Memory synchronization instructions

Memory accesses performed by the load and store instructions can occur out of order. Synchronizing instructions are provided to enforce strict memory-access ordering. See **Synchronizing Instructions**, [page 460](#) for more information.

In general, the PowerPC architecture defines a sequential-execution model. When a store instruction modifies an instruction-memory location, software synchronization is required to ensure subsequent instruction fetches from that location obtain the modified version of the instruction. See **Self-Modifying Code**, [page 503](#) for more information.

Operand-Address Calculation

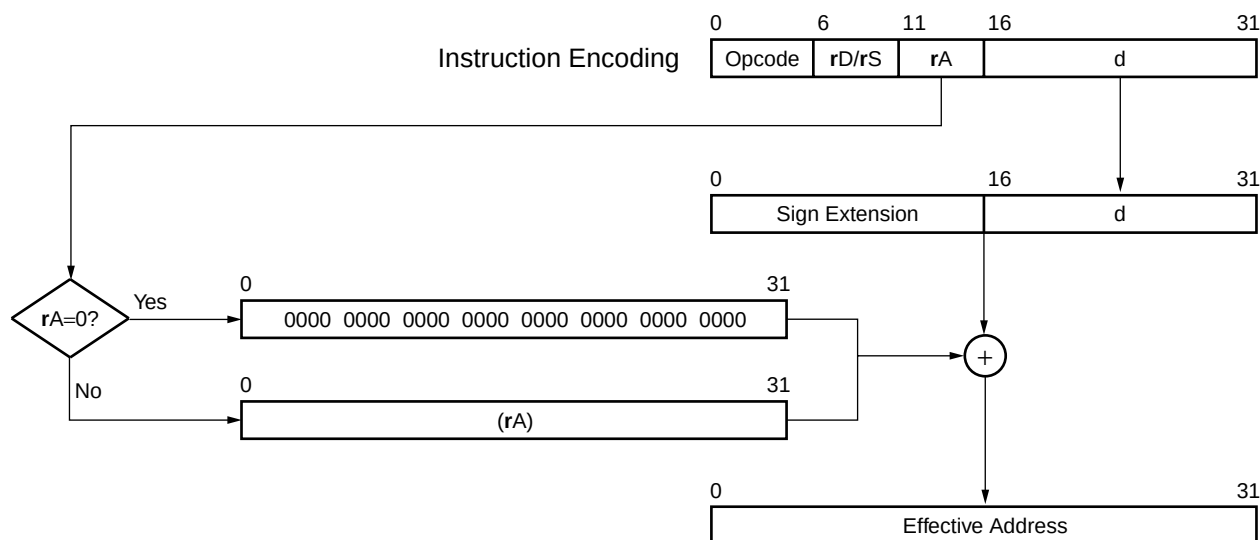
Integer load and store instructions generate effective addresses using one of three addressing modes: register-indirect with immediate index, register-indirect with index, or register indirect. These addressing modes are described in the following sections. For some instructions, update forms that load the calculated effective address into rA are also provided.

In the PPC405 processor, loads and stores to unaligned addresses can suffer from performance degradation. Refer to **Performance Effects of Operand Alignment**, [page 389](#) for more information.

Register-Indirect with Immediate Index

Load and store instructions using this addressing mode contain a signed, 16-bit immediate index (d operand) and a general-purpose register operand, rA. The index is sign-extended to 32 bits and added to the contents of rA to generate the effective address. If the rA instruction field is 0 (specifying r0), a value of zero—rather than the contents of r0—is added to the sign-extended immediate index. The option to specify rA or 0 is shown in the instruction description as (rA | 0).

Figure 3-17 shows how an effective address is generated when using register-indirect with immediate-index addressing.



UG011_02_033101

Figure 3-17: Register-Indirect with Immediate-Index Addressing

Register-Indirect with Index

Load and store instructions using this addressing mode contain two general-purpose register operands, **rA** and **rB**. The contents of these two registers are added to generate the effective address. If the **rA** instruction field is 0 (specifying **r0**), a value of zero—rather than the contents of **r0**—is added to **rB**. The option to specify **rA** or 0 is shown in the instruction description as **(rA | 0)**.

Figure 3-18 shows how an effective address is generated when using register-indirect with index addressing.

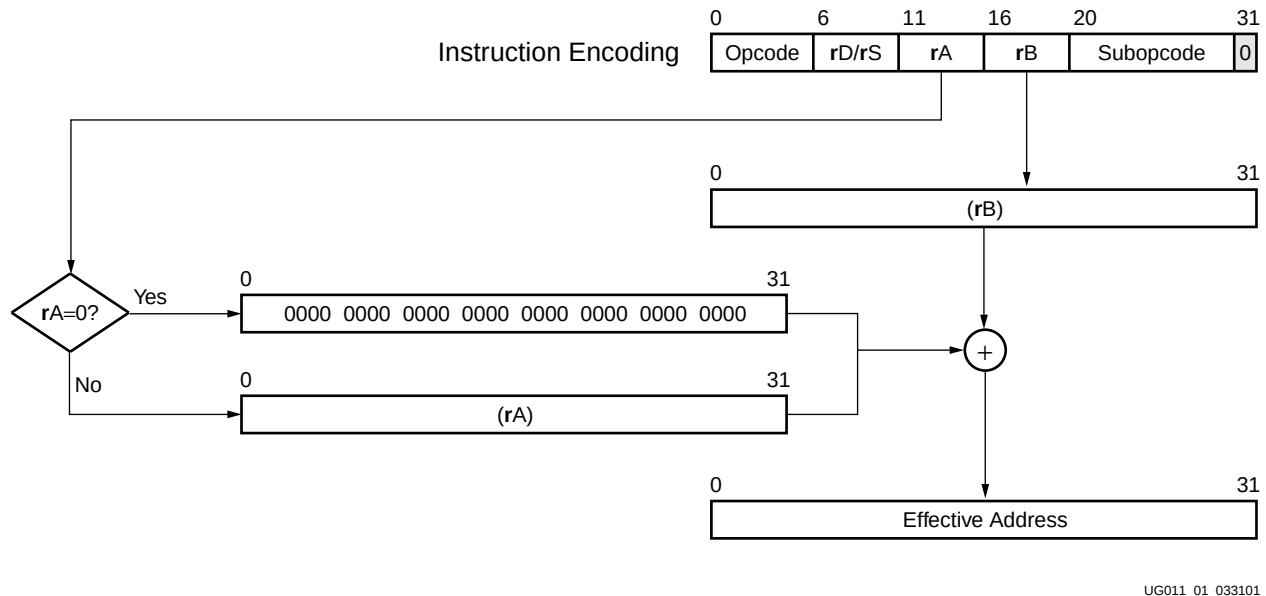


Figure 3-18: Register-Indirect with Index Addressing

Register Indirect

Only load-string and store-string instructions can use this addressing mode. This mode uses only the contents of the general-purpose register specified by the rA operand as the effective address. Rather than using the contents of r0, a zero in the rA operand causes an effective address of zero to be generated. The option to specify rA or 0 is shown in the instruction descriptions as (rA | 0).

Figure 3-19 shows how an effective address is generated when using register-indirect addressing.

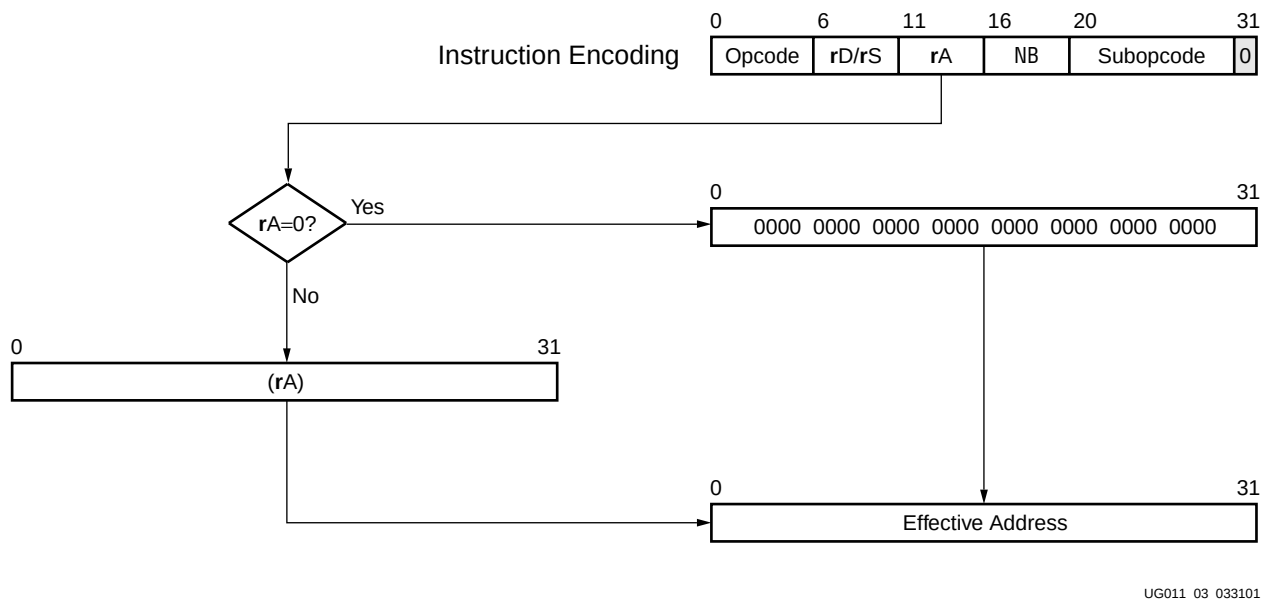


Figure 3-19: Register-Indirect Addressing

Load Instructions

Integer-load instructions read an operand from memory and store it in a GPR destination register, **rD**. Each type of load is characterized by what they do with unused high-order bits in **rD** when the operand size is less than a word (32 bits). *Load-and-zero* instructions clear the unused high-order bits in **rD** to zero. *Load-algebraic* instructions fill the unused high-order bits in **rD** with a copy of the most-significant bit in the operand.

Load-with-update instructions are provided, but the following two rules apply:

- **rA** must not be equal to 0. If **rA** = 0, the instruction form is invalid.
- **rA** must not be equal to **rD**. If **rA** = **rD**, the instruction form is invalid.

In the PPC405, the above invalid instruction forms produce a boundedly-undefined result. In other PowerPC implementations, those forms can cause a program exception.

Load Byte and Zero

Table 3-14 lists the PowerPC *load byte and zero* instructions. These instructions load a byte from memory into the lower-eight bits of **rD** and clear the upper-24 bits of **rD** to 0.

Table 3-14: Load Byte and Zero Instructions

Mnemonic	Name	Addressing Mode	Operand Syntax
lbz	Load Byte and Zero	Register-indirect with immediate index $EA = (rA \mid 0) + d$	rD,d(rA)
lbzu	Load Byte and Zero with Update	Register-indirect with immediate index $EA = (rA) + d$ $rA \leftarrow EA$ $rA \neq 0, rA \neq rD$	
lbzx	Load Byte and Zero Indexed	Register-indirect with index $EA = (rA \mid 0) + (rB)$	rD,rA,rB
lbzux	Load Byte and Zero with Update Indexed	Register-indirect with index $EA = (rA) + (rB)$ $rA \leftarrow EA$ $rA \neq 0, rA \neq rD$	

Load Halfword and Zero

Table 3-15 lists the PowerPC *load halfword and zero* instructions. These instructions load a halfword from memory into the lower-16 bits of **rD** and clear the upper-16 bits of **rD** to 0.

Table 3-15: Load Halfword and Zero Instructions

Mnemonic	Name	Addressing Mode	Operand Syntax
lh _z	Load Halfword and Zero	Register-indirect with immediate index $EA = (rA \mid 0) + d$	rD,d(rA)
lh _{zu}	Load Halfword and Zero with Update	Register-indirect with immediate index $EA = (rA) + d$ $rA \leftarrow EA$ $rA \neq 0, rA \neq rD$	
lh _{zx}	Load Halfword and Zero Indexed	Register-indirect with index $EA = (rA \mid 0) + (rB)$	rD,rA,rB
lh _{zux}	Load Halfword and Zero with Update Indexed	Register-indirect with index $EA = (rA) + (rB)$ $rA \leftarrow EA$ $rA \neq 0, rA \neq rD$	

Load Word and Zero

Table 3-16 lists the PowerPC *load word and zero* instructions. These instructions load a word from memory into rD.

Table 3-16: Load-Word and Zero Instructions

Mnemonic	Name	Addressing Mode	Operand Syntax
lw _z	Load Word and Zero	Register-indirect with immediate index $EA = (rA \mid 0) + d$	rD,d(rA)
lw _{zu}	Load Word and Zero with Update	Register-indirect with immediate index $EA = (rA) + d$ $rA \leftarrow EA$ $rA \neq 0, rA \neq rD$	
lw _{zx}	Load Word and Zero Indexed	Register-indirect with index $EA = (rA \mid 0) + (rB)$	rD,rA,rB
lw _{zux}	Load Word and Zero with Update Indexed	Register-indirect with index $EA = (rA) + (rB)$ $rA \leftarrow EA$ $rA \neq 0, rA \neq rD$	

Load Halfword Algebraic

Table 3-17 lists the PowerPC *load halfword algebraic* instructions. These instructions load a halfword from memory into the lower-16 bits of rD. The upper-16 bits of rD are filled with a copy of the most-significant bit (bit 16) of the operand.

Table 3-17: Load Halfword Algebraic Instructions

Mnemonic	Name	Addressing Mode	Operand Syntax
lha	Load Halfword Algebraic	Register-indirect with immediate index $EA = (rA \mid 0) + d$	rD,d(rA)
lhau	Load Halfword Algebraic with Update	Register-indirect with immediate index $EA = (rA) + d$ $rA \leftarrow EA$ $rA \neq 0, rA \neq rD$	
lhax	Load Halfword Algebraic Indexed	Register-indirect with index $EA = (rA \mid 0) + (rB)$	rD,rA,rB
lhaux	Load Halfword Algebraic with Update Indexed	Register-indirect with index $EA = (rA) + (rB)$ $rA \leftarrow EA$ $rA \neq 0, rA \neq rD$	

Store Instructions

Integer-store instructions read an operand from a GPR source register, rS , and write it into memory. Store-with-update instructions are provided, but the following two rules apply:

- rA must not be equal to 0. If $rA = 0$, the instruction form is invalid.
- If $rS = rA$, rS is written to memory first, and then the effective address is loaded into rS .

In the PPC405, the above invalid instruction form produces a boundedly-undefined result. In other PowerPC implementations, that form can cause a program exception.

Store Byte

Table 3-18 lists the PowerPC *store byte* instructions. These instructions store the lower-eight bits of rS into the specified byte location in memory.

Table 3-18: Store Byte Instructions

Mnemonic	Name	Addressing Mode	Operand Syntax
stb	Store Byte	Register-indirect with immediate index $EA = (rA \mid 0) + d$	$rS, d(rA)$
stbu	Store Byte with Update	Register-indirect with immediate index $EA = (rA) + d$ $rA \leftarrow EA$ $rA \neq 0$	
stbx	Store Byte Indexed	Register-indirect with index $EA = (rA \mid 0) + (rB)$	rS, rA, rB
stbux	Store Byte with Update Indexed	Register-indirect with index $EA = (rA) + (rB)$ $rA \leftarrow EA$ $rA \neq 0$	

Store Halfword

Table 3-19 lists the PowerPC *store halfword* instructions. These instructions store the lower-16 bits of rS into the specified halfword location in memory.

Table 3-19: Store Halfword Instructions

Mnemonic	Name	Addressing Mode	Operand Syntax
sth	Store Halfword	Register-indirect with immediate index $EA = (rA 0) + d$	$rS, d(rA)$
sthu	Store Halfword with Update	Register-indirect with immediate index $EA = (rA) + d$ $rA \leftarrow EA$ $rA \neq 0$	
sthx	Store Halfword Indexed	Register-indirect with index $EA = (rA 0) + (rB)$	rS, rA, rB
sthux	Store Halfword with Update Indexed	Register-indirect with index $EA = (rA) + (rB)$ $rA \leftarrow EA$ $rA \neq 0$	

Store Word

Table 3-20 lists the PowerPC *store word* instructions. These instructions store the entire contents of rS into the specified word location in memory.

Table 3-20: Store Word Instructions

Mnemonic	Name	Addressing Mode	Operand Syntax
stw	Store Word	Register-indirect with immediate index $EA = (rA 0) + d$	$rS, d(rA)$
stwu	Store Word with Update	Register-indirect with immediate index $EA = (rA) + d$ $rA \leftarrow EA$ $rA \neq 0$	
stwx	Store Word Indexed	Register-indirect with index $EA = (rA 0) + (rB)$	rS, rA, rB
stwux	Store Word with Update Indexed	Register-indirect with index $EA = (rA) + (rB)$ $rA \leftarrow EA$ $rA \neq 0$	

Load and Store with Byte-Reverse Instructions

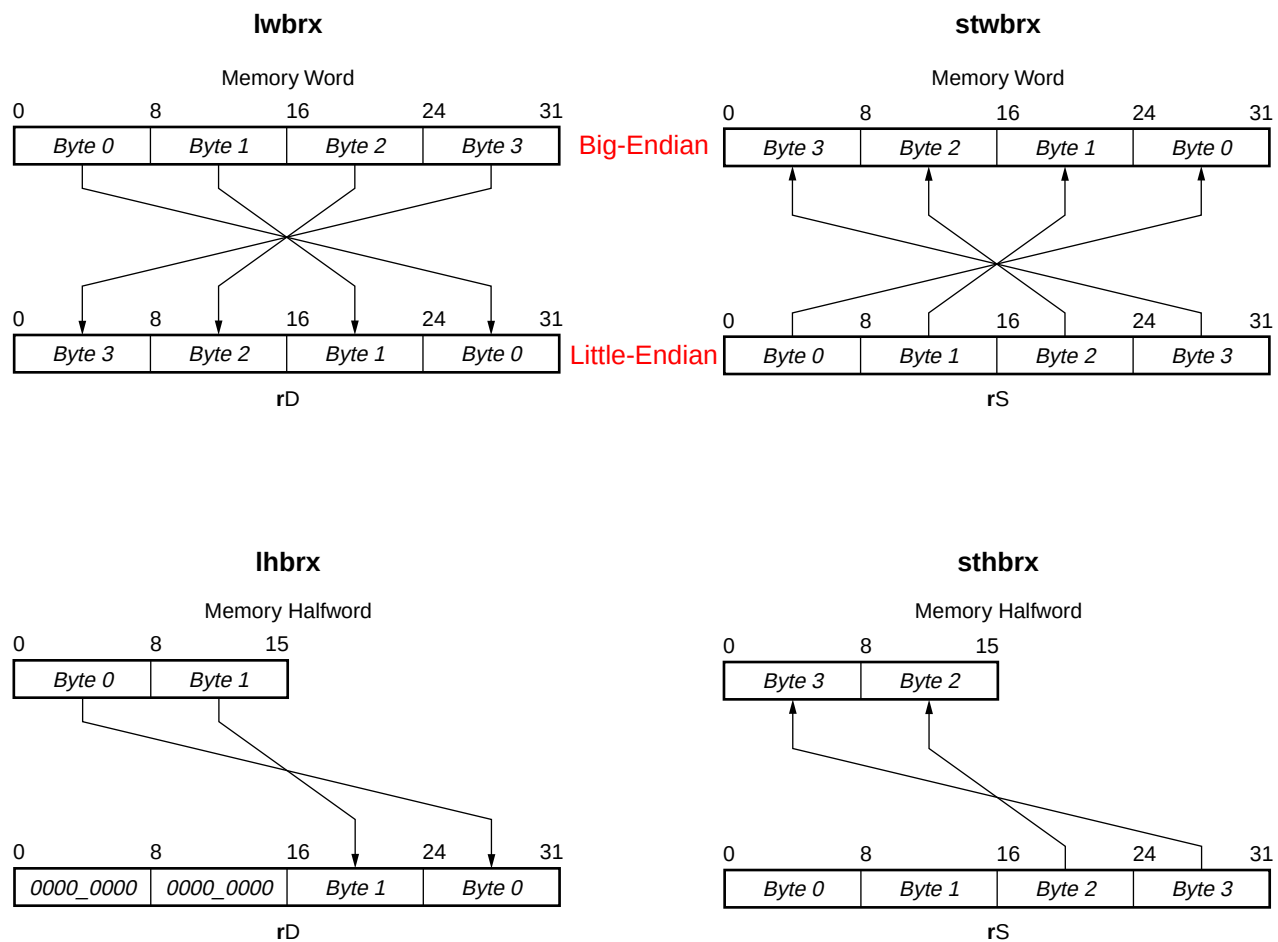
Table 3-21 lists the PowerPC *load and store with byte-reverse* instructions. **Figure 3-20** shows (using big-endian memory) how bytes are moved between memory and the GPRs for each of the byte-reverse instructions. When an **lhbrx** instruction is executed, the unloaded bytes in rD are cleared to 0.

When used in a system operating with the default big-endian byte order, these instructions have the effect of loading and storing data in little-endian order. Likewise, when used in a system operating with little-endian byte order, these instructions have the effect of loading

and storing data in big-endian order. For more information about big-endian and little-endian byte ordering, see **Byte Ordering**, page 385.

Table 3-21: Load and Store with Byte-Reverse Instructions

Mnemonic	Name	Addressing Mode	Operand Syntax
lhbrx	Load Halfword Byte-Reverse Indexed	Register-indirect with index $EA = (rA 0) + (rB)$	rD, rA, rB
lwbrx	Load Word Byte-Reverse Indexed		
sthbrx	Store Halfword Byte-Reverse Indexed	Register-indirect with index $EA = (rA 0) + (rB)$	rS, rA, rB
stwbrx	Store Word Byte-Reverse Indexed		



UG011_04_091301

Figure 3-20: Load and Store with Byte-Reverse Instructions

Load and Store Multiple Instructions

Table 3-22 lists the PowerPC *load and store multiple* instructions and their operation.

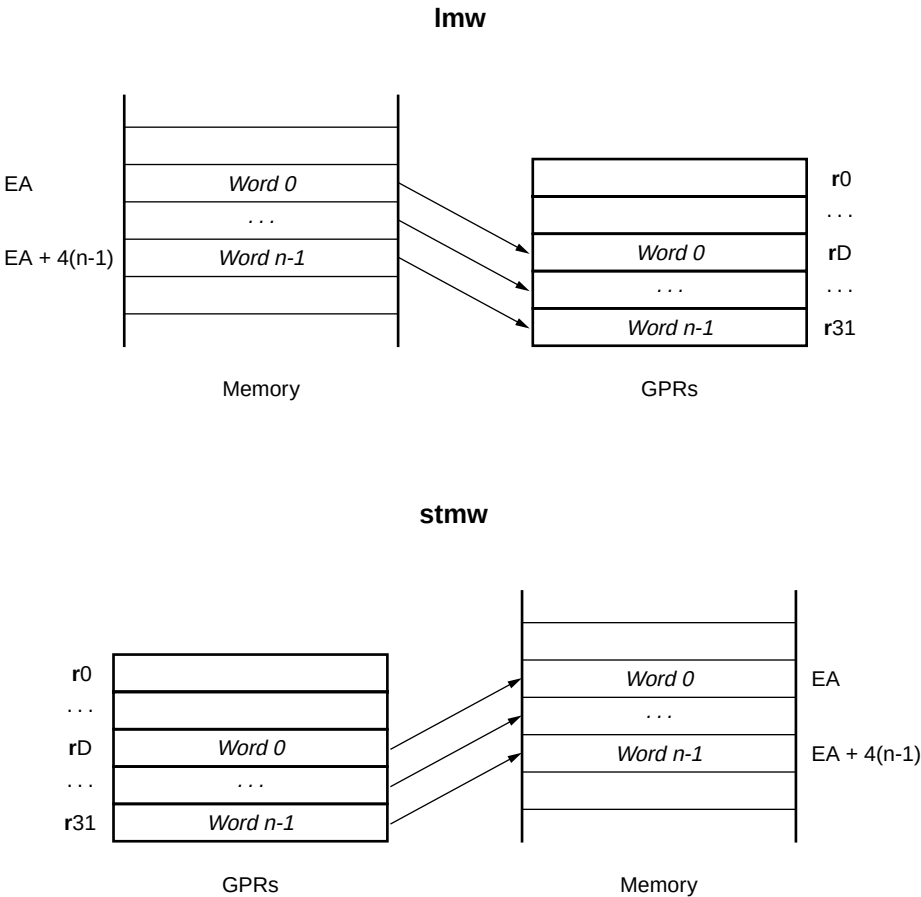
Figure 3-21 shows how bytes are moved between memory and the GPRs for each of these instructions.

These instructions are used to move blocks of data between memory and the GPRs. When the *load multiple word* instruction (**lmw**) is executed, rD through $r31$ are loaded with n

consecutive words from memory, where $n=32-rD$. For the **lmw** instruction, if **rA** is in the range of registers to be loaded, or if **rD=0**, the instruction form is invalid. When the *store multiple word* instruction (**stmw**) is executed, the n consecutive words in **rS** through **r31** are stored into memory, where $n=32-rS$.

Table 3-22: Load and Store Multiple Instructions

Mnemonic	Name	Addressing Mode	Operand Syntax
lmw	Load Multiple Word	Register-indirect with immediate index $EA = (rA \mid 0) + d$	$rD, d(rA)$
stmw	Store Multiple Word	Register-indirect with immediate index $EA = (rA \mid 0) + d$	$rS, d(rA)$



UG011_05_033101

Figure 3-21: Load and Store Multiple Instructions

Load and Store String Instructions

Table 3-23 lists the PowerPC *load and store string* instructions and their addressing modes. See the individual instruction listings in **Chapter 11, Instruction Set** for more information on their operation and restrictions on the instruction forms.

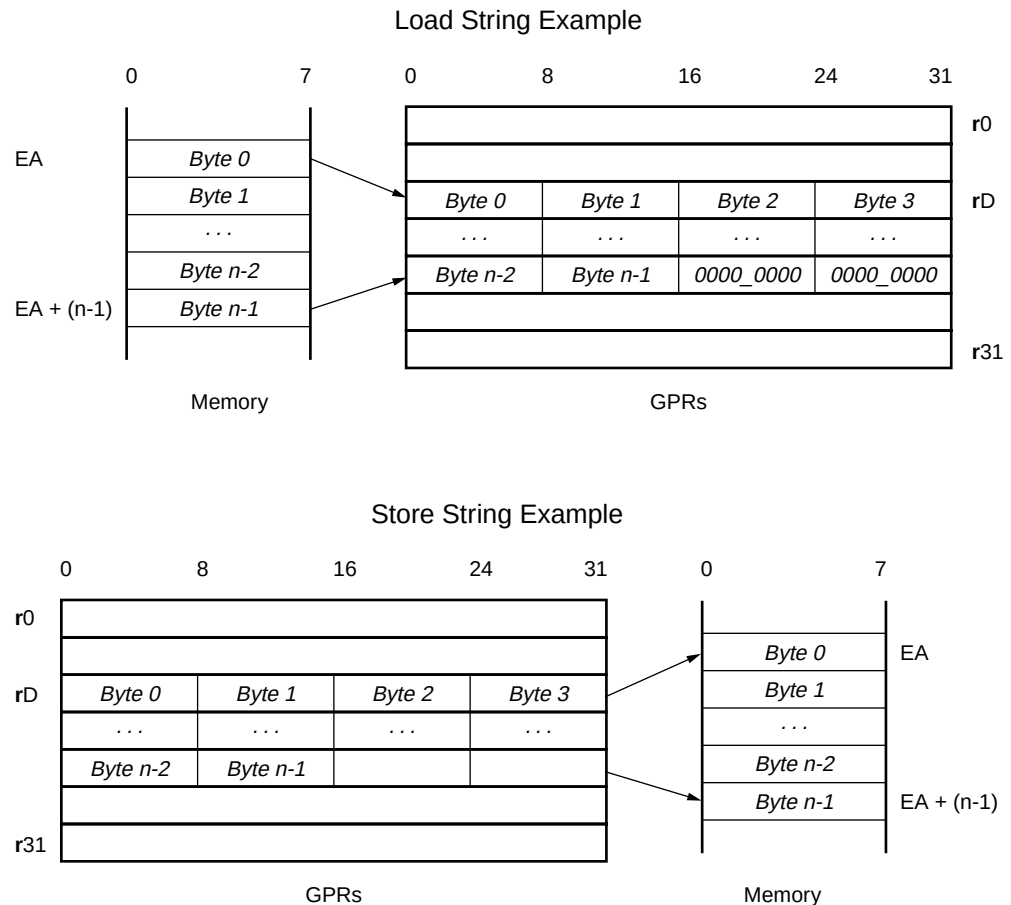
Table 3-23: Load and Store String Instructions

Mnemonic	Name	Addressing Mode	Operand Syntax
lswi	Load String Word Immediate	Register-indirect $EA = (rA \mid 0)$	rD, rA, NB
lswx	Load String Word Indexed	Register-indirect with index $EA = (rA \mid 0) + (rB)$	rD, rA, rB
stswi	Store String Word Immediate	Register-indirect $EA = (rA \mid 0)$	rS, rA, NB
stswx	Store String Word Indexed	Register-indirect with index $EA = (rA \mid 0) + (rB)$	rS, rA, rB

These instructions are used to move up to 32 consecutive bytes of data between memory and the GPRs without concern for alignment. The instructions can be used for short moves between arbitrary memory locations or for long moves between misaligned memory fields. Performance of these instructions is degraded if the leading and/or trailing bytes are not aligned on a word boundary (see **Performance Effects of Operand Alignment**, [page 389](#) for more information).

The immediate form of the instructions take the byte count, n , from the NB instruction field. If $NB=0$, then $n=32$. The indexed forms take the byte count from $XER[25:31]$. Unlike the immediate forms, if $XER[25:31]=0$, then $n=0$. For the lswx instruction, the contents of rD are undefined if $n=0$.

The n bytes are loaded into and stored from registers beginning with the most-significant register byte. For loads, any unfilled low-order register bytes are cleared to 0. The sequence of registers loaded or stored wraps through $r0$ if necessary. [Figure 3-22](#) shows an example of the string-instruction operation.



UG011_06_033101

Figure 3-22: Load and Store String Instructions

Integer Instructions

Integer instructions operate on the contents of GPRs. They use the GPRs (and sometimes immediate values coded in the instruction) as source operands. Results are written into GPRs. These instructions do not operate on memory locations. Integer instructions treat the source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation. For example, the *multiply high-word unsigned* (**mulhwu**) and *divide-word unsigned* (**divwu**) instructions interpret both operands as unsigned integers.

The following types of integer instructions are supported by the PowerPC architecture:

- Arithmetic Instructions
- Logical Instructions
- Compare Instructions
- Rotate Instructions
- Shift Instructions

The arithmetic, shift, and rotate instructions can update and/or read bits from the XER. Those instructions, plus the integer-logical instructions, can also update bits in the CR. Unless otherwise noted, when XER and/or CR are updated, they reflect the value written

to the destination register. XER and CR can be updated by the integer instructions in the following ways:

- The XER[CA] bit is updated to reflect the carry out of bit 0 in the result.
- The XER[OV] bit is set or cleared to reflect a result overflow. When XER[OV] is set, XER[SO] is also set to reflect a summary overflow. XER[SO] can only be cleared using the **mtspr** and **mcrxr** instructions. Instructions that update these bits have the overflow-enable (OE) bit set to 1 in the instruction encoding. This is indicated by the “o” suffix in the instruction mnemonic.
- Bits in CR0 (CR[0:3]) are updated to reflect a signed comparison of the result to zero. Instructions that update CR0 have the record (Rc) bit set to 1 in the instruction encoding. This is indicated by the “.” suffix in the instruction mnemonic. See **CR0 Field**, [page 397](#), for information on how these bits are updated.

Instructions that update XER[OV] or XER[CA] can delay the execution of subsequent instructions. See **Fixed-Point Exception Register (XER)**, [page 399](#) for more information on these register bits.

Arithmetic Instructions

The integer-arithmetic instructions support addition, subtraction, multiplication, and division between operands in the GPRs and in some cases between GPRs and signed-immediate values.

Integer-Addition Instructions

[Table 3-24](#) shows the PowerPC *integer-addition* instructions. The instructions in this table are grouped by the type of addition operation they perform. For each type of instruction shown, the “Operation” column indicates the addition-operation performed, and on an instruction-by-instruction basis, how the XER and CR registers are updated (if at all). “SIMM” indicates an immediate value that is sign-extended prior to being used in the operation.

The add-extended instructions can be used to perform addition on integers larger than 32 bits. For example, assume a 64-bit integer i is represented by the register pair $r3:r4$, where $r3$ contains the most-significant 32 bits of i , and $r4$ contains the least-significant 32 bits. The 64-bit integer j is similarly represented by the register pair $r5:r6$. The 64-bit result $i+j=r$ (represented by the pair $r7:r8$) is produced by pairing **adde** with **addc** as follows:

```

addc  r8,r6,r4    ! Add the least-significant words and record a
                  ! carry.
adde  r7,r5,r3    ! Add the most-significant words, using
                  ! previous carry.
```

Table 3-24: Integer-Addition Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Add Instructions</i>		rD is loaded with the sum (rA) + (rB).	
add	Add	XER and CR0 are <i>not</i> updated.	rD,rA,rB
add.	Add and Record	CR0 is updated to reflect the result.	
addo	Add with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
addo.	Add with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Table 3-24: Integer-Addition Instructions (Continued)

Mnemonic	Name	Operation	Operand Syntax
<i>Add-Carrying Instructions</i>		rD is loaded with the sum (rA) + (rB).	
addc	Add Carrying	XER[CA] is updated to reflect the result.	rD,rA,rB
addc.	Add Carrying and Record	XER[CA] and CR0 are updated to reflect the result.	
addco	Add Carrying with Overflow Enabled	XER[CA,OV,SO] are updated to reflect the result.	
addco.	Add Carrying with Overflow Enabled and Record	XER[CA,OV,SO] and CR0 are updated to reflect the result.	
<i>Add-Immediate Instructions</i>		rD is loaded with the sum (rA 0) + SIMM.	
addi	Add Immediate	XER and CR0 are <i>not</i> updated.	rD,rA,SIMM
addic	Add Immediate Carrying	XER[CA] is updated to reflect the result.	
addic.	Add Immediate Carrying and Record	XER[CA] and CR0 are updated to reflect the result.	
<i>Add Immediate-Shifted Instructions</i>		rD is loaded with the sum (rA 0) + (SIMM 0x0000).	
addis	Add Immediate Shifted	XER and CR0 are <i>not</i> updated.	rD,rA,SIMM
<i>Add-Extended Instructions</i>		rD is loaded with the sum (rA) + (rB) + XER[CA].	
adde	Add Extended	XER[CA] is updated to reflect the result.	rD,rA,rB
adde.	Add Extended and Record	XER[CA] and CR0 are updated to reflect the result.	
addeo	Add Extended with Overflow Enabled	XER[CA,OV,SO] are updated to reflect the result.	
addeo.	Add Extended with Overflow Enabled and Record	XER[CA,OV,SO] and CR0 are updated to reflect the result.	
<i>Add to Minus-One-Extended Instructions</i>		rD is loaded with the sum (rA) + XER[CA] + 0xFFFF_FFFF.	
addme	Add to Minus One Extended	XER[CA] is updated to reflect the result.	rD,rA
addme.	Add to Minus One Extended and Record	XER[CA] and CR0 are updated to reflect the result.	
addmeo	Add to Minus One Extended with Overflow Enabled	XER[CA,OV,SO] are updated to reflect the result.	
addmeo.	Add to Minus One Extended with Overflow Enabled and Record	XER[CA,OV,SO] and CR0 are updated to reflect the result.	

Table 3-24: Integer-Addition Instructions (Continued)

Mnemonic	Name	Operation	Operand Syntax
<i>Add to Zero-Extended Instructions</i>		rD is loaded with the sum (rA) + XER[CA].	
addze	Add to Zero Extended	XER[CA] is updated to reflect the result.	rD,rA
addze.	Add to Zero Extended and Record	XER[CA] and CR0 are updated to reflect the result.	
addzeo	Add to Zero Extended with Overflow Enabled	XER[CA,OV,SO] are updated to reflect the result.	
addzeo.	Add to Zero Extended with Overflow Enabled and Record	XER[CA,OV,SO] and CR0 are updated to reflect the result.	

Integer-Subtraction Instructions

Table 3-25 shows the PowerPC *integer-subtraction* instructions. The instructions in this table are grouped by the type of subtraction operation they perform. For each type of instruction shown, the “Operation” column indicates the subtraction-operation performed. The column also shows, on an instruction-by-instruction basis, how the XER and CR registers are updated (if at all). The subtraction operation is expressed as addition so that the two’s-complement operation is clear. “SIMM” indicates an immediate value that is sign-extended prior to being used in the operation.

The integer-subtraction instructions subtract the second operand (rA) from the third operand (rB). Simplified mnemonics are provided with a more familiar operand ordering, whereby the third operand is subtracted from the second. Simplified mnemonics are also defined for the addi instruction to provide a subtract-immediate operation. See **Subtract Instructions**, page 867 for more information.

The subtract-from extended instructions can be used to perform subtraction on integers larger than 32 bits. For example, assume a 64-bit integer *i* is represented by the register pair r3:r4, where r3 contains the most-significant 32 bits of *i*, and r4 contains the least-significant 32 bits. The 64-bit integer *j* is similarly represented by the register pair r5:r6. The 64-bit result $i-j=r$ (represented by the pair r7:r8) is produced by pairing **subfe** with **subfc** as follows:

```

subfc r8,r6,r4    ! Subtract the least-significant words and record a
                   ! carry.
subfe r7,r5,r3    ! Subtract the most-significant words, using
                   ! previous carry.

```

Table 3-25: Integer-Subtraction Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Subtract-From Instructions</i>		rD is loaded with the sum $\neg(rA) + (rB) + 1$.	
subf	Subtract from	XER and CR0 are <i>not</i> updated.	rD,rA,rB
subf.	Subtract from and Record	CR0 is updated to reflect the result.	
subfo	Subtract from with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
subfo.	Subtract from with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Table 3-25: Integer-Subtraction Instructions (Continued)

Mnemonic	Name	Operation	Operand Syntax
Subtract- From Carrying Instructions		rD is loaded with the sum $\neg(\text{rA}) + (\text{rB}) + 1$.	
subfc	Subtract from Carrying	XER[CA] is updated to reflect the result.	rD,rA,rB
subfc.	Subtract from Carrying and Record	XER[CA] and CR0 are updated to reflect the result.	
subfco	Subtract from Carrying with Overflow Enabled	XER[CA,OV,SO] are updated to reflect the result.	
subfco.	Subtract from Carrying with Overflow Enabled and Record	XER[CA,OV,SO] and CR0 are updated to reflect the result.	
Subtract-From Immediate Instructions		rD is loaded with the sum $\neg(\text{rA}) + \text{SIMM} + 1$.	
subfic	Subtract from Immediate Carrying	XER[CA] is updated to reflect the result.	rD,rA,SIMM
Subtract-From Extended Instructions		rD is loaded with the sum $\neg(\text{rA}) + (\text{rB}) + \text{XER[CA]}$.	
subfe	Subtract from Extended	XER[CA] is updated to reflect the result.	rD,rA,rB
subfe.	Subtract from Extended and Record	XER[CA] and CR0 are updated to reflect the result.	
subfeo	Subtract from Extended with Overflow Enabled	XER[CA,OV,SO] are updated to reflect the result.	
subfeo.	Subtract from Extended with Overflow Enabled and Record	XER[CA,OV,SO] and CR0 are updated to reflect the result.	
Subtract-From Minus-One-Extended Instructions		rD is loaded with the sum $\neg(\text{rA}) + \text{XER[CA]} + 0\text{x}\text{FFFF\_FFFF}$.	
subfme	Subtract from Minus One Extended	XER[CA] is updated to reflect the result.	rD,rA
subfme.	Subtract from Minus One Extended and Record	XER[CA] and CR0 are updated to reflect the result.	
subfmeo	Subtract from Minus One Extended with Overflow Enabled	XER[CA,OV,SO] are updated to reflect the result.	
subfmeo.	Subtract from Minus One Extended with Overflow Enabled and Record	XER[CA,OV,SO] and CR0 are updated to reflect the result.	
Subtract-From Zero-Extended Instructions		rD is loaded with the sum $\neg(\text{rA}) + \text{XER[CA]}$.	
subfze	Subtract from Zero Extended	XER[CA] is updated to reflect the result.	rD,rA
subfze.	Subtract from Zero Extended and Record	XER[CA] and CR0 are updated to reflect the result.	
subfzeo	Subtract from Zero Extended with Overflow Enabled	XER[CA,OV,SO] are updated to reflect the result.	
subfzeo.	Subtract from Zero Extended with Overflow Enabled and Record	XER[CA,OV,SO] and CR0 are updated to reflect the result.	

Negation Instructions

Table 3-26 shows the PowerPC *integer-negation* instructions. Negation takes the operand specified by rA and writes the two's-compliment equivalent in rD. For each instruction shown, the "Operation" column indicates (on an instruction-by-instruction basis) how the XER and CR registers are updated (if at all).

Table 3-26: Negation Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Negation Instructions</i>		rD is loaded with the sum $\neg(\mathbf{rA}) + 1$.	
neg	Negate	XER and CR0 are <i>not</i> updated.	rD,rA
neg.	Negate and Record	CR0 is updated to reflect the result.	
nego	Negate with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
nego.	Negate with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Multiply Instructions

Table 3-27 shows the PowerPC *integer-multiply* instructions. Multiplication of two 32-bit values can result in a 64-bit result. The multiply low-word instructions are used with the multiply high-word instructions to calculate the full 64-bit product. For each type of instruction shown, the “Operation” column indicates the multiplication-operation performed. The column also shows, on an instruction-by-instruction basis, how the XER and CR registers are updated (if at all). “SIMM” indicates an immediate value that is sign-extended prior to being used in the operation.

Table 3-27: Multiply Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply Low-Word Instructions</i>		rD is loaded with the low-32 bits of the product $(\mathbf{rA}) \times (\mathbf{rB})$.	
mullw	Multiply Low Word	XER and CR0 are <i>not</i> updated.	rD,rA,rB
mullw.	Multiply Low Word and Record	CR0 is updated to reflect the result.	
mullwo	Multiply Low Word with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
mullwo.	Multiply Low Word with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	
<i>Multiply Low-Word Immediate Instructions</i>		rD is loaded with the low-32 bits of the product $(\mathbf{rA}) \times \text{SIMM}$.	
mulli	Multiply Low Immediate	XER and CR0 are <i>not</i> updated.	rD,rA,SIMM
<i>Multiply High-Word Instructions</i>		rD is loaded with the high-32 bits of the product $(\mathbf{rA}) \times (\mathbf{rB})$.	
mulhw	Multiply High Word	XER and CR0 are <i>not</i> updated.	rD,rA,rB
mulhw.	Multiply High Word and Record	CR0 is updated to reflect the result.	
<i>Multiply High-Word Unsigned Instructions</i>		rD is loaded with the high-32 bits of the product $(\mathbf{rA}) \times (\mathbf{rB})$. The contents of rA and rB are interpreted as unsigned integers.	
mulhwu	Multiply High Word	XER and CR0 are <i>not</i> updated.	rD,rA,rB
mulhwu.	Multiply High Word and Record	CR0 is updated to reflect the result.	

Divide Instructions

Table 3-28 shows the PowerPC *integer-divide* instructions. Only the low-32 bits of the quotient are returned. The remainder is not supplied as a result of executing these instructions. For each type of instruction shown, the “Operation” column indicates the divide-operation performed. The column also shows, on an instruction-by-instruction basis, how the XER and CR registers are updated (if at all).

Table 3-28: Divide Instructions

Mnemonic	Name	Operation	Operand Syntax
Divide-Word Instructions		rD is loaded with the low-32 bits of the 64-bit quotient (rA) ÷ (rB).	
divw	Divide Word	XER and CR0 are <i>not</i> updated.	rD,rA,rB
divw.	Divide Word and Record	CR0 is updated to reflect the result.	
divwo	Divide Word with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
divwo.	Divide Word with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	
Divide-Word Unsigned Instructions		rD is loaded with the low-32 bits of the 64-bit quotient (rA) ÷ (rB). The contents of rA and rB are interpreted as unsigned integers.	
divwu	Divide Word Unsigned	XER and CR0 are <i>not</i> updated.	rD,rA,rB
divwu.	Divide Word Unsigned and Record	CR0 is updated to reflect the result.	
divwuo	Divide Word Unsigned with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
divwuo.	Divide Word Unsigned with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Logical Instructions

The logical instructions perform bit operations on the 32-bit operands. If an immediate value is specified as an operand, the processor either zero-extends or left-shifts it prior to performing the operation, depending on the instruction. If the instruction has the record (Rc) bit set to 1 in the instruction encoding, CR0 (CR[0:3]) is updated to reflect the result of the operation. A set Rc bit is indicated by the “.” suffix in the instruction mnemonic.

The logical instructions do not update any bits in the XER register.

In the operand syntax for logical instructions, the rA operand specifies a *destination* register rather than a source register. rS is used to specify one of the source registers.

AND and NAND Instructions

Table 3-29 shows the PowerPC *AND* and *NAND* instructions. For each type of instruction shown, the “Operation” column indicates the Boolean operation performed. The column also shows, on an instruction-by-instruction basis, whether the CR0 field is updated.

Table 3-29: AND and NAND Instructions

Mnemonic	Name	Operation	Operand Syntax
AND Instructions		rA is loaded with the logical result (rS) AND (rB).	
and	AND	CR0 is <i>not</i> updated.	rA,rS,rB
and.	AND and Record	CR0 is updated to reflect the result.	
AND-Immediate Instructions		rA is loaded with the logical result (rS) AND UIMM.	
andi.	AND Immediate and Record	CR0 is updated to reflect the result.	rA,rS,UIMM
AND Immediate-Shifted Instructions		rA is loaded with the logical result (rS) AND (UIMM 0x0000)	
andis.	AND Immediate Shifted and Record	CR0 is updated to reflect the result.	rA,rS,UIMM
AND with Complement Instructions		rA is loaded with the logical result (rS) AND $\neg$ (rB).	
andc	AND with Complement	CR0 is <i>not</i> updated.	rA,rS,rB
andc.	AND with Complement and Record	CR0 is updated to reflect the result.	
NAND Instructions		rA is loaded with the logical result $\neg$ ((rS) AND (rB)).	
nand	NAND	CR0 is <i>not</i> updated.	rA,rS,rB
nand.	NAND and Record	CR0 is updated to reflect the result.	

OR and NOR Instructions

Table 3-30 shows the PowerPC *OR* and *NOR* instructions. For each type of instruction shown, the “Operation” column indicates the Boolean operation performed. The column also shows, on an instruction-by-instruction basis, whether the CR0 field is updated.

Simplified mnemonics are provided for some common operations that use the OR and NOR instructions, such as move register and complement (not) register. See **Other Simplified Mnemonics**, page 870 for more information.

Table 3-30: OR and NOR Instructions

Mnemonic	Name	Operation	Operand Syntax
NOR Instructions		rA is loaded with the logical result $\neg((rS) \text{ OR } (rB))$.	
nor	NOR	CR0 is <i>not</i> updated.	rA,rS,rB
nor.	NOR and Record	CR0 is updated to reflect the result.	
OR Instructions		rA is loaded with the logical result (rS) OR (rB).	
or	OR	CR0 is <i>not</i> updated.	rA,rS,rB
or.	OR and Record	CR0 is updated to reflect the result.	
OR-Immediate Instructions		rA is loaded with the logical result (rS) OR UIMM.	
ori	OR Immediate	CR0 is <i>not</i> updated.	rA,rS,UIMM

Table 3-30: OR and NOR Instructions (*Continued*)

Mnemonic	Name	Operation	Operand Syntax
<i>OR Immediate-Shifted Instructions</i>		rA is loaded with the logical result (rS) OR (UIMM 0x0000)	
oris	OR Immediate Shifted	CR0 is <i>not</i> updated.	rA,rS,UIMM
<i>OR with Complement Instructions</i>		rA is loaded with the logical result (rS) OR $\neg$ (rB).	
orc	OR with Complement	CR0 is <i>not</i> updated.	rA,rS,rB
orc.	OR with Complement and Record	CR0 is updated to reflect the result.	

XOR and Equivalence Instructions

Table 3-31 shows the PowerPC *XOR and equivalence* (XNOR) instructions. For each type of instruction shown, the “Operation” column indicates the Boolean operation performed. The column also shows, on an instruction-by-instruction basis, whether the CR0 field is updated.

Table 3-31: XOR and Equivalence Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Equivalence Instructions</i>		rA is loaded with the logical result $\neg((rS) \text{ XOR } (rB))$.	
eqv	Equivalent	CR0 is <i>not</i> updated.	rA,rS,rB
eqv.	Equivalent and Record	CR0 is updated to reflect the result.	
<i>XOR Instructions</i>		rA is loaded with the logical result (rS) XOR (rB).	
xor	XOR	CR0 is <i>not</i> updated.	rA,rS,rB
xor.	XOR and Record	CR0 is updated to reflect the result.	
<i>XOR-Immediate Instructions</i>		rA is loaded with the logical result (rS) XOR UIMM.	
xori	XOR Immediate	CR0 is <i>not</i> updated.	rA,rS,UIMM
<i>XOR Immediate-Shifted Instructions</i>		rA is loaded with the logical result (rS) XOR (UIMM 0x0000)	
xoris	XOR Immediate Shifted	CR0 is <i>not</i> updated.	rA,rS,UIMM

Sign-Extension Instructions

Table 3-32 shows the *sign-extension* instructions. These instructions sign-extend the value in the rS register and write the result in the rA register. For each type of instruction shown, the “Operation” column indicates the operation performed. The column also shows, on an instruction-by-instruction basis, whether the CR0 field is updated.

Table 3-32: Sign-Extension Instructions

Mnemonic	Name	Operation	Operand Syntax
Extend-Sign Byte Instructions		rA[24:31] is loaded with (rS[24:31]). The remaining bits rA[0:23] are each loaded with a copy of (rS[24]).	
extsb	Extend Sign Byte	CR0 is <i>not</i> updated.	rA,rS
extsb.	Extend Sign Byte and Record	CR0 is updated to reflect the result.	
Extend-Sign Halfword Instructions		rA[16:31] is loaded with (rS[16:31]). The remaining bits rA[0:15] are each loaded with a copy of (rS[16]).	
extsh	Extend Sign Halfword	CR0 is <i>not</i> updated.	rA,rS
extsh.	Extend Sign Halfword and Record	CR0 is updated to reflect the result.	

Count Leading-Zeros Instructions

Table 3-33 shows the *count leading-zeros* instructions. These instructions count the number of consecutive zero bits in the **rS** register starting at bit 0. The count result is written to the **rA** register. For each type of instruction shown, the “Operation” column indicates the operation performed. The column also shows, on an instruction-by-instruction basis, whether the CR0 field is updated.

Table 3-33: Count Leading-Zeros Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Count Leading-Zeros Instructions</i>		rA is loaded with a count of leading zeros in rS .	
cntlzw	Count Leading Zeros Word	CR0 is <i>not</i> updated.	rA,rS
cntlzw.	Count Leading Zeros Word and Record	CR0 is updated to reflect the result. CR0[LT] is always cleared to 0.	

Compare Instructions

The integer-compare instructions support algebraic and logical comparisons between operands in the GPRs and between GPRs and immediate values. Immediate values are signed in algebraic comparisons and unsigned in logical comparisons.

All compare instructions have four operands. The first operand, **crfD**, specifies the field in the CR register that is updated with the comparison result. The left-most three bits in the CR field are updated to reflect a less-than, greater-than, or equal comparison. The fourth (least-significant) bit is updated with a copy of XER[SO]. The **crfD** operand can be omitted if the comparison results are written to CR0. See **CRn Fields (Compare Instructions)**, [page 398](#) for more information on the CR fields.

The second operand specifies the operand length. This is referred to the “L” bit in the compare-instruction encoding. When using the compare instructions on 32-bit PowerPC implementations like the PPC405, this bit *must* always be coded as 0. It cannot be omitted from the standard instruction syntax. Simplified mnemonics are provided that omit this operand. See **Compare Instructions**, [page 864](#) for more information.

The last two operands specify the quantities to be compared (the contents of a register and a register or immediate value).

Algebraic-Comparison Instructions

Table 3-34 shows the PowerPC *algebraic-comparison* instructions. During comparison, both operands are treated as signed integers. If a comparison is made with a signed-immediate value (SIMM), that value is sign-extended by the processor prior to performing the comparison.

Table 3-34: Algebraic-Comparison Instructions

Mnemonic	Name	Operation	Operand Syntax
cmp	Compare	crfD [LT,GT,EQ] are loaded with the result of algebraically comparing (rA) with (rB). CR [SO] is loaded with a copy of XER [SO].	crfD ,0, rA , rB
cmpi	Compare Immediate	crfD [LT,GT,EQ] are loaded with the result of algebraically comparing (rA) with SIMM . CR [SO] is loaded with a copy of XER [SO].	crfD ,0, rA , SIMM

Logical-Comparison Instructions

Table 3-35 shows the PowerPC *logical-comparison* instructions. During comparison, both operands are treated as unsigned integers. If a comparison is made with an unsigned-immediate value (UIMM), that value is zero extended by the processor prior to performing the comparison.

Table 3-35: Logical-Comparison Instructions

Mnemonic	Name	Operation	Operand Syntax
cmpl	Compare Logical	crfD [LT,GT,EQ] are loaded with the result of logically comparing (rA) with (rB). CR [SO] is loaded with a copy of XER [SO].	crfD ,0, rA , rB
cmpli	Compare Logical Immediate	crfD [LT,GT,EQ] are loaded with the result of logically comparing (rA) with UIMM . CR [SO] is loaded with a copy of XER [SO].	crfD ,0, rA , UIMM

Rotate Instructions

Rotate instructions operate on 32-bit data in the GPRs, returning the result in a second GPR. These instructions rotate data to the left—the direction of least-significant bit to most-significant bit. Bits rotated out of the most-significant bit (bit 0) are rotated into the least-significant bit (bit 31). Programmers can achieve apparent right rotation using these left-rotation instructions by specifying a rotation amount of $32-n$, where n is the number of bits to rotate right.

If the rotate instruction has the record (**Rc**) bit set to 1 in the instruction encoding, **CR0** (**CR**[0:3]) is updated to reflect the result of the operation. A set **Rc** bit is indicated by the “.” suffix in the instruction mnemonic. Rotate instructions do not update any bits in the **XER** register.

In the operand syntax for rotate instructions, the **rA** operand specifies the *destination* register rather than a source register. **rS** is used to specify the source register.

Simplified mnemonics using the rotate instructions are provided for easy coding of extraction, insertion, left or right justification, and other bit-manipulation operations. See **Rotate and Shift Instructions**, [page 865](#) for more information.

Mask Generation

The rotate instructions write their results into the destination register under the control of a mask specified in the rotate-instruction encoding. The mask is used to write or insert a partial result into the destination register.

Rotate masks are 32-bits long. Two instruction-opcode fields are used to specify the mask: MB and ME. MB is a 5-bit field specifying the starting bit position of the mask and ME is a 5-bit field specifying the ending bit position of the mask. The mask consists of all 1's from MB to ME *inclusive* and all 0's elsewhere. If MB > ME, the string of 1's wraps around from bit 31 to bit 0. In this case, 0's are found from ME to MB *exclusive*. The generation of an all-zero mask is not possible.

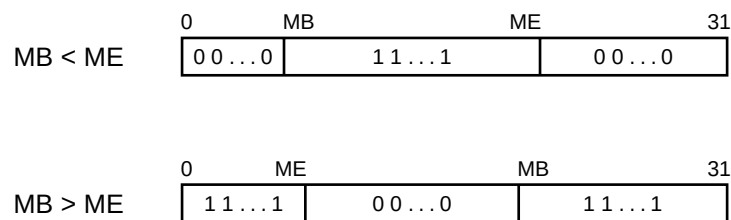
The function of the MASK(MB,ME) generator is summarized as:

```

if MB < ME then
    mask[MB:ME] = 1's
    mask[all remaining bits] = 0's
else
    mask[MB:31] = ones
    mask[0:ME] = ones
    mask[all remaining bits] = 0's

```

Figure 3-23 shows the generated mask for both cases.



UG011_15_033101

Figure 3-23: Rotate Mask Generation

Rotate Left then AND-with-Mask Instructions

Table 3-36 shows the PowerPC *rotate left then AND-with-mask* instructions. For each type of instruction shown, the “Operation” column indicates the rotate operation performed. The column also shows, on an instruction-by-instruction basis, whether the CR0 field is updated.

Table 3-36: Rotate Left then AND-with-Mask Instructions

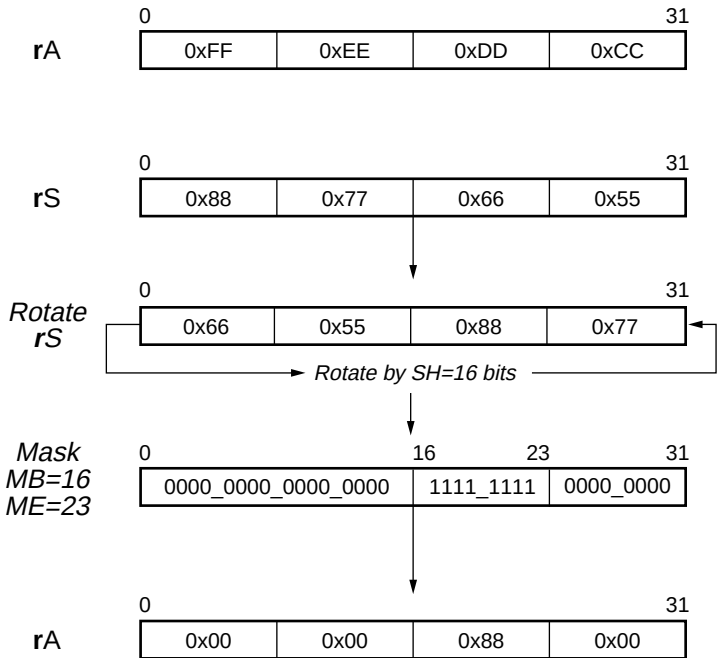
Mnemonic	Name	Operation	Operand Syntax
<i>Rotate Left then AND-with-Mask Immediate Instructions</i>		rA is loaded with the masked result of left-rotating (rS) the number of bits specified by SH. The mask is specified by operands MB and ME.	
rlwinm	Rotate Left Word Immediate then AND with Mask	CR0 is <i>not</i> updated.	rA,rS,SH,MB,ME
rlwinm.	Rotate Left Word Immediate then AND with Mask and Record	CR0 is updated to reflect the result.	

Table 3-36: Rotate Left then AND-with-Mask Instructions (Continued)

Mnemonic	Name	Operation	Operand Syntax
<i>Rotate Left then AND-with-Mask Instructions</i>		<i>rA</i> is loaded with the masked result of left-rotating (<i>rS</i>) the number of bits specified by (<i>rB</i>). The mask is specified by operands <i>MB</i> and <i>ME</i> .	
rlwnm	Rotate Left Word then AND with Mask	CR0 is <i>not</i> updated.	<i>rA</i> , <i>rS</i> , <i>rB</i> , <i>MB</i> , <i>ME</i>
rlwnm.	Rotate Left Word then AND with Mask and Record	CR0 is updated to reflect the result.	

These instructions left rotate GPR contents and logically AND the result with the mask prior to writing it into the destination GPR. The destination register contains the rotated result in the unmasked bit positions (mask bits with 1's), and 0's in the masked bit positions (mask bits with 0's). Rotation amounts are specified using an immediate field in the instruction (the SH opcode field) or using a value in a register.

Figure 3-24 shows an example of a rotate left then AND-with-mask immediate instruction. In this example, the rotation amount is 16 bits as specified by the SH field in the instruction. The mask specifies an unmasked byte in bit positions 16:23 (MB=16, ME=23) and masks all other bit positions. The example shows the original contents of the destination register, *rA*, and the source register, *rS*. *rS* is left-rotated 16 bits and the result is written to *rA* after ANDing with the mask. This has the effect of extracting byte 0 from *rS* (*rS*[0:7]) and placing it in byte 2 of *rA* (*rA*[16:23]).



UG011_16_033101

Figure 3-24: Rotate Left then AND-with-Mask Immediate Example

Rotate Left then Mask-Insert Instructions

Table 3-36 shows the PowerPC *rotate left then mask-insert* instructions. For each type of instruction shown, the “Operation” column indicates the rotate operation performed. The

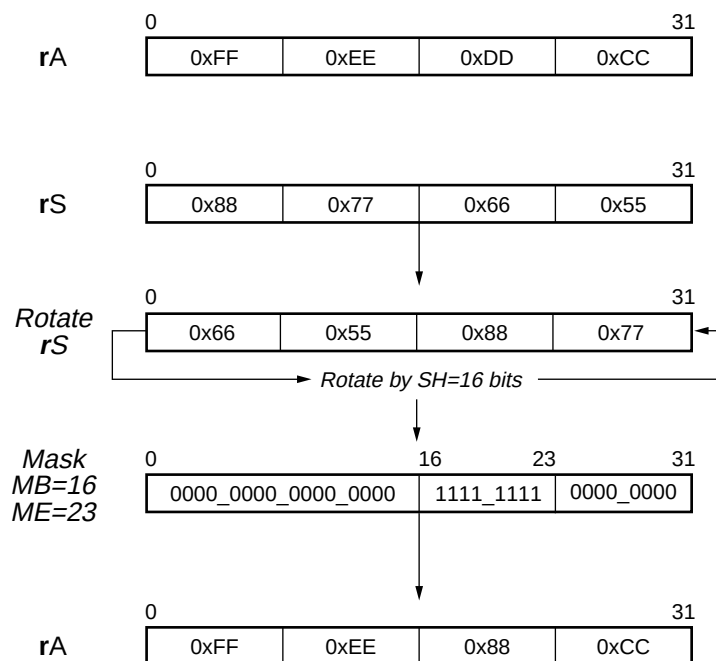
column also shows, on an instruction-by-instruction basis, whether the CR0 field is updated.

Table 3-37: Rotate Left then Mask-Insert Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Rotate Left then Mask-Insert Immediate Instructions</i>		The masked result of left-rotating (rS) the number of bits specified by SH is inserted into rA. The mask is specified by operands MB and ME.	
rlwimi	Rotate Left Word Immediate then Mask Insert	CR0 is <i>not</i> updated.	rA,rS,SH,MB,ME
rlwimi.	Rotate Left Word Immediate then Mask Insert and Record	CR0 is updated to reflect the result.	

These instructions left rotate GPR contents and insert the results into the destination GPR under control of the mask. The destination register contains the rotated result in the unmasked bit positions (mask bits with 1's) and the original contents of the destination register in the masked bit positions (mask bits with 0's). Rotation amounts are specified using an immediate field in the instruction (the SH opcode field).

Figure 3-25 shows an example of a rotate left then mask-insert immediate instruction. In this example, the rotation amount is 16 bits as specified by the SH field in the instruction. The mask specifies an unmasked byte in bit positions 16:23 (MB=16, ME=23) and masks all other bit positions. The example shows the original contents of the destination register, rA, and the source register, rS. rS is rotated 16 bits and the result is inserted into rA after ANDing with the mask. This has the effect of extracting byte 0 from rS (rS[0:7]) and inserting it into byte 2 of rA (rA[16:23]), leaving all remaining bytes in rA unmodified.



UG011_17_033101

Figure 3-25: Rotate Left then Mask-Insert Immediate Example

Shift Instructions

Shift instructions operate on 32-bit data in the GPRs and return the result in a GPR. Both logical and algebraic shifts are provided:

- *Logical left-shift* instructions shift bits from the direction of least-significant bit to most-significant bit. Bits shifted out of bit 0 are lost. The vacated bit positions on the right are filled with zeros.
- *Logical right-shift* instructions shift bits from the direction of most-significant bit to least-significant bit. Bits shifted out of bit 31 are lost. The vacated bit positions on the left are filled with zeros.
- *Algebraic right-shift* instructions shift bits from the direction of most-significant bit to least-significant bit. Bits shifted out of bit 31 are lost. The vacated bit positions on the left are filled with a copy of the original bit 0 (the value prior to starting the shift).

If the shift instruction has the record (Rc) bit set to 1 in the instruction encoding, CR0 (CR[0:3]) is updated to reflect the result of the operation. A set Rc bit is indicated by the “.” suffix in the instruction mnemonic. Algebraic right-shift instructions update XER[CA] to reflect the result of the operation but the other shift instructions do not modify XER[CA]. XER[OV,SO] are not modified by any shift instructions.

In the operand syntax for shift instructions, the rA operand specifies the *destination* register rather than a source register. rS is used to specify the source register.

Simplified mnemonics using the rotate instructions are provided for coding of logical shift-left immediate and logical shift-right immediate operations. See **Rotate and Shift Instructions**, [page 865](#) for more information.

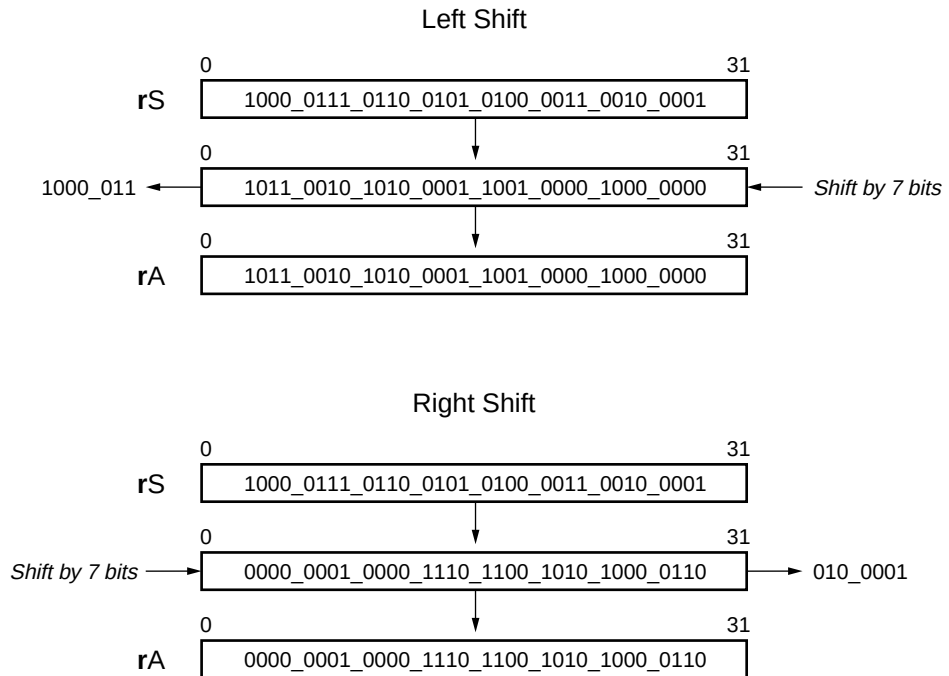
Logical-Shift Instructions

[Table 3-38](#) shows the PowerPC *logical-shift* instructions. For each type of instruction shown, the “Operation” column indicates the shift operation performed. The column also shows, on an instruction-by-instruction basis, whether the CR0 field is updated. XER is not updated by these instructions.

Table 3-38: Logical-Shift Instructions

Mnemonic	Name	Operation	Operand Syntax
Shift-Left-Logical Instructions		rA is loaded with the result of logically left-shifting (rS) the number of bits specified by (rB).	
slw	Shift Left Word	CR0 is <i>not</i> updated.	rA,rS,rB
slw.	Shift Left Word and Record	CR0 is updated to reflect the result.	
Shift-Right-Logical Instructions		rA is loaded with the result of logically right-shifting (rS) the number of bits specified by (rB).	
srw	Shift Right Word	CR0 is <i>not</i> updated.	rA,rS,rB
srw.	Shift Right Word and Record	CR0 is updated to reflect the result.	

[Figure 3-26](#) shows two examples of logical-shift operations. The top example shows a left shift of seven bits, and the bottom example shows a right shift of seven bits. As is seen in these examples, bits shifted out of the register are lost and vacated bits are filled with zeros.



UG011_18_033101

Figure 3-26: Logical-Shift Examples

Algebraic-Shift Instructions

Table 3-39 shows the PowerPC *algebraic-shift* instructions. For each type of instruction shown, the “Operation” column indicates the shift operation performed. The column also shows, on an instruction-by-instruction basis, whether the CR0 field is updated. XER[CA] is always updated by these instructions to reflect the result.

The shift-right-algebraic instructions can be followed by an **addze** instruction to implement a divide-by- 2^n operation. See **Multiple-Precision Shifts**, page 876, for more information.

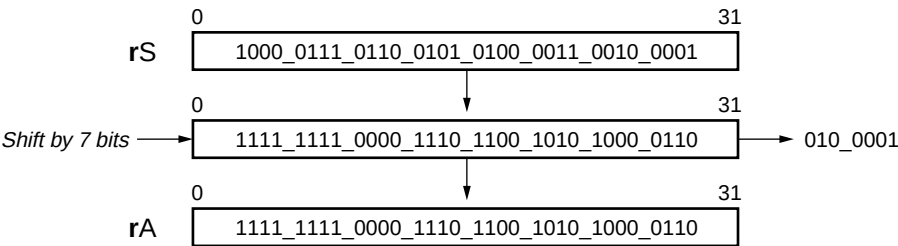
Table 3-39: Algebraic-Shift Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Shift-Right-Algebraic Immediate Instructions</i>		rA is loaded with the result of algebraically right-shifting (rS) the number of bits specified by SH.	
srawi	Shift Right Algebraic Word Immediate	CR0 is <i>not</i> updated. XER[CA] is updated to reflect the result.	rA,rS,SH
srawi.	Shift Right Algebraic Word Immediate and Record	CR0 and XER[CA] are updated to reflect the result.	

Table 3-39: Algebraic-Shift Instructions (Continued)

Mnemonic	Name	Operation	Operand Syntax
<i>Shift-Right-Algebraic Instructions</i>		rA is loaded with the result of algebraically right-shifting (rS) the number of bits specified by (rB).	
sraw	Shift Right Algebraic Word	CR0 is <i>not</i> updated. XER[CA] is updated to reflect the result.	rA,rS,rB
sraw.	Shift Right Algebraic Word and Record	CR0 and XER[CA] are updated to reflect the result.	

Figure 3-27 shows an example of an algebraic-shift operation. In this example, a shift of seven bits is performed. Bits shifted out of the least-significant register bit are lost and vacated bits on the left side are filled with a copy of the original bit 0 (prior to the shift). In this example, the original value of bit 0 is 0b1.



UG011_19_033101

Figure 3-27: Algebraic-Shift Example

Multiply-Accumulate Instruction-Set Extensions

The PPC405 supports an *integer multiply-accumulate* instruction-set extension that provides functions usable by certain computationally intensive applications, such as those that implement DSP algorithms. These instructions comply with the architectural requirements for auxiliary-processor units (APUs) defined by the PowerPC embedded-environment architecture and the PowerPC Book-E architecture. They are considered implementation-dependent instructions and are not part of the PowerPC architecture, the PowerPC embedded-environment architecture, or the PowerPC Book-E architecture. Programs that use these instructions are not portable to all PowerPC implementations.

The multiply-accumulate instruction-set extensions include multiply-accumulate instructions, negative multiply-accumulate instructions, and multiply-halfword instructions.

Modulo and Saturating Arithmetic

The multiply-accumulate and negative multiply-accumulate instructions produce a 33-bit intermediate result. The method used to store this result in the 32-bit destination register depends on whether the instruction performs *modulo arithmetic* or *saturating arithmetic*.

With modulo-arithmetic instructions, the most-significant bit in the intermediate result is discarded and the low-32 bits of this result are stored in the destination register.

With saturating-arithmetic instructions, the low 32-bits of the intermediate result are stored in the destination register if the intermediate result does not overflow 32-bits. However, if the intermediate result overflows what is representable in 32-bits, the

instruction loads the nearest representable value into the destination register. For the various instruction forms, these results are:

- Signed arithmetic—if the result exceeds $2^{31}-1$ ($> 0x7FFF\_FFFF$), the instruction loads the destination register with $2^{31}-1$.
- Signed arithmetic—if the result is less than -2^{31} ($< 0x8000\_0000$), the instruction loads the destination register with -2^{31} .
- Unsigned arithmetic—if the result exceeds $2^{32}-1$ ($> 0xFFFF\_FFFF$), the instruction loads the destination register with $2^{32}-1$.

Multiply-Accumulate Instructions

Multiply-Accumulate Cross-Halfword to Word Instructions

Table 3-40 shows the PPC405 *integer multiply-accumulate cross-halfword to word* instructions. These instructions take the lower halfword of the first source operand ($rA[16:31]$) and multiply it with the upper halfword of the second source operand ($rB[0:15]$), producing a 32-bit product. The product is signed or unsigned, depending on the instruction. This product is added to the value in the destination register, rD , producing a 33-bit intermediate result. Generally, rD is loaded with the lower-32 bits of the 33-bit intermediate result. However, if the instruction performs saturating arithmetic and the intermediate result overflows, rD is loaded with the nearest representable value (see **Modulo and Saturating Arithmetic**, above).

For each type of instruction shown in Table 3-40, the “Operation” column indicates the multiply-accumulate operation performed. The column also shows, on an instruction-by-instruction basis, how the XER and CR registers are updated (if at all).

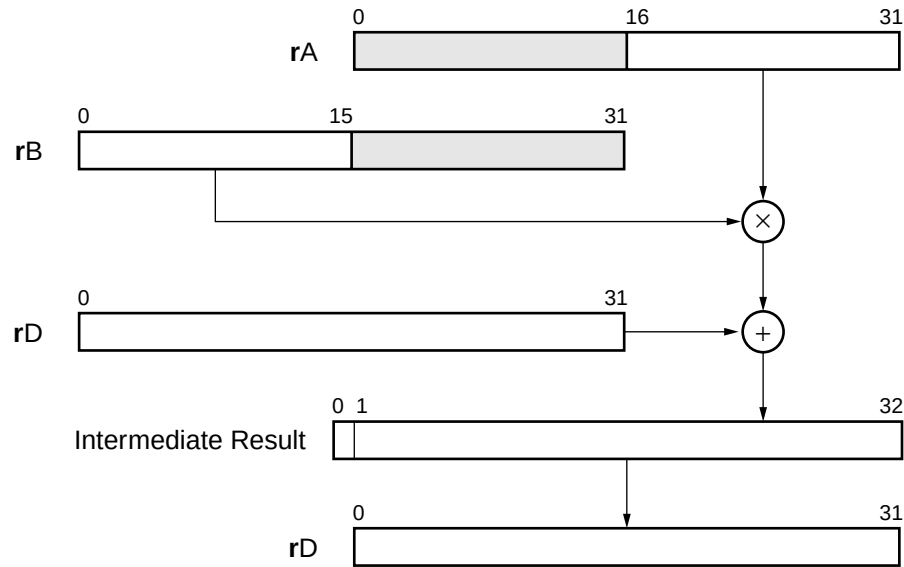
Table 3-40: Multiply-Accumulate Cross-Halfword to Word Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply-Accumulate Cross-Halfword to Word Modulo Signed Instructions</i>		rD is added to the signed product $(rA[16:31]) \times (rB[0:15])$, producing a 33-bit result. The low-32 bits of this result are stored in rD .	
macchw	Multiply Accumulate Cross Halfword to Word Modulo Signed	XER and CR0 are <i>not</i> updated.	rD, rA, rB
macchw.	Multiply Accumulate Cross Halfword to Word Modulo Signed and Record	CR0 is updated to reflect the result.	
macchwo	Multiply Accumulate Cross Halfword to Word Modulo Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
macchwo.	Multiply Accumulate Cross Halfword to Word Modulo Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Table 3-40: Multiply-Accumulate Cross-Halfword to Word Instructions (Continued)

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply-Accumulate Cross-Halfword to Word Saturate Signed Instructions</i>		rD is added to the signed product $(rA[16:31]) \times (rB[0:15])$, producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in rD . Otherwise, the nearest-representable value is stored in rD .	
macchws	Multiply Accumulate Cross Halfword to Word Saturate Signed	XER and CR0 are <i>not</i> updated.	rD, rA, rB
macchws.	Multiply Accumulate Cross Halfword to Word Saturate Signed and Record	CR0 is updated to reflect the result.	
macchwso	Multiply Accumulate Cross Halfword to Word Saturate Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
macchwso.	Multiply Accumulate Cross Halfword to Word Saturate Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	
<i>Multiply-Accumulate Cross-Halfword to Word Saturate Unsigned Instructions</i>		rD is added to the unsigned product $(rA[16:31]) \times (rB[0:15])$, producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in rD . Otherwise, the nearest-representable value is stored in rD .	
macchwsu	Multiply Accumulate Cross Halfword to Word Saturate Unsigned	XER and CR0 are <i>not</i> updated.	rD, rA, rB
macchwsu.	Multiply Accumulate Cross Halfword to Word Saturate Unsigned and Record	CR0 is updated to reflect the result.	
macchwsuo	Multiply Accumulate Cross Halfword to Word Saturate Unsigned with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
macchwsuo.	Multiply Accumulate Cross Halfword to Word Saturate Unsigned with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	
<i>Multiply-Accumulate Cross-Halfword to Word Modulo Unsigned Instructions</i>		rD is added to the unsigned product $(rA[16:31]) \times (rB[0:15])$, producing a 33-bit result. The low-32 bits of this result are stored in rD .	
macchwu	Multiply Accumulate Cross Halfword to Word Modulo Unsigned	XER and CR0 are <i>not</i> updated.	rD, rA, rB
macchwu.	Multiply Accumulate Cross Halfword to Word Modulo Unsigned and Record	CR0 is updated to reflect the result.	
macchwuo	Multiply Accumulate Cross Halfword to Word Modulo Unsigned with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
macchwuo.	Multiply Accumulate Cross Halfword to Word Modulo Unsigned with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Figure 3-28 shows the operation of the integer multiply-accumulate cross-halfword to word instructions.



UG011_20_033101

Figure 3-28: Multiply-Accumulate Cross-Halfword to Word Operation

Multiply-Accumulate High-Halfword to Word Instructions

Table 3-41 shows the PPC405 *multiply-accumulate high-halfword to word* instructions. These instructions multiply the high halfword of both source operands, $rA[0:15]$ and $rB[0:15]$, producing a 32-bit product. The product is signed or unsigned, depending on the instruction. This product is added to the value in the destination register, rD , producing a 33-bit intermediate result. Generally, rD is loaded with the lower-32 bits of the 33-bit intermediate result. However, if the instruction performs saturating arithmetic and the intermediate result overflows, rD is loaded with the nearest representable value (see **Modulo and Saturating Arithmetic**, page 441).

For each type of instruction shown in Table 3-41, the “Operation” column indicates the multiply-accumulate operation performed. The column also shows, on an instruction-by-instruction basis, how the XER and CR registers are updated (if at all).

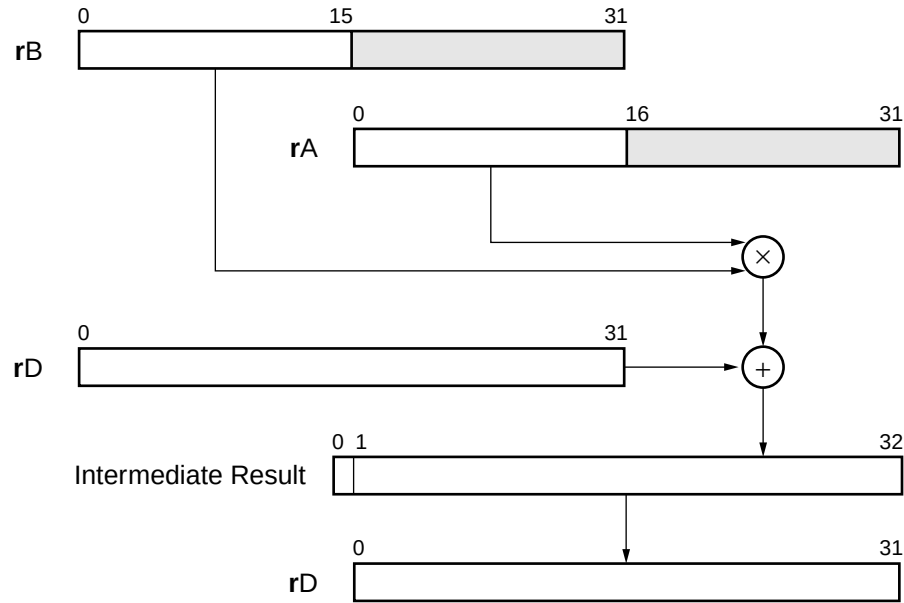
Table 3-41: Multiply-Accumulate High-Halfword to Word Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply-Accumulate High-Halfword to Word Modulo Signed Instructions</i>		rD is added to the signed product $(rA[0:15]) \times (rB[0:15])$, producing a 33-bit result. The low-32 bits of this result are stored in rD .	
machhw	Multiply Accumulate High Halfword to Word Modulo Signed	XER and CR0 are <i>not</i> updated.	rD, rA, rB
machhw.	Multiply Accumulate High Halfword to Word Modulo Signed and Record	CR0 is updated to reflect the result.	
machhwo	Multiply Accumulate High Halfword to Word Modulo Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
machhwo.	Multiply Accumulate High Halfword to Word Modulo Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Table 3-41: Multiply-Accumulate High-Halfword to Word Instructions (*Continued*)

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply-Accumulate High-Halfword to Word Saturate Signed Instructions</i>		rD is added to the signed product $(rA[0:15]) \times (rB[0:15])$, producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in rD . Otherwise, the nearest-representable value is stored in rD .	
machhws	Multiply Accumulate High Halfword to Word Saturate Signed	XER and CR0 are <i>not</i> updated.	rD, rA, rB
machhws.	Multiply Accumulate High Halfword to Word Saturate Signed and Record	CR0 is updated to reflect the result.	
machhwsO	Multiply Accumulate High Halfword to Word Saturate Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
machhwsO.	Multiply Accumulate High Halfword to Word Saturate Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	
<i>Multiply-Accumulate High-Halfword to Word Saturate Unsigned Instructions</i>		rD is added to the unsigned product $(rA[0:15]) \times (rB[0:15])$, producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in rD . Otherwise, the nearest-representable value is stored in rD .	
machhwsu	Multiply Accumulate High Halfword to Word Saturate Unsigned	XER and CR0 are <i>not</i> updated.	rD, rA, rB
machhwsu.	Multiply Accumulate High Halfword to Word Saturate Unsigned and Record	CR0 is updated to reflect the result.	
machhwsuo	Multiply Accumulate High Halfword to Word Saturate Unsigned with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
machhwsuo.	Multiply Accumulate High Halfword to Word Saturate Unsigned with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	
<i>Multiply-Accumulate High-Halfword to Word Modulo Unsigned Instructions</i>		rD is added to the unsigned product $(rA[0:15]) \times (rB[0:15])$, producing a 33-bit result. The low-32 bits of this result are stored in rD .	
machhwu	Multiply Accumulate High Halfword to Word Modulo Unsigned	XER and CR0 are <i>not</i> updated.	rD, rA, rB
machhwu.	Multiply Accumulate High Halfword to Word Modulo Unsigned and Record	CR0 is updated to reflect the result.	
machhwuo	Multiply Accumulate High Halfword to Word Modulo Unsigned with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
machhwuo.	Multiply Accumulate High Halfword to Word Modulo Unsigned with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Figure 3-29 shows the operation of the multiply-accumulate high-halfword to word instructions.



UG011_21_033101

Figure 3-29: Multiply-Accumulate High-Halfword to Word Operation

Multiply-Accumulate Low-Halfword to Word Instructions

Table 3-42 shows the PPC405 *multiply-accumulate low-halfword to word* instructions. These instructions multiply the low halfword of both source operands, rA[16:31] and rB[16:31], producing a 32-bit product. The product is signed or unsigned, depending on the instruction. This product is added to the value in the destination register, rD, producing a 33-bit intermediate result. Generally, rD is loaded with the lower-32 bits of the 33-bit intermediate result. However, if the instruction performs saturating arithmetic and the intermediate result overflows, rD is loaded with the nearest representable value (see **Modulo and Saturating Arithmetic**, page 441).

For each type of instruction shown in Table 3-42, the “Operation” column indicates the multiply-accumulate operation performed. The column also shows, on an instruction-by-instruction basis, how the XER and CR registers are updated (if at all).

Table 3-42: Multiply-Accumulate Low-Halfword to Word Instructions

Mnemonic	Name	Operation	Operand Syntax
Multiply-Accumulate Low-Halfword to Word Modulo Signed Instructions		rD is added to the signed product (rA[16:31]) × (rB[16:31]), producing a 33-bit result. The low-32 bits of this result are stored in rD.	
maclhw	Multiply Accumulate Low Halfword to Word Modulo Signed	XER and CR0 are <i>not</i> updated.	rD,rA,rB
maclhw.	Multiply Accumulate Low Halfword to Word Modulo Signed and Record	CR0 is updated to reflect the result.	
maclhwo	Multiply Accumulate Low Halfword to Word Modulo Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
maclhwo.	Multiply Accumulate Low Halfword to Word Modulo Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	
Multiply-Accumulate Low-Halfword to Word Saturate Signed Instructions		rD is added to the signed product (rA[16:31]) × (rB[16:31]), producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in rD. Otherwise, the nearest-representable value is stored in rD.	
maclhws	Multiply Accumulate Low Halfword to Word Saturate Signed	XER and CR0 are <i>not</i> updated.	rD,rA,rB
maclhws.	Multiply Accumulate Low Halfword to Word Saturate Signed and Record	CR0 is updated to reflect the result.	
maclhwso	Multiply Accumulate Low Halfword to Word Saturate Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
maclhwso.	Multiply Accumulate Low Halfword to Word Saturate Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	
Multiply-Accumulate Low-Halfword to Word Saturate Unsigned Instructions		rD is added to the unsigned product (rA[16:31]) × (rB[16:31]), producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in rD. Otherwise, the nearest-representable value is stored in rD.	
maclhwsu	Multiply Accumulate Low Halfword to Word Saturate Unsigned	XER and CR0 are <i>not</i> updated.	rD,rA,rB
maclhwsu.	Multiply Accumulate Low Halfword to Word Saturate Unsigned and Record	CR0 is updated to reflect the result.	
maclhwsuo	Multiply Accumulate Low Halfword to Word Saturate Unsigned with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
maclhwsuo.	Multiply Accumulate Low Halfword to Word Saturate Unsigned with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Table 3-42: Multiply-Accumulate Low-Halfword to Word Instructions (*Continued*)

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply-Accumulate Low-Halfword to Word Modulo Unsigned Instructions</i>		rD is added to the unsigned product (rA [16:31]) $\times$ (rB [16:31]), producing a 33-bit result. The low-32 bits of this result are stored in rD .	
macldwu	Multiply Accumulate Low Halfword to Word Modulo Unsigned	XER and CR0 are <i>not</i> updated.	rD,rA,rB
macldwu.	Multiply Accumulate Low Halfword to Word Modulo Unsigned and Record	CR0 is updated to reflect the result.	
macldwuo	Multiply Accumulate Low Halfword to Word Modulo Unsigned with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
macldwuo.	Multiply Accumulate Low Halfword to Word Modulo Unsigned with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Figure 3-30 shows the operation of the multiply-accumulate low-halfword to word instructions.

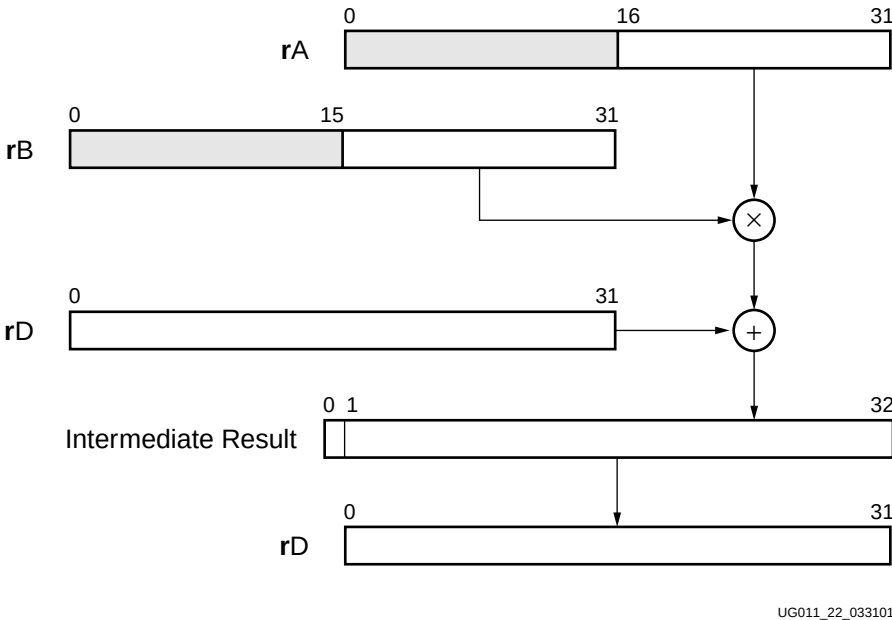


Figure 3-30: Multiply-Accumulate Low-Halfword to Word Operation

Negative Multiply-Accumulate Instructions

Negative Multiply-Accumulate Cross-Halfword to Word Instructions

Table 3-43 shows the PPC405 *negative multiply-accumulate cross-halfword to word* instructions. These instructions take the lower halfword of the first source operand ($rA[16:31]$) and multiply it with the upper halfword of the second source operand ($rB[0:15]$), producing a signed 32-bit product. This product is negated and added to the value in the destination register, rD , producing a 33-bit intermediate result (this is the same as subtracting the product from rD). Generally, rD is loaded with the lower-32 bits of the 33-bit intermediate result. However, if the instruction performs saturating arithmetic and the intermediate result overflows, rD is loaded with the nearest representable value (see **Modulo and Saturating Arithmetic**, above).

For each type of instruction shown in Table 3-43, the “Operation” column indicates the negative multiply-accumulate operation performed. The column also shows, on an instruction-by-instruction basis, how the XER and CR registers are updated (if at all).

Table 3-43: Negative Multiply-Accumulate Cross-Halfword to Word Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Negative Multiply-Accumulate Cross-Halfword to Word Modulo Signed Instructions</i>		The signed product $(rA[16:31]) \times (rB[0:15])$ is subtracted from rD , producing a 33-bit result. The low-32 bits of this result are stored in rD .	
nmacchw	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed	XER and CR0 are <i>not</i> updated.	rD,rA,rB
nmacchw.	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed and Record	CR0 is updated to reflect the result.	
nmacchwo	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
nmacchwo.	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	
<i>Negative Multiply-Accumulate Cross-Halfword to Word Saturate Signed Instructions</i>		The signed product $(rA[16:31]) \times (rB[0:15])$ is subtracted from rD , producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in rD . Otherwise, the nearest-representable value is stored in rD .	
nmacchws	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed	XER and CR0 are <i>not</i> updated.	rD,rA,rB
nmacchws.	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed and Record	CR0 is updated to reflect the result.	
nmacchwso	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
nmacchwso.	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Figure 3-31 shows the operation of the negative multiply-accumulate cross-halfword to word instructions.

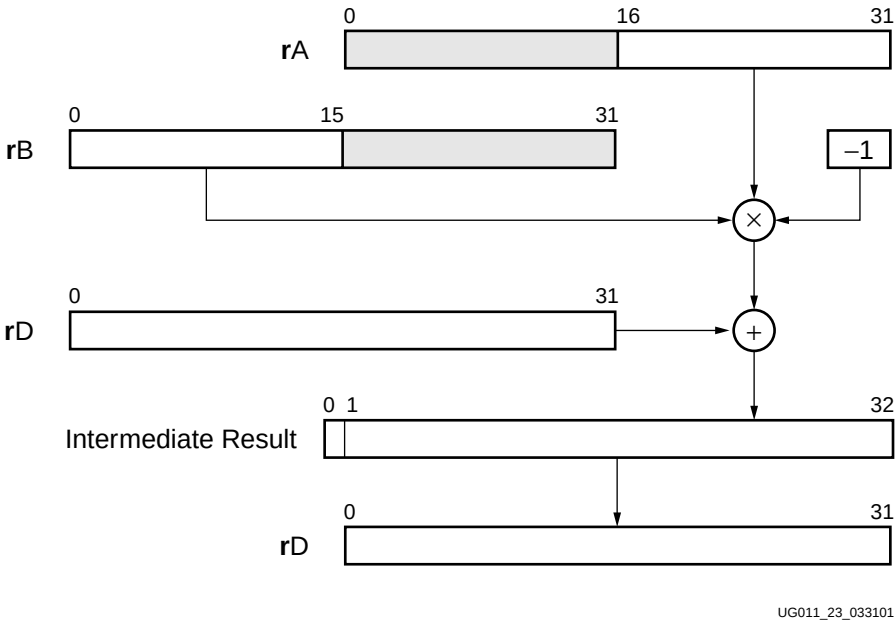


Figure 3-31: Negative Multiply-Accumulate Cross-Halfword to Word Operation

Negative Multiply-Accumulate High-Halfword to Word Instructions

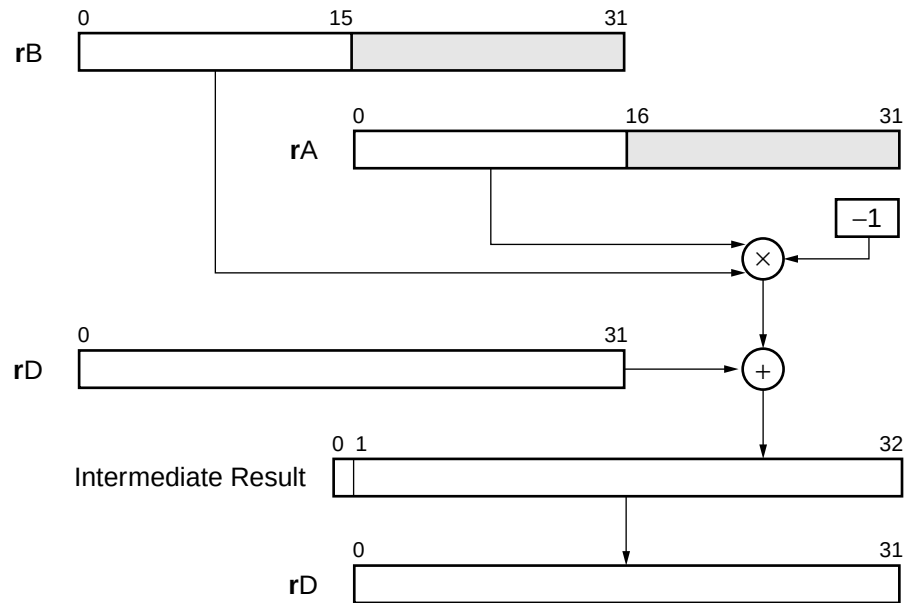
Table 3-44 shows the PPC405 *negative multiply-accumulate high-halfword to word* instructions. These instructions multiply the high halfword of both source operands, $rA[0:15]$ and $rB[0:15]$, producing a signed 32-bit product. This product is negated and added to the value in the destination register, rD , producing a 33-bit intermediate result (this is the same as subtracting the product from rD). Generally, rD is loaded with the lower-32 bits of the 33-bit intermediate result. However, if the instruction performs saturating arithmetic and the intermediate result overflows, rD is loaded with the nearest representable value (see **Modulo and Saturating Arithmetic**, page 441).

For each type of instruction shown in Table 3-44, the “Operation” column indicates the negative multiply-accumulate operation performed. The column also shows, on an instruction-by-instruction basis, how the XER and CR registers are updated (if at all).

Table 3-44: Negative Multiply-Accumulate High-Halfword to Word Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Negative Multiply-Accumulate High-Halfword to Word Modulo Signed Instructions</i>		The signed product ($rA[0:15] \times rB[0:15]$) is subtracted from rD , producing a 33-bit result. The low-32 bits of this result are stored in rD .	
nmachhw	Negative Multiply Accumulate High Halfword to Word Modulo Signed	XER and CR0 are <i>not</i> updated.	rD, rA, rB
nmachhw.	Negative Multiply Accumulate High Halfword to Word Modulo Signed and Record	CR0 is updated to reflect the result.	
nmachhwo	Negative Multiply Accumulate High Halfword to Word Modulo Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
nmachhwo.	Negative Multiply Accumulate High Halfword to Word Modulo Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	
<i>Negative Multiply-Accumulate High-Halfword to Word Saturate Signed Instructions</i>		The signed product ($rA[0:15] \times rB[0:15]$) is subtracted from rD , producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in rD . Otherwise, the nearest-representable value is stored in rD .	
nmachhws	Negative Multiply Accumulate High Halfword to Word Saturate Signed	XER and CR0 are <i>not</i> updated.	rD, rA, rB
nmachhws.	Negative Multiply Accumulate High Halfword to Word Saturate Signed and Record	CR0 is updated to reflect the result.	
nmachhwso	Negative Multiply Accumulate High Halfword to Word Saturate Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
nmachhwso.	Negative Multiply Accumulate High Halfword to Word Saturate Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Figure 3-32 shows the operation of the negative multiply-accumulate high-halfword to word instructions.



UG011_24_033101

Figure 3-32: Negative Multiply-Accumulate High-Halfword to Word Operation

Negative Multiply-Accumulate Low-Halfword to Word Instructions

Table 3-45 shows the PPC405 *negative multiply-accumulate low-halfword to word* instructions. These instructions multiply the low halfword of both source operands, $rA[16:31]$ and $rB[16:31]$, producing a signed 32-bit product. This product is negated and added to the value in the destination register, rD , producing a 33-bit intermediate result (this is the same as subtracting the product from rD). Generally, rD is loaded with the lower-32 bits of the 33-bit intermediate result. However, if the instruction performs saturating arithmetic and the intermediate result overflows, rD is loaded with the nearest representable value (see **Modulo and Saturating Arithmetic**, page 441).

For each type of instruction shown in Table 3-45, the “Operation” column indicates the negative multiply-accumulate operation performed. The column also shows, on an instruction-by-instruction basis, how the XER and CR registers are updated (if at all).

Table 3-45: Negative Multiply-Accumulate Low-Halfword to Word Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Negative Multiply-Accumulate Low-Halfword to Word Modulo Signed Instructions</i>		The signed product $(rA[16:31]) \times (rB[16:31])$ is subtracted from rD , producing a 33-bit result. The low-32 bits of this result are stored in rD .	
nmacldw	Negative Multiply Accumulate Low Halfword to Word Modulo Signed	XER and CR0 are <i>not</i> updated.	rD,rA,rB
nmacldw.	Negative Multiply Accumulate Low Halfword to Word Modulo Signed and Record	CR0 is updated to reflect the result.	
nmacldwo	Negative Multiply Accumulate Low Halfword to Word Modulo Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
nmacldwo.	Negative Multiply Accumulate Low Halfword to Word Modulo Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	
<i>Negative Multiply-Accumulate Low-Halfword to Word Saturate Signed Instructions</i>		The signed product $(rA[16:31]) \times (rB[16:31])$ is subtracted from rD , producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in rD . Otherwise, the nearest-representable value is stored in rD .	
nmacldws	Negative Multiply Accumulate Low Halfword to Word Saturate Signed	XER and CR0 are <i>not</i> updated.	rD,rA,rB
nmacldws.	Negative Multiply Accumulate Low Halfword to Word Saturate Signed and Record	CR0 is updated to reflect the result.	
nmacldwso	Negative Multiply Accumulate Low Halfword to Word Saturate Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
nmacldwso.	Negative Multiply Accumulate Low Halfword to Word Saturate Signed with Overflow Enabled and Record	XER[OV,SO] and CR0 are updated to reflect the result.	

Figure 3-33 shows the operation of the negative multiply-accumulate low-halfword to word instructions.

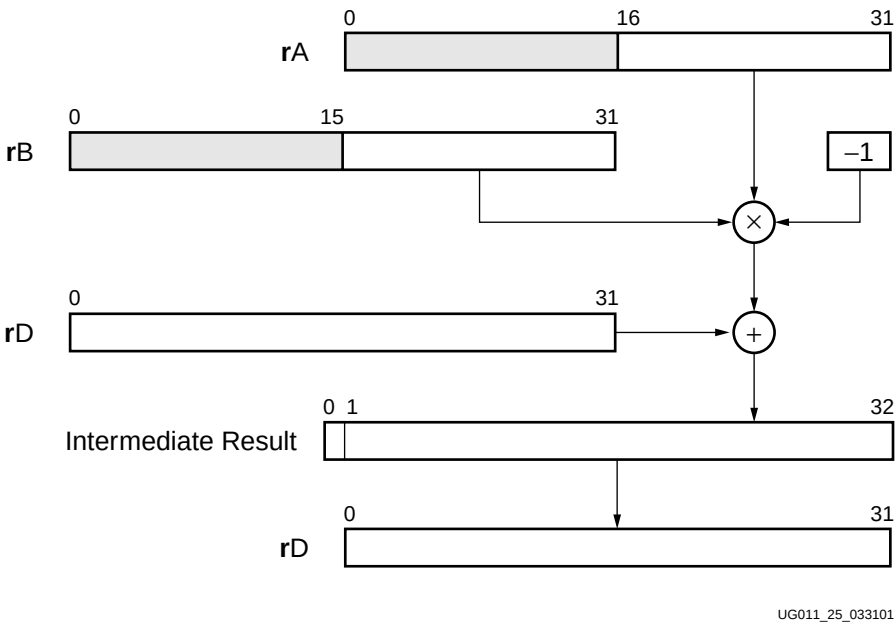


Figure 3-33: Negative Multiply-Accumulate Low-Halfword to Word Operation

Multiply Halfword to Word Instructions

Multiply Cross-Halfword to Word Instructions

Table 3-46 shows the PPC405 *multiply cross-halfword to word* instructions. These instructions take the lower halfword of the first source operand ($rA[16:31]$) and multiply it with the upper halfword of the second source operand ($rB[0:15]$), producing a 32-bit product. The product is signed or unsigned, depending on the instruction.

For each type of instruction shown in Table 3-46, the “Operation” column indicates the multiply operation performed. The column also shows, on an instruction-by-instruction basis, how the CR register is updated (if at all). The XER register is not updated by these instructions.

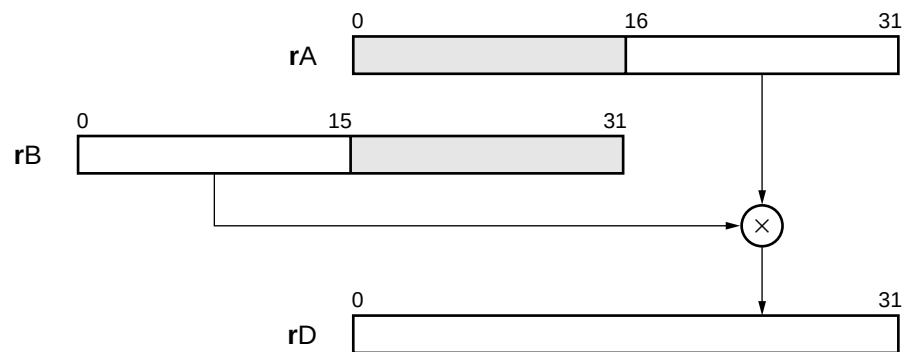
Table 3-46: Multiply Cross-Halfword to Word Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply Cross-Halfword to Word Signed Instructions</i>		rD is loaded with the signed product $(rA[16:31]) \times (rB[0:15])$.	
mulchw	Multiply Cross Halfword to Word Signed	CR0 is <i>not</i> updated.	rD, rA, rB
mulchw.	Multiply Cross Halfword to Word Signed and Record	CR0 is updated to reflect the result.	

Table 3-46: Multiply Cross-Halfword to Word Instructions (*Continued*)

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply Cross-Halfword to Word Unsigned Instructions</i>		rD is loaded with the unsigned product $(rA[16:31]) \times (rB[0:15])$.	
mulchwu	Multiply Cross Halfword to Word Unsigned	CR0 is <i>not</i> updated.	rD,rA,rB
mulchwu.	Multiply Cross Halfword to Word Unsigned and Record	CR0 is updated to reflect the result.	

Figure 3-34 shows the operation of the multiply cross-halfword to word instructions.



UG011_26_033101

Figure 3-34: Multiply Cross-Halfword to Word Operation

Multiply High-Halfword to Word Instructions

Table 3-47 shows the PPC405 *multiply high-halfword to word* instructions. These instructions multiply the high halfword of both source operands, $rA[0:15]$ and $rB[0:15]$, producing a 32-bit product. The product is signed or unsigned, depending on the instruction.

For each type of instruction shown in Table 3-47, the “Operation” column indicates the multiply operation performed. The column also shows, on an instruction-by-instruction basis, how the CR register is updated (if at all). The XER register is not updated by these instructions.

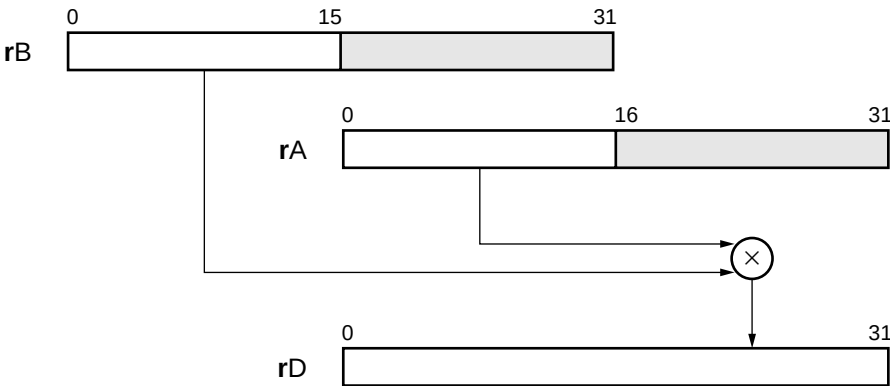
Table 3-47: Multiply High-Halfword to Word Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply High-Halfword to Word Signed Instructions</i>		rD is loaded with the signed product $(rA[0:15]) \times (rB[0:15])$.	
mulhhw	Multiply High Halfword to Word Signed	CR0 is <i>not</i> updated.	rD,rA,rB
mulhhw.	Multiply High Halfword to Word Signed and Record	CR0 is updated to reflect the result.	

Table 3-47: Multiply High-Halfword to Word Instructions (Continued)

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply High-Halfword to Word Unsigned Instructions</i>		rD is loaded with the unsigned product (rA[0:15]) × (rB[0:15]).	
mulhhwu	Multiply High Halfword to Word Unsigned	CR0 is <i>not</i> updated.	rD,rA,rB
mulhhwu.	Multiply High Halfword to Word Unsigned and Record	CR0 is updated to reflect the result.	

Figure 3-35 shows the operation of the multiply high-halfword to word instructions.



UG011_27_033101

Figure 3-35: Multiply High-Halfword to Word Operation

Multiply Low-Halfword to Word Instructions

Table 3-48 shows the PPC405 *multiply low-halfword to word* instructions. These instructions multiply the low halfword of both source operands, rA[16:31] and rB[16:31], producing a 32-bit product. The product is signed or unsigned, depending on the instruction.

For each type of instruction shown in Table 3-48, the “Operation” column indicates the multiply operation performed. The column also shows, on an instruction-by-instruction basis, how the CR register is updated (if at all). The XER register is not updated by these instructions.

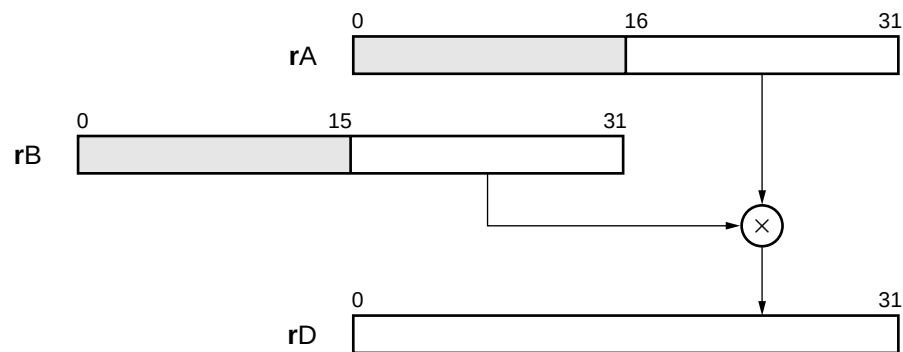
Table 3-48: Multiply Low-Halfword to Word Instructions

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply Low-Halfword to Word Signed Instructions</i>		rD is loaded with the signed product (rA[16:31]) × (rB[16:31]).	
mullhw	Multiply Low Halfword to Word Signed	CR0 is <i>not</i> updated.	rD,rA,rB
mullhw.	Multiply Low Halfword to Word Signed and Record	CR0 is updated to reflect the result.	

Table 3-48: Multiply Low-Halfword to Word Instructions (Continued)

Mnemonic	Name	Operation	Operand Syntax
<i>Multiply Low-Halfword to Word Unsigned Instructions</i>		rD is loaded with the unsigned product (rA[16:31]) × (rB[16:31]).	
mullhwu	Multiply Low Halfword to Word Unsigned	CR0 is <i>not</i> updated.	rD,rA,rB
mullhwu.	Multiply Low Halfword to Word Unsigned and Record	CR0 is updated to reflect the result.	

Figure 3-36 shows the operation of the multiply low-halfword to word instructions.



UG011_28_033101

Figure 3-36: Multiply Low-Halfword to Word Operation

Floating-Point Emulation

The PPC405 is an integer processor and does not support the execution of floating-point instructions in hardware. System software can provide floating-point emulation support using one of two methods.

The preferred method is to supply a call interface to subroutines within a floating-point run-time library. The individual subroutines can emulate the operation of floating-point instructions. This method requires the recompilation of floating-point software in order to add the call interface and link in the library routines.

Alternatively, system software can use the program interrupt. Attempted execution of floating-point instructions on the PPC405 causes a program interrupt to occur due to an illegal instruction. The interrupt handler must be able to decode the illegal instruction and call the appropriate library routines to emulate the floating-point instruction using integer instructions. This method is not preferred due to the overhead associated with executing the interrupt handler. However, this method supports software containing PowerPC floating-point instructions without requiring recompilation. See **Program Interrupt (0x0700)**, page 547, for more information.

Processor-Control Instructions

In user mode, processor-control instructions are used to read from and write to the condition register (CR) and the special-purpose registers (SPRs). Instructions that access the time base are also considered processor-control instructions, but are discussed separately in **Chapter 8, Timer Resources**.

Condition-Register Move Instructions

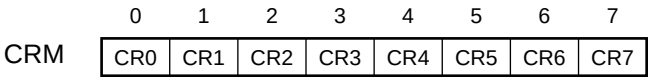
The *condition-register move* instructions shown in [Table 3-49](#) are used to read and write the condition register using a GPR as a destination or source register, and for writing a CR field from the XER register. Not included in this category are other instructions that access the CR. See **Condition-Register Logical Instructions**, [page 412](#), for information on instructions used to manipulate bits and fields in the CR. See **Conditional Branch Control**, [page 403](#), for information on how certain branch instructions use values in the CR as branch conditions.

Table 3-49: Condition-Register Move Instructions

Mnemonic	Name	Operation	Operand Syntax
mcrxr	Move to Condition Register from XER	The CR field specified by the crfD operand is loaded with XER[0:3]. The remaining bits in the CR are not modified. The contents of XER[0:3] are cleared to 0.	crfD
mfcrr	Move from Condition Register	rD is loaded with the contents of CR.	rD
mtcrf	Move to Condition Register Fields	CR is loaded with the contents of rS under the control of a field mask specified by the CRM operand.	CRM, rS

mtcrf Field Mask (CRM)

The **mtcrf** instruction uses an 8-bit field mask (CRM) specified in the instruction encoding to control which CR fields are loaded from **rS**. As shown in [Figure 3-37](#), each bit in CRM corresponds to one of the 4-bit CR fields, with the most-significant CRM bit corresponding to CR0 and the least-significant CRM bit corresponding to CR7. When **mtcrf** is executed, a CR field is loaded with the corresponding bits in **rS** only when the associated CRM mask bit is set to 1. If the mask bit is cleared to 0, the CR field is unchanged.



UG011_31_033101

Figure 3-37: mtcrr Field Mask (CRM) Format

[Figure 3-38](#) shows an example of how the CRM field is used. In this example, CRM = 0b01100100, causing CR1, CR2, and CR5 to be updated with the corresponding bits in **rS**. All remaining CR fields are unchanged.

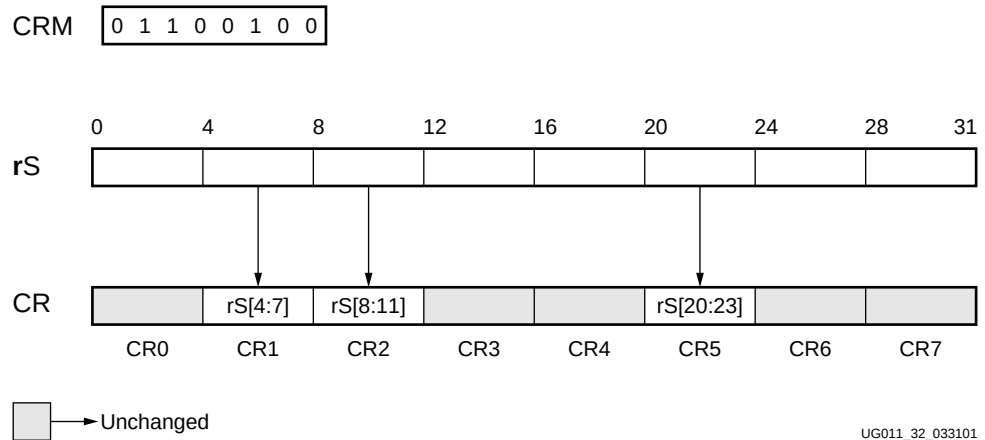


Figure 3-38: mtcrlf Example

Special-Purpose Register Instructions

The *special-purpose register* instructions shown in Table 3-50 are used to read and write the special-purpose registers (SPRs) using a GPR as a destination or source register. The SPR number (SPRN) shown in the operand syntax column appears as a *decimal* value in the assembler listing. Within the instruction opcode, this number is encoded using a *split-field notation*. For more information, see **Split-Field Notation**, page 607.

Table 3-50: Special-Purpose Register Instructions

Mnemonic	Name	Operation	Operand Syntax
mfspir	Move from Special Purpose Register	rD is loaded with the contents of the SPR specified by SPRN.	rD,SPRN
mtspir	Move to Special Purpose Register	The SPR specified by SPRN is loaded with the contents of rS.	SPRN,rS

Synchronizing Instructions

Table 3-51 lists the PowerPC *synchronization* instructions. The types of synchronization defined by the PowerPC architecture are described in **Synchronization Operations**, page 378.

Table 3-51: Synchronizing Instructions

Mnemonic	Name	Operation	Operand Syntax
eieio	Enforce In-Order Execution of I/O	Provides an ordering function for loads and stores. All storage accesses that precede eieio complete before storage accesses following eieio .	—
isync	Instruction Synchronize	Ensures all previous instructions complete before the isync instruction completes. isync also prevents other instructions from beginning execution until the isync instruction completes. Prefetched instructions are discarded so that subsequent instructions are fetched and executed in the context established by instructions preceding the isync . Memory-access ordering is <i>not</i> guaranteed. Memory accesses caused by previous instructions are not necessarily ordered with respect to memory accesses by other devices.	
sync	Synchronize	Ensures that all instructions preceding the sync instruction appear to complete before the sync instruction completes, and that no subsequent instructions are executed until after the sync instruction completes. Memory accesses caused by previous instructions are completed with respect to memory accesses by other devices.	

Implementation of **eieio** and **sync** Instructions

In the PPC405, **eieio** and **sync** are implemented identically for the following reasons:

- The PowerPC architecture only requires the **eieio** instruction to perform storage synchronization, but it does allow PowerPC processors to implement **eieio** as an execution-synchronizing instruction. The PPC405 implements **eieio** in such a manner.
- As defined by the PowerPC architecture, **sync** is used to synchronize memory accesses across all processors in a multiprocessor environment. Because the PPC405 does not provide hardware support for multiprocessor memory coherency, **sync** does not guarantee memory ordering across multiple PPC405 processors. This results in the same storage-synchronization capability as the **eieio** instruction.

In implementations that provide hardware support for multiprocessor memory coherency, **sync** can take significantly longer to execute than **eieio**. PPC405 programmers should consider whether their software is expected to run on other platforms and use the **sync** instruction in favor of **eieio** only when necessary.

Synchronization Effects of PowerPC Instructions

Additional PowerPC instructions can cause synchronizing operations to occur. All instructions that result in some form of synchronization are listed in [Table 3-52](#).

Table 3-52: Synchronization Effects of PowerPC Instructions

Context Synchronizing	Execution Synchronizing	Storage Synchronizing
isync rfci ² rfi ² sc	eieio ¹ isync mtmsr ² rfci ² rfi ² sc sync	eieio sync
Notes: 1. As implemented on the PPC405. 2. Privileged instruction.		

Semaphore Synchronization

Table 3-53 lists the PowerPC *semaphore-synchronization* instructions. These instructions are used to implement common semaphore operations, including test and set, compare and swap, exchange memory, and fetch and add. Examples of these semaphore operations are found in **Synchronization Examples**, [page 873](#).

Table 3-53: Semaphore Synchronization Instructions

Mnemonic	Name	Operation	Operand Syntax
lwarx	Load Word and Reserve Indexed	rD is loaded with the word in memory addressed using register-indirect with index addressing: $EA = (rA 0) + (rB)$ A reservation corresponding to the address is maintained by the processor.	rD,rA,rB
stwcx.	Store Word Conditional Indexed	An effective address is computed using register-indirect with index addressing: $EA = (rA 0) + (rB)$ If a reservation exists, the contents of rS are stored into the memory word specified by the effective address, and the reservation is cleared. If a reservation does not exist, rS is not stored. CR0[EQ] is set to 1 if the reservation exists, otherwise it is cleared to 0.	rS,rA,rB

The **lwarx** and **stwcx.** instructions are typically used by system programs and are called by application programs as needed. Generally, a program uses **lwarx** to load a semaphore from memory, causing a reservation to be set (the processor maintains the reservation internally). The program can compute a result based on the semaphore value and conditionally store the result back to the same memory location using the **stwcx.** instruction. The conditional store is performed based on the existence of the reservation established by the preceding **lwarx** instruction. If the reservation exists when the store is executed, the store is performed and CR0[EQ] is set to 1. If the reservation does not exist when the store is executed, the target memory location is not modified and CR0[EQ] is cleared to 0.

If the store is successful, the sequence of instructions from the semaphore load to the semaphore store appear to be executed *atomically*—no other device modified the semaphore location between the read and the update. Other devices can read from the semaphore location during the operation.

For a semaphore operation to work properly, the **lwarx** instruction must be paired with an **stwcx.** instruction, and both must specify identical effective addresses. The reservation granularity in the PPC405 is a word. For both instructions, the effective address must be word aligned, otherwise an alignment exception occurs.

In the PPC405, the conditional store is always performed when a reservation exists, even if the store address does not match the load address that set the reservation. This operation is allowed by the PowerPC architecture, but is not guaranteed to be supported on all PowerPC implementations. It is good programming practice to always specify identical addresses for **lwarx** and **stwcx.** pairs.

The PPC405 can maintain only one reservation at a time. The address associated with the reservation can be changed by executing a subsequent **lwarx** instruction. The conditional store is performed based upon the reservation established by the *last* **lwarx** instruction executed. Executing an **stwcx.** instruction always clears a reservation held by the processor, whether the address matches that established by the **lwarx.**

Exceptions do not clear reservations, although an interrupt handler can clear a reservation.

Memory-Control Instructions

Table 3-54 lists the PowerPC *memory-control* instructions available to programs running in user mode. See **Cache Instructions**, [page 492](#) for a detailed description of each instruction.

Table 3-54: Memory-Control Instructions, User Mode

Mnemonic	Name
dcba	Data Cache Block Allocate
dcbf	Data Cache Block Flush
dcbst	Data Cache Block Store
dcbt	Data Cache Block Touch
dcbtst	Data Cache Block Touch for Store
dcbz	Data Cache Block Set to Zero
icbi	Instruction Cache Block Invalidate
icbt	Instruction Cache Block Touch

PPC405 Privileged-Mode Programming Model

This chapter presents an overview of the processor resources and instructions available to privileged-mode programs running on the PPC405. These resources and instructions are part of the *privileged-programming model*. From privileged mode, software can access all processor resources and can execute all instructions supported by the PPC405. Typically, only system software runs in privileged mode and applications run in user mode.

The remaining chapters in this book present portions of the system-programming resources in greater detail, as follows:

- **Chapter 5, Memory-System Management** describes the resources available for managing the caches and memory protection.
- **Chapter 6, Virtual-Memory Management** describes the PPC405 address-translation capabilities.
- **Chapter 7, Exceptions and Interrupts** describes the exception mechanism and how the processor interrupts program execution so that exceptions can be handled.
- **Chapter 8, Timer Resources** describes the time base and timer registers.
- **Chapter 9, Debugging** describes the resources available in the PPC405 for debugging software and hardware.

Privileged Registers

Figure 4-1 shows additional registers supported by the PPC405 in privileged mode. These registers are accessed by software only when the processor is operating in privileged mode. In the PPC405, all privileged registers are 32 bits wide except for the time base, as described in **Time Base, page 560**.

The machine-state register, SPR general-purpose registers, and processor-version register are described in the following sections of this chapter. This chapter also describes device control registers which are implemented outside the PPC405 but are accessed by software running on the PPC405. The remaining privileged registers are described in other chapters as follows:

- The core-configuration register (CCR0) is described in **Cache Control, page 492**.
- The processor ID register (PID) is described in **Virtual Mode, page 508**.
- The zone-protection register (ZPR) is described in **Virtual-Mode Access Protection, page 518**.
- The storage-attribute control registers are described in **Memory-System Control, page 487**.
- The exception-handling registers are described in **Interrupt-Handling Registers, page 533**.

- The debug registers are described in **Debug Registers**, [page 573](#).
- The timer registers, including the time base, are described in **Timer Resources**, [page 559](#).



UG011_33_033101

Figure 4-1: PPC405 Privileged Registers

Special-Purpose Registers

All privileged PPC405 registers except for the machine-state register are *special-purpose registers*, or SPRs. See [Appendix A, Special-Purpose Registers, page 806](#) for a complete list of all SPRs (user and privileged) supported by the PPC405.

SPRs are read and written using the *move from special-purpose register (mfspr)* and *move to special-purpose register (mtspr)* instructions. See [Special-Purpose Register Instructions, page 471](#), for more information on these instructions. Simplified instruction mnemonics are available for the **mtspr** and **mfspr** instructions when accessing certain SPRs. See [Special-Purpose Registers, page 866](#), for more information.

Machine-State Register

The machine-state register (MSR) is a 32-bit register that defines the processor state. [Figure 4-2](#) shows the format of the MSR. The bits in the MSR are defined as shown in [Table 4-1](#). All system software can read and write the MSR using the *move from machine-state register (mfmsr)* and *move to machine-state register (mtmsr)* instructions. The external-interrupt enable (MSR[EE]) bit can also be updated using the *write external enable* instructions (**wrtee** and **wrteei**). See [Machine-State Register Instructions, page 471](#), for more information on these instructions.

The MSR is also modified during execution of the *system-call* instruction (**sc**), *return-from-interrupt* instructions (**rfi** and **rfdi**), and by the exception mechanism during a control transfer to an interrupt handler.

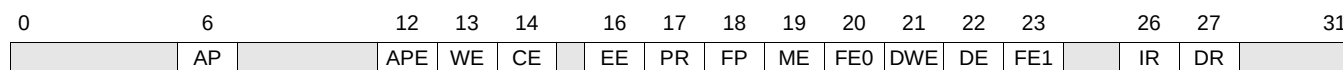


Figure 4-2: Machine-State Register (MSR)

Table 4-1: Machine-State Register (MSR) Bit Definitions

Bit	Name	Function	Description
0:5		Reserved	
6	AP	Auxiliary Processor Available (Unsupported)	This bit is unsupported and ignored by the PPC405D5. Software should clear this bit to 0.
7:11		Reserved	
12	APE	APU Exception Enable (Unsupported)	This bit is unsupported and ignored by the PPC405D5. Software should clear this bit to 0.
13	WE	Wait State Enable 0—Disabled. 1—Enabled.	When in the wait state, the processor stops fetching and executing instructions, and no longer performs memory accesses. The processor remains in the wait state until an interrupt or a reset occurs, or an external debug tool clears WE. See Processor Wait State, page 472 , for more information.
14	CE	Critical Interrupt Enable 0—Disabled. 1—Enabled.	Controls the critical-input interrupt and the watchdog-timer interrupt. See Interrupt Reference, page 538 , for more information on these interrupts.
15		Reserved	
16	EE	External Interrupt Enable 0—Disabled. 1—Enabled.	Controls the external interrupts, the programmable-interval timer interrupt, and the fixed-interval timer interrupt. See Interrupt Reference, page 538 , for more information on each interrupt.

Table 4-1: Machine-State Register (MSR) Bit Definitions (Continued)

Bit	Name	Function	Description
17	PR	Privilege Level 0—Privileged mode. 1—User mode.	Controls the privilege level of the processor. See Processor Operating Modes , page 379 , for more information.
18	FP	Floating-Point Available (Unsupported)	This bit is unsupported and ignored by the PPC405D5. Software should clear this bit to 0.
19	ME	Machine-Check Enable. 0—Disabled. 1—Enabled.	Controls the machine-check interrupt. See Machine-Check Interrupt (0x0200) , page 540 , for more information.
20	FE0	Floating-Point Exception-Mode 0 (Unsupported)	This bit is unsupported and ignored by the PPC405. Software should clear this bit to 0.
21	DWE	Debug Wait Enable 0—Disabled. 1—Enabled.	Controls the debug wait mode. See Debug-Wait Mode , page 573 , for more information.
22	DE	Debug Interrupt Enable 0—Disabled. 1—Enabled.	Controls the debug interrupt. See Debug Interrupt (0x2000) , page 557 , for more information.
23	FE1	Floating-Point Exception-Mode 1 (Unsupported)	This bit is unsupported and ignored by the PPC405D5. Software should clear this bit to 0.
24:25		Reserved	
26	IR	Instruction Relocate 0—Instruction-address translation is disabled. 1—Instruction-address translation is enabled.	Controls instruction-address translation. See Chapter 6, Virtual-Memory Management , for more information. When address translation is disabled, the processor is running in real mode. See Real Mode , page 507 , for an introduction.
27	DR	Data Relocate 0—Data-address translation is disabled. 1—Data-address translation is enabled.	Controls data-address translation. See Chapter 6, Virtual-Memory Management , for more information. When address translation is disabled, the processor is running in real mode. See Real Mode , page 507 , for an introduction.
28:31		Reserved	

The initial state of the MSR following a processor reset is described in **Machine-State Register**, [page 598](#).

SPR General-Purpose Registers

The SPR general-purpose registers (SPRG0–SPRG7) are 32-bit registers that can be used for any purpose by system software running in privileged mode. The values stored in these registers do not affect the operation of the PPC405 processor.

Four of the registers (SPRG4–SPRG7) are available from user mode with *read-only access*. Application software can read the contents of SPRG4–SPRG7, but cannot modify them.

The format of all SPRG n registers is shown in [Figure 4-3](#).

0

31

General-Purpose System-Software Data

Figure 4-3: **SPR General-Purpose Registers (SPRG0–SPRG7)**

The SPRG n registers are privileged SPRs with the following addresses:

- SPRG0—272 (0x110)
- SPRG1—273 (0x111)
- SPRG2—274 (0x112)
- SPRG3—275 (0x113)
- SPRG4—276 (0x114)
- SPRG5—277 (0x115)
- SPRG6—278 (0x116)
- SPRG7—279 (0x117)

These registers are read and written using the **mf spr** and **mt spr** instructions. User-mode software that reads SPRG4–SPRG7 accesses them using different SPR numbers (see [page 401](#)).

Processor-Version Register

The processor-version register (PVR) is a 32-bit read-only register that uniquely identifies the processor. [Figure 4-4](#) shows the format of the PVR.

The PVR's PCL bits [22:25] vary according to the Virtex-II Pro™ device type. The PVR has a total value of 0x2001_0820 in the 2VP4 and 2VP7 devices (each containing a single processor block), and 0x2001_0860 in the 2VP20 and 2VP50 devices (containing two and four processor blocks respectively). The bit definitions are shown in [Table 4-2](#).

0	11	12	15	16	21	22	25	26	31
OWN				PCF	CAS		PCL	AID	

Figure 4-4: **Processor-Version Register (PVR)**Table 4-2: **Processor-Version Register (PVR) Bit Definitions**

Bit	Name	Function/Value	Description
0:11	OWN	Owner Identifier 0b 0010_0000_0000 (0x200)	Identifies Xilinx as the owner of the processor core.
12:15	PCF	Processor Core Family 0b 0001 (0x1)	Identifies the processor as belonging to the 405 processor-core family.
16:21	CAS	Cache Array Sizes 0b 0000_10 (0x02)	Identifying the processor as containing 16KB instruction and 16KB data caches.
22:25	PCL	Processor Core Revision Level 0b 00_00 (0x0) for 2VP4, 2VP7 devices 0b 00_01 (0x1) for 2VP20, 2VP50 devices	Identifies the processor-core revision level. This value is incremented when a revision is made to the processor core. Differs according to the Xilinx Virtex-II Pro device type.
26:31	AID	ASIC Identifier 0b 10_0000 (0x20)	

The PVR is a privileged *read-only* SPR with an address of 287 (0x11F). It is read using the **mf spr** instruction. Write access is not supported.

Device Control Registers

Device control registers (DCRs) are 32-bit registers implemented in FPGA logic gates. They are not contained within the processor core. The PowerPC embedded-environment architecture and PowerPC Book-E architecture define the existence of a DCR-address space and the instructions that access the DCRs, but they do not define what the DCRs do or how they are to be used. System developers can define DCRs for use in controlling the operations of on-chip buses, peripherals, and some processor behavior. The processor reads and writes the DCRs over the DCR-bus interface using the **mfdcr** and **mtdcr** instructions.

See the *PPC405 Processor Block Manual* for more information on implementing and using DCRs.

Privileged Instructions

Table 4-3 lists the privileged instructions supported by the PPC405. Attempted use of these instructions when running in user mode causes a program exception.

Table 4-3: PPC405 Privileged Instructions

System Linkage	Processor Control	Memory-System Management	Virtual-Memory Management
rfci	mfdcr	dcbi	tlbia
rfi	mfmsr	dccci	tlbre
sc	mf spr ⁽¹⁾	dcread	tlbsx
	mt dcr	iccci	tlbsync
	mtmsr	icread	tlbwe
	mt spr ⁽²⁾		
	wrt ee		
	wrt eei		

Notes:

- Except for CTR, LR, SPRG4–SPRG7, and XER.
- Except for CTR, LR, and XER.

System Linkage

Application (user-mode) programs transfer control to system-service routines (privileged-mode programs) using the *system-call* instruction, **sc**. Executing the **sc** instruction causes a system-call exception to occur. The system-call interrupt handler determines which system-service routine to call and whether the calling application has permission to call that service. If permission is granted, the system-call interrupt handler performs the actual procedure call to the system-service routine on behalf of the application program. This call is typically performed using a branch instruction that updates the link register with the return address.

The execution environment expected by the system-service routine requires the execution of prologue instructions to set up that environment. Those instructions usually create the block of storage that holds procedural information (the *activation record*), update and initialize pointers, and save volatile registers (registers the system-service routine uses). Prologue code can be inserted by the linker when creating an executable module, or it can be included as stub code in either the system-call interrupt handler or the system-library routines.

Returns from the system-service routine reverse the process described above. Control is transferred back to the system-call interrupt handler using a branch to link-register

instruction. Epilog code is executed to unwind and deallocate the activation record, restore pointers, and restore volatile registers. The interrupt handler executes a return-from-interrupt instruction (**rfi**) to return to the application.

Table 4-4 lists the PowerPC *system-linkage* instructions. The **sc** instruction can be executed from user mode and privileged mode. The **rfi** and **rfci** instructions are executed only from privileged mode.

Table 4-4: System-Linkage Instruction

Mnemonic	Name	Operation	Operand Syntax
rfi	Return from Interrupt	Return from noncritical-interrupt handler. See Returning from Interrupt Handlers , page 530 , for more information.	—
rfci	Return from Critical Interrupt	Return from critical-interrupt handler. See Returning from Interrupt Handlers , page 530 , for more information.	—
sc	System Call	Causes a system-call exception to occur. See System-Call Interrupt (0x0C00) , page 550 , for more information.	—

Processor-Control Instructions

In privileged mode, processor-control instructions are used to read from and write to the machine-state register and the special-purpose registers. Instructions that access the time base registers are also considered processor-control instructions, but are discussed separately in [Chapter 8, Timer Resources](#).

Machine-State Register Instructions

The *machine-state register* instructions shown in **Table 4-5** are used to read and write the machine-state register (MSR) using a GPR as a destination or source register. The **mtmsr** instruction shown in **Table 4-5** is execution synchronizing. See **Execution Synchronization**, [page 378](#), for more information.

Table 4-5: Machine-State Register Instructions

Mnemonic	Name	Operation	Operand Syntax
mfmsr	Move from Machine State Register	rD is loaded with the contents of the machine-state register.	rD
mtmsr	Move to Machine State Register	The machine-state register is loaded with the contents of rS.	rS
wrtee	Write External Enable	MSR[EE] (bit 16) is loaded with the value in rS ₁₆ .	rS
wrteei	Write External Enable Immediate	MSR[EE] (bit 16) is loaded with the immediate value of the instruction E field.	E

Special-Purpose Register Instructions

The *special-purpose register* instructions shown in **Table 4-6** are used to read and write the special-purpose registers (SPRs) using a GPR as a destination or source register. The SPR number (SPRN) shown in the operand syntax column can be specified as a decimal or hexadecimal value in the assembler listing. Within the instruction opcode, this number is

encoded using a *split-field notation*. For more information, see **Split-Field Notation**, [page 607](#).

Table 4-6: Special-Purpose Register Instructions

Mnemonic	Name	Operation	Operand Syntax
mf spr	Move from Special Purpose Register	rD is loaded with the contents of the SPR specified by SPRN.	rD,SPRN
mt spr	Move to Special Purpose Register	The SPR specified by SPRN is loaded with the contents of rS.	SPRN,rS

Simplified instruction mnemonics are available for the **mtspr** and **mf spr** instructions when accessing certain SPRs. See **Special-Purpose Registers**, [page 866](#), for more information.

Device Control Register Instructions

The *device control register* instructions shown in [Table 4-7](#) are used to read and write the device control registers (DCRs) using a GPR as a destination or source register. The DCR number (DCRN) shown in the operand syntax column can be specified as a decimal or hexadecimal value in the assembler listing. Within the instruction opcode, this number is encoded using a *split-field notation*. For more information, see **Split-Field Notation**, [page 607](#).

Table 4-7: Device Control Register Instructions

Mnemonic	Name	Operation	Operand Syntax
mf dcr	Move from Device Control Register	rD is loaded with the contents of the DCR specified by DCRN.	rD,DCRN
mt dcr	Move to Device Control Register	The DCR specified by DCRN is loaded with the contents of rS.	DCRN,rS

Processor Wait State

Software-controlled power management is possible through the use of the processor *wait state*. Wait state is a low-power operating mode that can be used to conserve processor energy when the processor is not busy. Wait state is entered when software sets the *wait-state enable* bit (MSR[WE]) to 1.

When in the wait state, the processor stops fetching and executing instructions, and no longer performs memory accesses. The processor continues to respond to interrupts, and can be restarted through the use of external interrupts or timer interrupts. Wait state can also be exited when an external debug tool clears WE or when a reset occurs.

Memory-System Management

This chapter describes how software can manage the interaction between the PPC405 processor and the memory system. Memory-system management includes cache control, the use of storage attributes, and memory-coherency considerations. The virtual-memory environment is described separately in [Chapter 6, Virtual-Memory Management](#).

Memory-System Organization

[Figure 5-1](#) shows the memory-system organization supported by the PPC405. The processor implements separate internal instruction and data caches, an architectural construct known as the *Harvard cache model*. The PPC405 does not provide hardware support for attachment of a level-2 (L2) or higher caches. The processor communicates with system memory over the processor local bus (PLB), usually through a memory controller.

The PowerPC architecture does not define the type, organization, implementation, or existence of internal or external caches. The cache structure of other PowerPC processors can differ from that implemented by the PPC405. To maximize portability, software that operates on multiple PowerPC implementations should always assume implementation of a Harvard cache model.

Separate instruction and data *on-chip-memory* (OCM) can be attached to the PPC405 cache controllers using a dedicated processor interface. The performance of OCM accesses can be identical to that of a cache hit, depending on how much block RAM (BRAM) is connected to the processor through the OCM controllers. Refer to the [PPC405 Processor Block Manual](#) for more information on the OCM and OCM controllers.

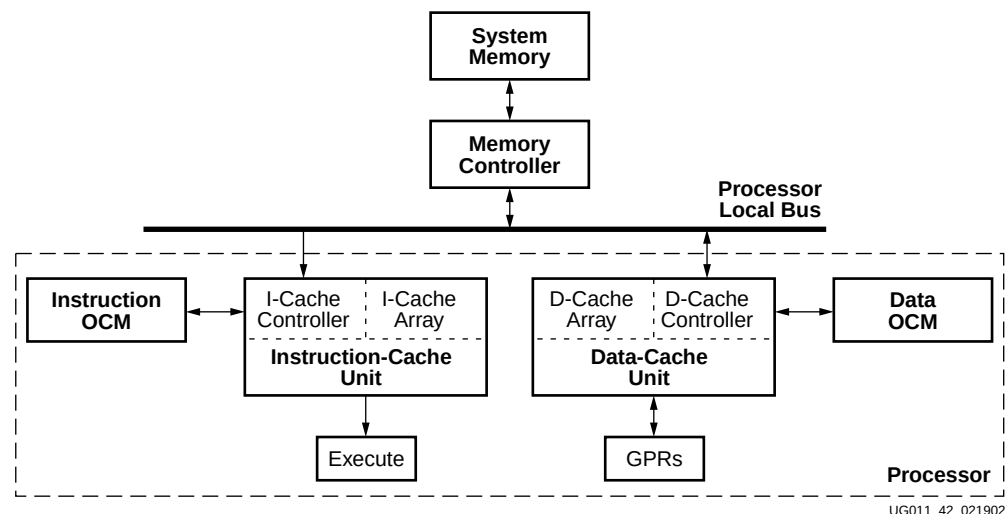


Figure 5-1: PPC405 Memory-System Organization

Memory-System Features

The PPC405 memory system supports the following features:

- Separate 64-bit instruction and 64-bit data interfaces to the processor local bus (PLB).
- Separate 64-bit instruction and 32-bit data interfaces to the on-chip memory (OCM).
- Single-cycle access to the OCM (depending on how much BRAM is connected to the processor), matching the access time for cache hits.
- Independent, programmable PLB-request priority for the instruction and data interfaces.
- Support for big-endian and little-endian memory systems.
- Support for unaligned load and store operations.
- Separate instruction and data caches (Harvard cache model) with the following characteristics:
 - 16 KB 2-way set-associative cache arrays.
 - 32-byte cachelines.
 - Programmable line allocation for instruction fetches, data loads, and data stores.
 - Non-blocking access for cache hits during line fills (the data cache is also non-blocking during cache flushes).
 - Critical-word bypass for cache misses.
 - Programmable PLB request size for non-cacheable memory requests.
 - A complete set of cache-control instructions.
- Specific features supported by the instruction-cache include:
 - A virtually-indexed and physically-tagged cache array.
 - Programmable address pipelining and prefetching for cache misses and non-cacheable requests.
 - Buffering of up to eight non-cacheable instructions in the fill buffer.
 - Support for non-cacheable hits into the fill buffer.
 - Flash invalidate—one instruction invalidates the entire cache.
- Specific features supported by the data-cache include:
 - A physically-indexed and physically-tagged cache array.
 - Flexible control over write-back and write-through strategies for each cacheable memory region.
 - Address pipelining for cache misses.
 - Buffering of up to 32 bytes of data in the fill buffer.
 - Support for non-cacheable hits into the fill buffer.
 - Handling of up to two pending cacheline flushes.
 - Handling of up to three pending stores before causing a pipeline stall.

Cache Organization

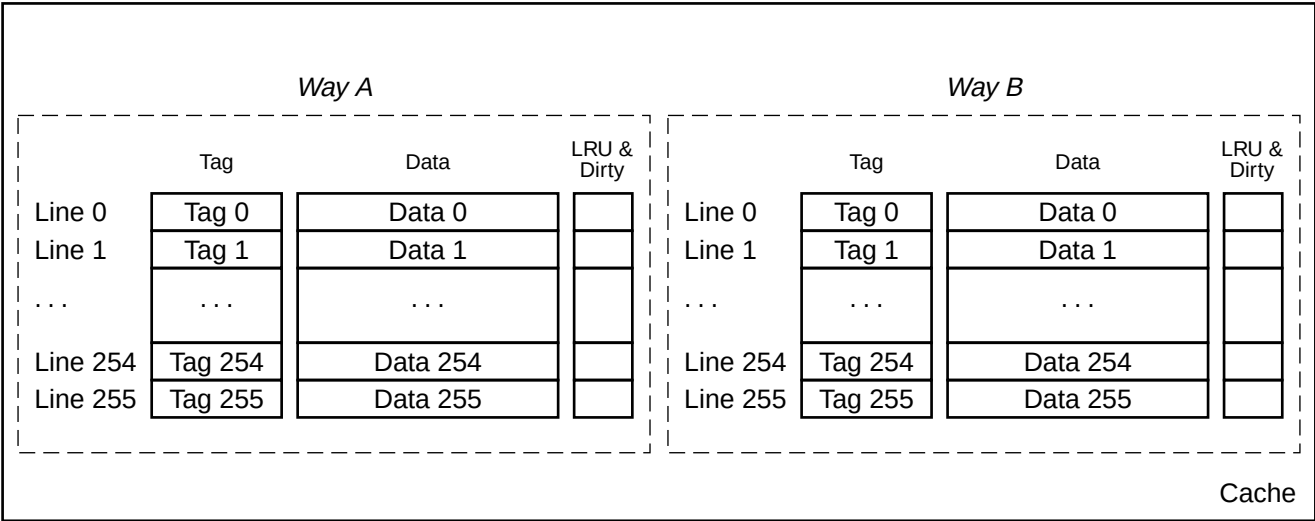
The PPC405 contains an instruction-cache unit and a data-cache unit. Each cache unit contains a 16 KB, 2-way set-associative cache array, plus control logic for managing cache accesses. The caches contain copies of the most frequently used instructions and data and can typically be accessed much faster than system memory.

Figure 5-2 shows the logical structure of the PPC405 cache arrays. Each cache array is organized as a collection of *cachelines*. There are a total of 512 cachelines in a cache array, divided evenly into two *ways* (one way contains 256 lines). Line *n* from way A and line *n* from way B make up a *set* of cachelines, also known as a *congruence class*. A cache array contains a total of 256 sets, or congruence classes.

Each cacheline contains the following pieces of information:

- A *tag* used to uniquely identify the line within the congruence class.
- 32 bytes of *data* that are a copy of a contiguous, 32-byte block of system memory, aligned on a 32-byte address boundary. The data can represent either instructions (in the instruction cache) or operands (in the data cache).
- An *LRU* bit that specifies which cacheline within the congruence class is least-recently used. Each time a cacheline is accessed, the cache controller marks the *other* line within that congruence class as least-recently used. When a new cacheline is read from memory during a cacheline fill, the line in the congruence class marked least-recently used is replaced.
- A *dirty* bit that indicates whether the cacheline contains modified information. A modified cacheline contains data that is more recent than the copy in system memory. The instruction cache does not have a dirty bit.

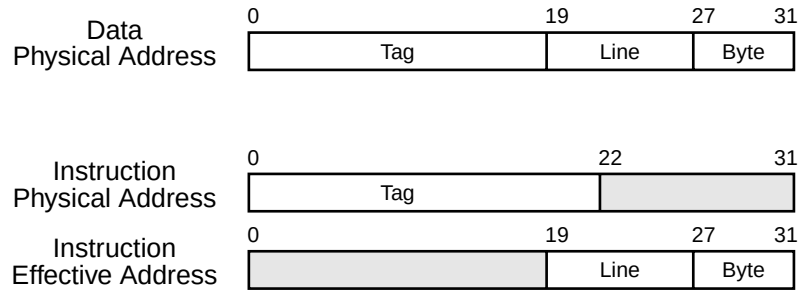
The 512 total lines of 32 bytes each yields a 16 KB cache size.



UG011_34_033101

Figure 5-2: Logical Structure of the PPC405 Cache Arrays

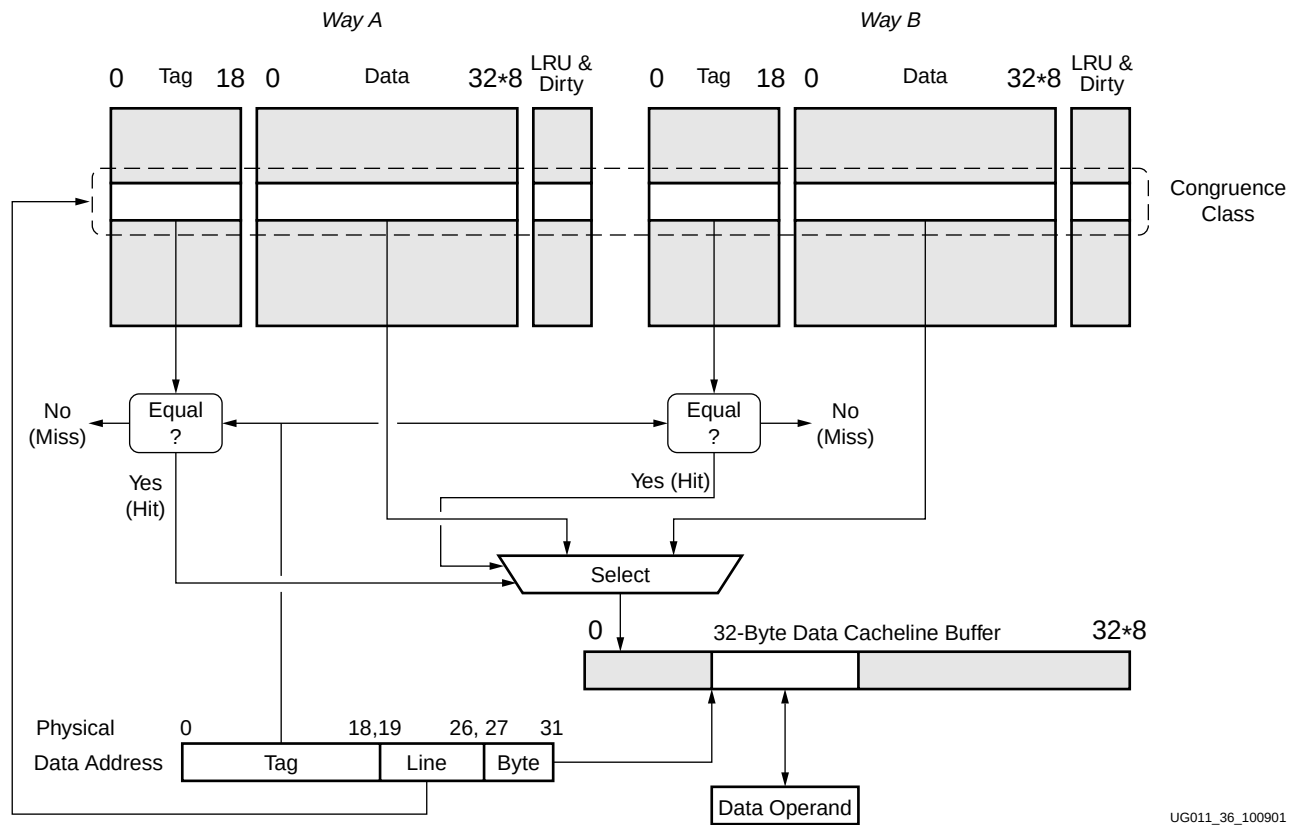
Data is selected from the data cache using fields within the data address. Likewise, an instruction is selected from the instruction cache using fields within the instruction address. The data cache is *physically tagged* and *physically indexed*. This means that the physical address alone is used to access the data-cache array. The instruction cache is *physically tagged* and *virtually indexed*. Here, the effective address is used to specify a congruence class (set of lines) within the cache, and the physical address is used to specify a specific tag. The instruction cache is accessed in this manner for performance reasons, but care is required to avoid cache synonyms (see **Instruction-Cache Synonyms**, page 478). **Figure 5-3** shows the address fields used in accessing the two caches.



UG011_35_033101

Figure 5-3: Address Fields Used to Access Caches

Figure 5-4 shows an example of how the physical-address fields are used to select a data operand from the data-cache array. The instruction cache operates in a similar manner, using fields from both the physical address and the effective address.



UG011_36_100901

Figure 5-4: Data-Cache Access Example

Referring to Figure 5-4, the line field in the data address is used to select a congruence class from the cache array. The congruence class contains two lines, one from each way. Each line contains a tag, meaning two tags are present in a congruence class. The tag field in the data address is compared to both tags in the congruence class. A *hit* occurs when the data-address tag field is equal to one of the two tags. A *miss* occurs when the data-address tag field is not equal to either of the tags.

When a hit occurs, the cacheline with the matching tag is selected. The data in the selected cacheline is loaded into the 32-byte data-cacheline buffer. The byte field in the data address is used as an offset into the line buffer. The data located at that byte offset (byte, halfword, or word) is read from or written to the line buffer, depending on the operation that initiated the cache access.

Access into the instruction cache operates in a near-identical fashion. The difference is in how the 32-byte instruction line buffer is accessed. The line buffer is accessed using the byte field from the instruction effective address. However, the low-order two bits (EA_{30:31}) are ignored, aligning the access on a word boundary. Four bytes are always read from this word-aligned location in the instruction cacheline buffer.

Instruction-Cache Operation

Figure 5-5 shows how instructions flow from the instruction-cache unit (ICU) to the execution pipeline.

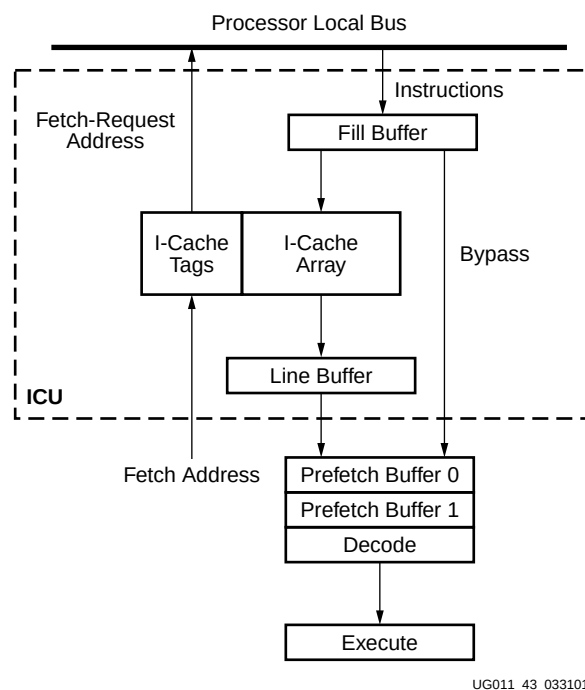


Figure 5-5: Instruction Flow from the Instruction-Cache Unit

All instruction-fetch requests are handled by the ICU. If a fetch address is cacheable, the ICU examines the instruction cache for a hit. When a hit occurs, the cacheline is read from the instruction cache and loaded into the line buffer. Individual instructions are sent from the line buffer to the instruction queue. From there they are either loaded into one of the prefetch buffers or are immediately decoded, depending on the current state of the decode and execution pipelines. Up to two instructions per clock cycle can be sent to the instruction queue from the line buffer.

When a cache miss occurs, or when an instruction address is not cacheable, the ICU sends the fetch-address request to system memory over the processor local bus (PLB). A cache miss results in a cacheline fill, which appears as an eight-word request on the PLB. The request size for non-cacheable instructions can be either four words (half line) or eight words (full line) and is programmable using the CCR0 register (see **Core-Configuration Register**, page 495). Full-line (cacheable and non-cacheable) and half-line fetch requests are always completed (never aborted), even if the instruction stream branches before the

remaining instructions are received. As instructions are received by the ICU from the PLB, they are placed in the fill buffer.

The ICU requests the target instruction first, but the order instructions are returned depends on the design of the PLB device that handles the request (typically a memory controller). When the ICU receives the target instruction, it is immediately forwarded from the fill buffer to the instruction queue over the bypass path. The remaining instructions are received from the PLB and placed in the fill buffer. Subsequent instruction fetches read an instruction from the fill buffer if it is already present in the buffer. If a cache miss occurred, the instruction-cacheline is loaded with the fill-buffer contents after all instructions are received.

Instruction Cacheability Control

Control of instruction cacheability depends on the address-translation mode as follows:

- In real mode, the instruction-cache cacheability register (ICCR) specifies which physical-memory regions are cacheable. See **Instruction-Cache Cacheability Register (ICCR)**, [page 490](#), for more information.
- In virtual mode, the storage-attribute fields in the page-translation look-aside buffer entry (TLB entry) specify which virtual-memory regions are cacheable. See **Storage-Attribute Fields**, [page 514](#), for more information.

After a processor reset, the processor operates in real mode and all physical-memory regions are marked as non-cacheable (all ICCR bits are cleared to 0). Prior to specifying memory regions as cacheable, software must invalidate the instruction cache by executing the **iccci** instruction. (see **Cache Instructions**, [page 492](#), for information on this instruction). After the cache is invalidated, the ICCR can be configured.

Core-Configuration Register, [page 495](#), describes additional software controls that can be used to manage instruction prefetching from cacheable and non-cacheable memory.

Instruction-Cache Hint Instruction

The PowerPC embedded-environment architecture and PowerPC Book-E architecture define an *instruction-cache block touch* (**icbt**) instruction that can be used to improve instruction-cache performance. Software uses **icbt** to indicate that instruction-fetching is likely to occur from the specified address in the near future. When PLB bandwidth is available, the processor can prefetch the instruction-cacheline associated with the **icbt** operand address. This instruction executes as a no-operation if loading the cacheline results in a page-translation exception or a protection exception.

Instruction-Cache Synonyms

NOTE: *The following information applies only if instruction address translation is enabled.*

Proper cache operation depends on a physical address being cached by at most one cacheline. An instruction-cache *synonym* exists when a single physical address is cached by multiple instruction-cachelines. This can occur when software uses page translation to map multiple virtual addresses to the same physical address. Cache synonyms pose serious problems for system software when managing memory-access protection, page translation, and coherency.

In the PPC405, the instruction cache is physically tagged and virtually indexed. When translation is enabled, the physical address is translated from the virtual address. A synonym can exist when common bit ranges in the virtual address and physical address are used to access the cache. This occurs when bits in the virtual index are involved in translating physical-tag bits.

To illustrate the problem, assume 4 KB page translation maps two virtual addresses, 0x8888_8000 and 0xFFFF_F000, to the same physical address, 0x4444_4000 (see **Chapter 6, Virtual-Memory Management** for information on address translation). When a 4 KB page address is translated, the translation mechanism maps each effective-page number (EA_{0:19})

to the same physical-page number ($RA_{0:19}$). Both effective-page numbers ($0x8888\_8$ and $0xFFFF\_F$) are translated into the physical-page number $0x4444\_4$. The effective-page offset ($0x000$) is not translated and is used as the physical-page offset ($RA_{20:31} = EA_{20:31}$).

The ICU uses $RA_{0:21}$ as the tag and $EA_{19:26}$ as the index when accessing the instruction cache. Overlap between tag and index exists in the bit range 19:21. However, only EA_{19} is used to both index the cache and translate part of the physical tag ($EA_{20:21}$ is not used to translate 4 KB virtual pages). In this example, a synonym exists because the effective addresses differ in EA_{19} . The two virtual addresses select different cachelines, even though the address translation mechanism maps them to a single physical address.

Because the PPC405 supports variable page sizes, different high-order EA bits are used to translate pages. The result is that synonyms can occur to varying degrees based on page size:

- 1 KB pages—three bits ($EA_{19:21}$) are used in indexing and tag comparison, resulting in as many as eight synonyms
- 4 KB pages—one bit (EA_{19}) is used in indexing and tag comparison, resulting in two possible synonyms

The following two options are available for preventing cache synonyms:

- Avoid mapping multiple virtual pages into a single physical page when using 1 KB or 4 KB pages sizes
- Use pages sizes of 16 KB or greater if multiple virtual pages must be mapped into a single physical page

Data-Cache Operation

Figure 5-6 shows how data flows between the data-cache unit (DCU) and the general-purpose registers.

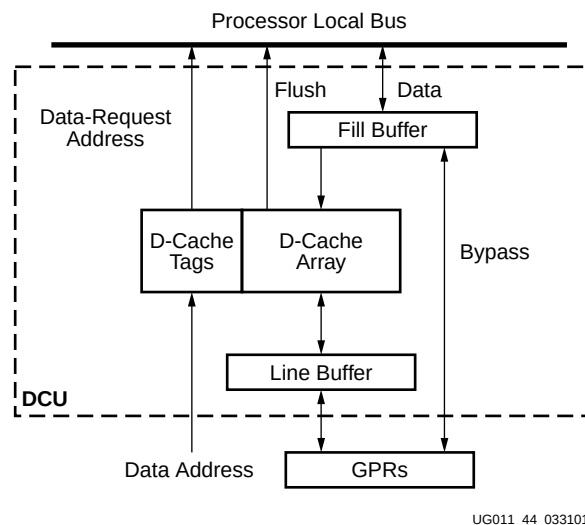


Figure 5-6: Data Flow to/from the Data-Cache Unit

All data-load requests and data-store requests are handled by the DCU. If a data address is cacheable, the DCU examines the data cache for a hit. A hit causes the cacheline to be read from the data cache and loaded into the line buffer. For a load hit, the data value is read from the line buffer and written to a GPR. For a store hit, the data value is read from the GPR and written to the line buffer and the line buffer is stored back into the data cache. The data cache supports byte writeability to improve the performance of byte and halfword stores. Load hits and store hits can be completed in one clock cycle.

If a cache miss occurs or if the data address is not cacheable, the DCU sends the data-address request to system memory over the processor local bus (PLB). Store misses to write-back memory and all load misses cause a cacheline fill. The size of all cacheline fill requests over the PLB is 32 bytes. The request size for a store to write-through memory (cache hit and cache miss) is one word (four bytes). The request size for a non-cacheable data access is programmable using the CCR0 register (see **Core-Configuration Register**, [page 495](#)). Cacheline fills are always completed (never aborted) even if the processor does not require any other bytes in the line. As data is received by the DCU from the PLB, it is placed in the fill buffer.

During a cacheline fill, the DCU requests the target data (load or store) first. However, the order data is returned depends on the design of the PLB device that handles the request (typically a memory controller). When the DCU receives target load data, it is forwarded immediately to the GPR over the bypass path. When the DCU receives target store data, it is immediately replaced by the GPR source value using the bypass path. The remaining data is received from the PLB and placed in the fill buffer. Subsequent loads and stores access the fill buffer if the data is present in the buffer. The data cacheline is loaded with the fill-buffer contents after all data are received.

If a cacheline fill replaces a dirty (modified) cacheline, the processor causes a *cacheline flush* to occur prior to loading the cacheline from the fill buffer. A cacheline flush updates system memory with the modified data from the cache. All 32 bytes in a cacheline are written sequentially to system memory over the PLB, including unmodified bytes.

Data Cacheability Control

Control of data cacheability depends on the address-translation mode:

- Real mode
- Virtual mode

Real Mode

In real mode, the data-cache cacheability register (DCCR) specifies which physical-memory regions are cacheable. See **Data-Cache Cacheability Register (DCCR)**, [page 490](#), for more information.

After a processor reset, the processor operates in real mode and all physical-memory regions are marked as non-cacheable (all DCCR bits are cleared to 0). Prior to specifying memory regions as cacheable, software must invalidate all data-cache congruence classes by executing the **dccci** instruction once for each class (see **Cache Instructions**, [page 492](#), for information on this instruction). After the congruence classes are invalidated, the DCCR can be configured.

Virtual Mode

In virtual mode, the storage-attribute fields in the page-translation look-aside buffer entry (TLB entry) specify which virtual-memory regions are cacheable. See **Storage-Attribute Fields**, [page 514](#), for more information.

Data-Cache Write Policy

Cacheable data can be written to the data cache using two write policies:

- Write-back caching
- Write-through caching

Write-Back Caching

In a *write-back* caching policy, the data cache is updated by a write hit but system memory is not updated. A write miss causes the cache to allocate a new cacheline and update that line—system memory is not updated.

Write-back caching can improve system performance by minimizing processor local bus activity. Write-back cachelines are only written to memory during cacheline replacement

or when explicitly flushed using a **dcbf** or **dcbst** instruction. Only modified cachelines are written.

Write-Through Caching

In a *write-through* caching policy, both the data cache and system memory are updated by a write hit. A write miss updates only system memory—a new cacheline is not allocated.

Write-through caching can simplify the work of maintaining coherency between the data cache and system memory. See **Software Management of Cache Coherency**, [page 499](#), for more information.

Control of the data-cache write policy depends on the address-translation mode:

- In real mode, the data-cache write-through register (DCWR) specifies the write policy for each physical-memory region. See **Data-Cache Write-Through Register (DCWR)**, [page 489](#), for more information.
- In virtual mode, the storage-attribute fields in the page-translation entry (TLB entry) specify the data-cache write policy for virtual-memory regions. See **Storage-Attribute Fields**, [page 514](#), for more information.

The write policy is in effect only when a memory region is defined as cacheable. Otherwise, it is ignored.

Data-Cache Allocation Control

Software can control data-cacheline allocation and data PLB-request size by using the core-configuration register 0 (CCR0):

- Load misses from cacheable memory can be prevented from allocating cachelines by using the load without allocate bit, CCR0[LWOA]. This can provide a performance advantage if memory reads are infrequent and tend to access non-contiguous addresses.
- Loads from non-cacheable memory (and those that do not allocate cachelines, as described above) can be programmed to generate eight-word PLB requests, or to generate only the number of data requested by the CPU. This is controlled using the load-word-as-line bit, CCR0[LWL]. If CCR0[LWL]=1, the DCU requests eight words. Using an eight-word request size provides the fastest access to sequential non-cacheable memory. The requested data remains in the data-cache fill buffer until one of the following occur:
 - A subsequent load replaces the contents of the fill buffer.
 - A store to an address contained in the fill buffer occurs.
 - A **dcbi** or **dccci** instruction is executed that affects an address in the fill buffer.
 - A **sync** instruction is executed.

Note that if CCR0[LWL]=1 and the target non-cacheable region is also marked as guarded (i.e., the G storage attribute is set to 1), the DCU will request only the data requested by the CPU.

- Store misses to cacheable memory can be prevented from allocating cachelines by using the store without allocate bit, CCR0[SWOA]. Software can use this bit to prevent a store miss to write-back memory from allocating a cacheline. Instead, the store updates system memory as if a write-through caching policy were in effect. Unlike write-through caching, store hits to write-back memory *do not* automatically update system memory when this bit is set.

See **Core-Configuration Register**, [page 495](#), for more information on these control bits.

Data-Cache Performance

In general, a data-cache hit completes in one cycle without stalling the processor. The DCU can perform certain cache operations in parallel to improve performance. Combinations of load and store operations—cacheline fills, cacheline flushes, and operations that hit in the

cache—can occur simultaneously. However, data-cache performance ultimately depends on software-execution dynamics and on the design of the external-memory controller. These two factors can combine to adversely affect data-cache performance by introducing pipeline stalls.

Pipeline Stalls

A *pipeline stall* occurs when instruction execution must wait for data to be loaded from or stored to memory. If the DCU can access the data immediately, no pipeline stall occurs. If the DCU cannot perform the access immediately, a pipeline stall can occur and continues until the DCU completes the access. The following events and operations can cause the DCU to stall the pipeline:

- A cache miss occurs or software accesses non-cacheable memory. This causes the DCU to retrieve data from system memory, which can take many cycles.
- The fill buffer contents (when full) are transferred to the data cache. During this time no other cache access can be performed. The process takes three cycles if the replaced cacheline is unmodified and four cycles if the replaced cacheline is modified.
- A load from non-cacheable memory is followed by other non-cacheable loads. The loads require at least four cycles to complete.
- More than two loads are pending completion in the DCU. The DCU can accept a second load if the first load cannot be completed immediately. If a subsequent DCU request of any kind is made, it is not accepted until the previous loads are completed by the DCU.
- A store to non-cacheable memory is followed by other non-cacheable stores. The stores require at least two cycles to complete.
- More than three stores are pending completion in the DCU. The DCU can accept a third store if the first two stores cannot be completed immediately. If a subsequent DCU request of any kind is made, it is not accepted until the previous stores are completed by the DCU.
- A data-cache control instruction (for example, **dcba** or **dcbst**) is executed. This causes a pipeline stall until all previous DCU operations complete execution, including loads and stores.
- More than two cacheline fills are pending.
- More than two cacheline flushes are pending.
- The on-chip memory (OCM) interface asserts a hold signal. The DCU can accept one additional load or store before causing a pipeline stall.

Data-Cache PLB Priority

The processor asserts a *data-cache to PLB priority* (DPP) signal when a PLB request is issued by the DCU. The DPP signal tells the PLB arbiter the priority that should be assigned to the DCU request. DPP is a two-bit signal. The high-order bit (DPP₀) is controlled by the DCU. The low-order bit (DPP₁) can be controlled by software using the DDP1 field in the CCR0 register. See [Table 5-6, page 496](#), for more information on using this CCR0 field.

Table 5-1 shows the conditions under which the DCU asserts and deasserts DPP₀. As is shown in the table, loads from system memory have highest priority and always immediately assert DPP₀.

Table 5-1: Data-Cache to PLB Priority Examples

If the Current DCU Operation...	...Has the Following DPP ₀ Value...	The Next DCU Operation...	...Updates DPP ₀ as Shown
Load from system memory.	Assert	See first column	
Store to system memory	Deassert	Any stalled DCU operation	Assert
dcbf		Cache hit	Deassert
dcbf, dcbst		Non-cacheable load	Assert
dcbf, dcbst		Cacheline flush	Assert
dcbt		Cache hit	Deassert
dcbi, dccci, dcbz	Deassert	See first column	

Data-Cache Hint Instructions

The PowerPC architecture defines data-cache instructions that can be used to improve memory performance by providing hints to the processor that memory locations are likely to be accessed in the near future. They are:

- *Data-cache block touch (dcbt)*—This instruction indicates that memory loads are likely to occur from the specified address. The processor can prefetch the cacheline associated with the address as a result of executing this instruction.
- *Data-cache block touch for store (dcbtst)*—This instruction indicates that memory stores are likely to occur to the specified address. The processor can prefetch the cacheline associated with the address as a result of executing this instruction.

Depending on how a processor implementation interacts with the memory subsystem, **dcbt** and **dcbst** can behave differently. On the PPC405, however, **dcbt** and **dcbtst** are implemented identically. These instructions execute as a no-operation if loading the cacheline were to result in a page-translation exception or a protection exception.

The following instructions can also be used as hint instructions when the contents of an address in system memory are not important:

- *Data-cache block allocate (dcba)*—This instruction allocates a cacheline corresponding to the specified address.
- *Data-cache block zero (dcbz)*—This instruction allocates a cacheline corresponding to the specified address and clears the cacheline contents to zero. It can be used to initialize cacheable memory locations.

dcba and **dcbz** do not access memory when allocating a cacheline. It is possible for these instructions to allocate cachelines for non-existent physical-memory addresses. A subsequent attempt to store the cacheline contents back to system memory can result in system problems or cause a machine-check exception to occur.

The **dcba** instruction executes as a no-operation if loading the cacheline were to result in a page-translation exception or a protection exception. On the other hand, **dcbz** causes a data-storage interrupt to occur if loading the cacheline results in a page-translation exception or a protection exception.

Accessing Memory

Memory (collectively, system memory and cache memory) is accessed when instructions are fetched and when a program executes load and store instructions. Other conditions not specified by a program can cause memory accesses to occur, such as cacheline fills and

cache flushes. The coherency and ordering of these memory accesses are influenced by the processor implementation, the memory system design, and software execution.

Memory Coherency

Coherency describes the ordering of reads from and writes to a single memory location. A memory system is *coherent* when the value read from a memory address is always the last value written to the address. In a system where all devices read and write from a single, shared system memory, memory is always coherent. In systems with memory-caching devices, maintaining coherency is less straightforward. For example, a processor cache can contain a more recent value for a memory location than system memory. The memory system is coherent only when a mechanism is provided to ensure a device receives the cached value rather than the system-memory value when read.

The PPC405 *does not* support memory-coherency management in hardware. Certain situations exist where coherency can be lost between system memory and the processor caches. On the PPC405, these situations require software management of memory coherency. See **Software Management of Cache Coherency**, [page 499](#), for more information.

Atomic Memory Access

An access is *atomic* if it is always performed in its entirety with no software-visible fragmentation. Only the following single-register accesses are guaranteed to be atomic:

- Byte accesses.
- Halfword accesses aligned on halfword boundaries.
- Word accesses aligned on word boundaries.

No other access is guaranteed to be atomic, particularly the following:

- Load and store operations using unaligned operands.
- Accesses resulting from execution of the **lww**, **stww**, **lswi**, **lswx**, **stswi**, or **stswx** instructions.
- Accesses resulting from execution of cache-management instructions.

The **lwarx**/**stwcx** instruction combination can be used to perform an atomic memory access. The **lwarx** instruction is a load from a word-aligned memory location that has two side effects:

- A reservation for a subsequent **stwcx** instruction is created.
- The memory coherence mechanism is notified that a reservation exists for the memory location accessed by the **lwarx**.

The **stwcx** instruction conditionally stores to a word-aligned memory location based on the existence of a reservation created by **lwarx**. See **Synchronizing Instructions**, [page 460](#), for more information on using these instructions.

Ordering Memory Accesses

The PowerPC architecture specifies a weakly-consistent memory model for shared-memory multiprocessor systems. The order a processor performs memory accesses, the order those accesses complete in memory, and the order those accesses are viewed as occurring by another processor can all differ. This model provides an opportunity for significantly improved performance over a model applying stronger consistency rules. However, the responsibility for memory-access ordering is placed on the programmer.

When a program requires strict access ordering for proper execution, the programmer must insert the appropriate ordering or synchronizing instructions into the program. The PowerPC architecture provides the ability to enforce memory-access ordering among multiple programs that share memory. Similar means are provided for programs that share memory with other hardware devices, such as I/O devices. These are:

- *Enforce in-order execution of I/O instruction*—The **eieio** instruction forces load and store memory-access ordering. The instruction acts as a barrier between all loads and stores that precede it and those that follow it. **eieio** can be used to ensure that a sequence of load and store operations to an I/O-device control register are performed in the desired order.
- *Synchronize instruction*—The **sync** instruction guarantees that all preceding coherent memory accesses initiated by a program appear to complete before the **sync** instruction completes. No subsequent instructions appear to execute until after the **sync** instruction completes.

On processors that support hardware-enforced shared-memory coherency, the **sync** instruction also provides synchronization *between* devices that access memory. The PPC405 does not provide hardware-enforced shared-memory coherency support. On the PPC405, the **sync** instruction is implemented identically to **eieio**.

In systems supporting hardware-enforced shared-memory coherency, **sync** can take significantly longer to execute than **eieio**. Programmers should avoid using **sync** when **eieio** performs the required ordering.

Preventing Inappropriate Speculative Accesses

PowerPC processors can perform speculative memory accesses, either to fetch instructions or to load data. A speculative access is any access not required by the sequential-execution model. For example, fetching instructions beyond an unresolved conditional branch is considered speculative. If the branch prediction is incorrect, the program (as executed) never requires the speculatively fetched instructions from the mispredicted path.

Sometimes speculative accesses are inappropriate. For example, an attempt to fetch instructions from addresses that do not contain instructions can cause a program to fail. Speculatively reading data from a memory-mapped I/O device can cause undesirable system behavior. Speculatively reading data from a peripheral status register that is cleared automatically after a read can cause unintentional loss of status information.

The PPC405 does not perform speculative data loads, but can speculatively fetch instructions. Branch prediction can cause speculative fetching of up to five cacheable instructions, or two non-cacheable instructions. If a **bctr** or **blr** instruction is predicted as taken, speculative fetching down the predicted path does not begin until all updates of the CTR or LR ahead of the predicted branch are complete. This prevents speculative accesses from unrelated addresses residing temporarily in the CTR and LR.

Using Guarded Storage

Speculative accesses can be prevented by assigning the guarded storage attribute (G) to memory locations (see **Guarded (G)**, [page 488](#)). An access to a guarded memory location is not performed until that access is required by the sequential-execution model and is no longer speculative. There is a considerable performance penalty associated with accessing guarded memory locations, so the guarded storage attribute should be used only when required.

Guarded storage can be specified in two ways, depending on the address-translation mode:

- In real mode (MSR[IR]=0), the storage-guarded register (SGR) controls assignment of the guarded attribute to memory locations.
- In virtual mode (MSR[IR]=1), the page-translation look-aside buffer (TLB) for a virtual-memory page contains a G field that controls assignment of the guarded attribute to memory locations.

Marking a memory location as guarded does not completely prevent speculative accesses from that memory location. Speculative accesses from guarded storage can occur in the following cases:

- Load instructions—If the memory location is already cached, the location can be speculatively accessed.
- Instruction fetch, real mode—If the instruction address is already cached, the instruction can be speculatively fetched. If the instruction address is required by the sequential-execution model and is in the same physical page or next physical page as the previous instruction, it can be speculatively fetched. A real-mode physical page is a contiguous 1 KB block of physical memory, aligned on a 1 KB address boundary.
- Instruction fetch, virtual mode—In virtual mode, attempts to fetch instructions either from guarded storage or from no-execute memory locations normally cause an instruction-storage interrupt to occur. However, the instruction can be cached prior to designating the address as guarded or no-execute. If the instruction address is present in the cache, the instruction can be speculatively fetched, even if it is later marked as guarded or no-execute.

Using Unconditional Branches

Speculative accesses can be prevented without using the guarded storage attribute. This is done by placing unconditional branches immediately before memory regions that should not be speculatively accessed. When an unconditional branch is fetched by the processor, it recognizes it as a break in program flow and knows that the sequential instructions following the branch are not executed. The processor does not speculatively fetch those instructions and instead fetches from the branch target. Placing unconditional branches at the end of physical memory and at addresses bordering I/O devices prevents speculative accesses from occurring outside the appropriate regions.

The system-call and interrupt-return instructions (**sc**, **rfi**, and **rfci**) are not recognized by the processor as breaks in program flow and speculative fetches can occur past those instructions. This can cause problems when one of the speculatively fetched instructions is a **bctr** or **blr**. For example:

```
handler: first instruction
more instructions
rfi
subroutine: bctr
```

The processor can speculatively fetch the **bctr** target, which is the first instruction of a subroutine unrelated to the interrupt handler. Here, the CTR might contain an invalid address. To prevent prefetching the **bctr**, software can insert an unconditional branch between the **rfi** and **bctr**. The branch can specify itself as the target to guarantee that only a valid instruction address is speculatively fetched.

Another example is one where a system-service routine is called to initialize the CTR with a branch-target address, as follows:

```
some instructions
sc
bctr
```

An unconditional branch cannot be inserted after the **sc** because the system-service routine returns to the instruction following **sc** when complete. Instead, software can use an **mtctr** instruction to initialize the CTR with a non-sensitive address prior to calling the service routine. Speculative fetches down the **bctr** path occur from the non-sensitive address. The **mtctr** also prevents speculative fetching until the processor updates CTR.

The system-trap instructions (**tw** and **twi**) do not require the special handling described above. These instructions are typically used by a debugger that sets breakpoints by replacing instructions with trap instructions. For example, in the sequence:

```
mtlr
blr
```

Replacing the **mtlr** above with **tw** or **twi** leaves the LR uninitialized. It would be inappropriate to prefetch from the **blr** target in this situation. The processor is designed to prevent speculative prefetching when executing the system-trap instructions.

Memory-System Control

Software manages memory-system operation using a combination of synchronization instructions (described in the previous section) and storage attributes. These resources provide program control over memory coherency, memory-access ordering, and speculative memory accesses

Storage Attributes

Storage attributes are used by system software to control how the processor accesses memory. These attributes are used to control cacheability, endianness (byte-ordering), and speculative accesses. PPC405 software can control five different storage attributes. Three attributes—write through (W), caching inhibited (I), and guarded (G)—are defined by the PowerPC architecture. Two attributes—user-defined (U0) and endian (E)—are defined by the PowerPC embedded environment architecture (the PowerPC Book-E architecture also supports these attributes).

The PowerPC architecture defines a memory-coherency attribute (M), but this attribute has no effect when used in PPC405 systems.

Management of storage attributes depends on whether address translation is used to access memory. In virtual mode, the page translation (TLB) entry for a virtual-memory region defines the storage attributes (see **Storage-Attribute Fields**, [page 514](#)). In real mode, the storage-attribute control registers are used to define the storage attributes (see **Storage-Attribute Control Registers**, [page 488](#)).

The following sections describe the function of each attribute.

Write Through (W)

The write-through storage attribute controls the caching policy of a memory region.

When the W attribute is cleared to 0, the memory region has a write-back caching policy. Writes that hit the cache update the cacheline but they do not update system memory. Writes that miss the cache allocate a new cacheline and update that line, but they do not update system memory.

When the W attribute is set to 1, the memory region has a write-through caching policy. Writes that hit the cache update both the cacheline and system memory. Writes that miss the cache update system memory and do not allocate a new cacheline.

Caching Inhibited (I)

The caching-inhibited storage attribute controls the cacheability of a memory region. The value of this attribute and its effect on memory depends on whether the memory access is performed in virtual mode or real mode.

In virtual mode, a memory region is cacheable when the I attribute is cleared to 0. When the I attribute is set to 1, the memory region is not cacheable. Non-cacheable memory accesses bypass the cache and access system memory. It is considered a programming error when a memory-access target is resident in the cache and the I attribute is set to 1. The result of such an access are undefined.

The interpretation of this attribute is reversed in real-mode, which uses the data-cache cacheability register (DCCR) and the instruction-cache cacheability register (ICCR). Here, setting I to 1 enables cacheability and clearing I to 0 disables cacheability. See **Storage-Attribute Control Registers**, [page 488](#), for more information.

Memory Coherency (M)

The memory-coherency storage attribute controls memory coherency in multiprocessor environments. Because the PPC405x3 core does not provide hardware support for multiprocessor memory coherency, setting or clearing the M storage attribute has no effect.

See **Software Management of Cache Coherency**, [page 499](#), for more information on memory coherency.

Guarded (G)

The guarded storage attribute controls speculative accesses into a memory region.

When the G attribute is cleared to 0, speculative accesses from the memory region can occur.

When the G attribute is set to 1, speculative memory accesses (instruction prefetches and data loads) are not permitted. The G storage attribute is typically used to protect memory-mapped I/O from improper access. An instruction fetch from a guarded region does not occur until all previous instructions have completed execution, guaranteeing that the access is not speculative. Prefetching is disabled for a guarded region. Performance is degraded significantly when executing out of guarded regions, and software should avoid unnecessarily marking instruction regions as guarded.

See **Preventing Inappropriate Speculative Accesses**, [page 485](#) for more information on guarded storage.

User Defined (U0)

The user-defined storage attribute controls implementation-dependent (processor and/or system) behavior of an access into a memory region. For example, some embedded-system implementations use the U0 attribute to identify memory regions containing compressed instructions. In those implementations, memory regions with U0=1 contain compressed instructions, and memory regions with U0=0 contain uncompressed instructions.

If desired, system software can cause an exception to occur when a data store is performed to U0 memory locations. This exception condition can be enabled using the U0-exception enable bit (U0XE) in the CCR0 register (see **Core-Configuration Register**, [page 495](#)). When CCR0[U0XE]=1, a store to memory locations with U0=1 cause a data-storage interrupt to occur. When CCR0[U0XE]=0, stores to U0 memory locations do not cause an exception. See **Data-Storage Interrupt (0x0300)**, [page 542](#) for information on identifying U0 exceptions.

If no U0 behavior is implemented by the embedded system, setting and clearing the U0 attribute has no effect on instruction fetches or data loads. However, the U0-exception enable can be used to trigger data-storage interrupts as described above whether the system defines U0 behavior.

Endian (E)

The endian attribute controls the byte ordering of accesses into a memory region.

When the E attribute is cleared to 0, memory accesses use big-endian byte ordering. When the E attribute is set to 1, memory accesses use little-endian byte ordering. See **Byte Ordering**, [page 385](#) for more information on big-endian and little-endian memory accesses.

Storage-Attribute Control Registers

The storage-attribute control registers specify the real-mode storage attributes. In virtual mode, these registers are ignored and storage attributes are taken from the page translation entries (TLB entries). See **Storage-Attribute Fields**, [page 514](#) for information on virtual-mode storage attributes.

The storage-attribute control-registers are 32-bit registers. Each bit is associated with a 128 MB memory region: bit 0 controls the lowest 128 MB region, bit 1 controls the next-lowest 128 MB region, and so on. Together, the 32 register bits provide storage control across the entire 4 GB physical-address space. The five most-significant effective-address bits (EA_{0:4}) are used to select a specific bit within the register. [Table 5-2](#) shows the address ranges associated with each register bit.

Table 5-2: Storage-Attribute Control-Register Address Ranges

Register Bit Indexed with EA _{0:4}	Address Range	Register Bit Indexed with EA _{0:4}	Address Range
0	0x0000_0000 to 0x07FF_FFFF	16	0x8000_0000 to 0x87FF_FFFF
1	0x0800_0000 to 0x0FFF_FFFF	17	0x8800_0000 to 0x8FFF_FFFF
2	0x1000_0000 to 0x17FF_FFFF	18	0x9000_0000 to 0x97FF_FFFF
3	0x1800_0000 to 0x1FFF_FFFF	19	0x9800_0000 to 0x9FFF_FFFF
4	0x2000_0000 to 0x27FF_FFFF	20	0xA000_0000 to 0xA7FF_FFFF
5	0x2800_0000 to 0x2FFF_FFFF	21	0xA800_0000 to 0xAFFF_FFFF
6	0x3000_0000 to 0x37FF_FFFF	22	0xB000_0000 to 0xB7FF_FFFF
7	0x3800_0000 to 0x3FFF_FFFF	23	0xB800_0000 to 0xBFFF_FFFF
8	0x4000_0000 to 0x47FF_FFFF	24	0xC000_0000 to 0xC7FF_FFFF
9	0x4800_0000 to 0x4FFF_FFFF	25	0xC800_0000 to 0xCFFF_FFFF
10	0x5000_0000 to 0x57FF_FFFF	26	0xD000_0000 to 0xD7FF_FFFF
11	0x5800_0000 to 0x5FFF_FFFF	27	0xD800_0000 to 0xDFFF_FFFF
12	0x6000_0000 to 0x67FF_FFFF	28	0xE000_0000 to 0xE7FF_FFFF
13	0x6800_0000 to 0x6FFF_FFFF	29	0xE800_0000 to 0xEFFF_FFFF
14	0x7000_0000 to 0x77FF_FFFF	30	0xF000_0000 to 0xF7FF_FFFF
15	0x7800_0000 to 0x7FFF_FFFF	31	0xF800_0000 to 0xFFFF_FFFF

The following sections describe the six storage-attribute control registers in the PPC405.

Data-Cache Write-Through Register (DCWR)

The data-cache write-through register (DCWR) specifies real-mode caching policy (the W storage attribute). Its format is shown in Figure 5-7. Each bit in the DCWR controls whether a physical-memory region (as shown in Table 5-2) has a write-back or write-through caching policy. This register controls only the data-cache caching policy. The caching policy is not applicable to the instruction cache because writes into the instruction-cache are not supported.

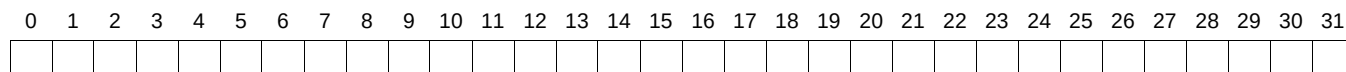


Figure 5-7: Data-Cache Write-Through Register (DCWR)

When a bit in the DCWR is cleared to 0, the specified memory region has a write-back caching policy. Writes that hit the cache update the cacheline but they do not update system memory. Writes that miss the cache allocate a new cacheline and update that line, but they do not update system memory. When the bit is set to 1, the specified memory region has a write-through caching policy. Writes that hit the cache update both the cacheline and system memory. Writes that miss the cache update system memory, but they do not allocate a new cacheline.

After a processor reset, all bits in the DCWR are cleared to 0. This establishes a write-back caching policy for all real-mode memory.

The DCWR is a privileged SPR with an address of 954 (0x3BA) and can be read and written using the **mfspr** and **mtspr** instructions.

Data-Cache Cacheability Register (DCCR)

The data-cache cacheability register (DCCR) specifies real-mode data-memory cacheability (the I storage attribute). Its format is shown in Figure 5-8. Each bit in the DCCR controls whether a physical-memory region (as shown in Table 5-2) is cacheable in the data cache.

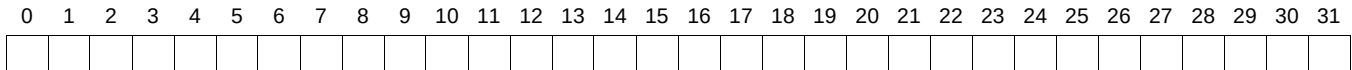


Figure 5-8: Data-Cache Cacheability Register (DCCR)

When a bit in the DCCR is cleared to 0, the specified memory region is not cacheable. Memory accesses bypass the data cache and access main memory. It is considered a programming error if a memory address is cached by the data cache when the corresponding bit in the DCCR is cleared to 0. The result of such an access are undefined. When the bit is set to 1, the specified memory region is cacheable, and its caching policy is governed by the DCWR register.

After a processor reset, all bits in the DCCR are cleared to 0, indicating that physical memory is not cacheable by the data cache. Prior to specifying memory regions as cacheable, software must invalidate all data-cache congruence classes by executing the **dccci** instruction once for each class (see **Cache Instructions**, page 492 for more information). After the congruence classes are invalidated, the DCCR can be configured.

The interpretation of the I attribute is reversed in virtual-mode when using page translations (TLB entries) to specify cacheability. See **Caching Inhibited (I)**, page 487 for more information.

The DCCR is a privileged SPR with an address of 1018 (0x3FA) and can be read and written using the **mfspr** and **mtspr** instructions.

Instruction-Cache Cacheability Register (ICCR)

The instruction-cache cacheability register (ICCR) specifies real-mode instruction-memory cacheability (the I storage attribute). Its format is shown in Figure 5-9. Each bit in the ICCR controls whether a physical-memory region (as shown in Table 5-2) is cacheable in the instruction cache.

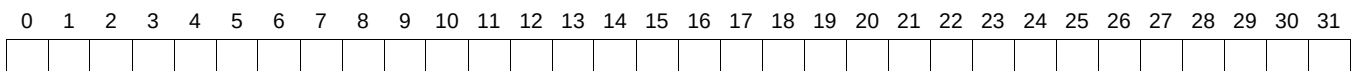


Figure 5-9: Instruction-Cache Cacheability Register (ICCR)

When a bit in the ICCR is cleared to 0, the specified memory region is not cacheable. Memory accesses bypass the instruction cache and access main memory. It is considered a programming error if a memory address is cached by the instruction cache when the corresponding bit in the ICCR is cleared to 0. The result of such an access are undefined. When the bit is set to 1, the specified memory region is cacheable.

After a processor reset, all bits in the ICCR are cleared to 0, indicating that physical memory is not cacheable by the instruction cache. Prior to specifying memory regions as cacheable, software must execute the **iccci** instruction, which invalidates the entire instruction cache (see **Cache Instructions**, page 492 for more information). After the cache is invalidated, the ICCR can be configured.

The polarity of the I attribute is opposite in virtual-mode when using page translations (TLB entries) to specify cacheability. See **Caching Inhibited (I)**, page 487 for more information.

The ICCR is a privileged SPR with an address of 1019 (0x3FB) and can be read and written using the **mfspr** and **mtspr** instructions.

Storage Guarded Register (SGR)

The storage guarded register (SGR) specifies guarded memory in real-mode (the G storage attribute). Its format is shown in [Figure 5-10](#). Each bit in the SGR controls whether a physical-memory region (as shown in [Table 5-2](#)) is guarded against speculative accesses. This register affects instruction memory only. Speculative loads are not performed on the PPC405, so guarding data memory has no effect. See **Preventing Inappropriate Speculative Accesses**, [page 485](#) for more information.

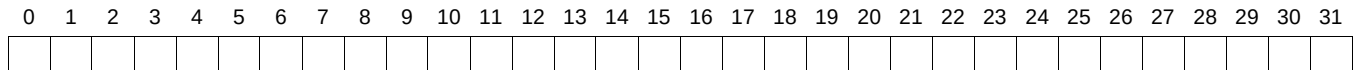


Figure 5-10: Storage Guarded Register (SGR)

When a bit in the SGR is cleared to 0, the specified memory region is not guarded and speculative accesses from the memory region can occur. When the bit is set to 1, the specified memory region is guarded and speculative accesses are not permitted.

After a processor reset, all bits in the SGR are set to 1. This establishes all of real-mode memory as guarded.

The SGR is a privileged SPR with an address of 953 (0x3B9) and can be read and written using the **mfsprr** and **mtsprr** instructions.

Storage User-Defined 0 Register (SU0R)

The storage user-defined 0 register (SU0R) specifies the implementation-dependent behavior of real-mode memory accesses (the U0 storage attribute). Its format is shown in [Figure 5-11](#). Some embedded-system implementations use the SU0R to identify physical memory regions (as shown in [Table 5-2](#)) containing compressed instructions. In those implementations, memory regions with U0=1 contain compressed instructions and memory regions with U0=0 contain uncompressed instructions.

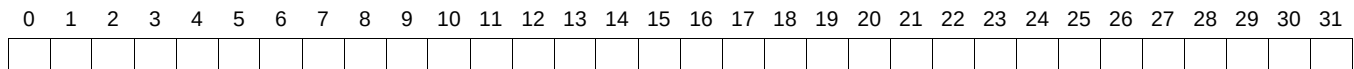


Figure 5-11: Storage User-Defined 0 Register (SU0R)

System software can use the U0 storage attribute to implement real-mode write protection. Writes to memory regions with U0=1 cause a data-storage exception if the U0 exception condition is enabled. This exception condition is enabled by setting the U0-exception enable bit (U0XE) in the CCR0 register to 1 (see **Core-Configuration Register**, [page 495](#)). When CCR0[U0XE]=0, writes to physical-memory locations do not cause an exception when the corresponding SU0R bit is set. See **Data-Storage Interrupt (0x0300)**, [page 542](#) for information on the U0 exception condition.

After a processor reset, all bits in the SU0R are cleared to 0.

The SU0R is a privileged SPR with an address of 956 (0x3BC) and can be read and written using the **mfsprr** and **mtsprr** instructions.

Storage Little-Endian Register (SLER)

The storage little-endian register (SLER) specifies the byte ordering for real-mode memory accesses (the E storage attribute). Its format is shown in [Figure 5-12](#). Each bit in the SLER controls whether a physical-memory region (as shown in [Table 5-2](#)) is accessed using big-endian or little-endian byte ordering. See **Byte Ordering**, [page 385](#) for more information on big-endian and little-endian memory accesses.

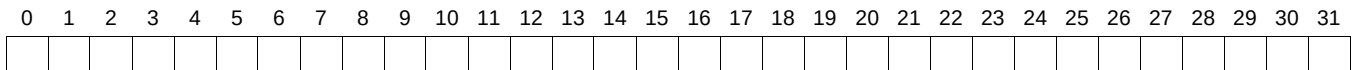


Figure 5-12: Storage Little-Endian Register (SLER)

When a bit in the SLER is cleared to 0, the specified memory region is accessed using big-endian ordering. When the bit is set to 1, the specified memory region is accessed using little-endian ordering.

After a processor reset, all bits in the SLER are cleared to 0. This specifies big-ending accesses for all real-mode memory.

The SLER is a privileged SPR with an address of 955 (0x3BB) and can be read and written using the **mf spr** and **mt spr** instructions.

Cache Control

Cache Instructions

The following sections describe the user and privileged instructions used in cache management. Within the instruction name, the term *cache block* often appears. A cache block is synonymous with a cacheline.

Table 5-3 summarizes which cache-control instructions are privileged and which instructions can be executed in user mode.

Table 5-3: Privileged and User Cache-Control Instructions

Instruction Cache		Data Cache	
Mnemonic	Privilege Level	Mnemonic	Privilege Level
icbi	User	dcba	User
icbt	User	dcbf	User
iccci	Privileged	dcbi	Privileged
icread	Privileged	dcbst	User
		dcbt	User
		dcbtst	User
		dcbz	User
		dccci	Privileged
		dcread	Privileged

Instruction-Cache Control Instructions

Table 5-4 shows the *instruction-cache control* instructions supported by the PPC405. These instructions provide the ability to invalidate the entire cache array or a single cacheline, prefetch instructions into the cache, and debug the cache.

Table 5-4: Instruction-Cache Control Instructions

Mnemonic	Name	Operation	Operand Syntax
icbi	Instruction Cache Block Invalidate	If the instruction specified by the effective address (EA) is cached by the instruction cache, the cacheline containing that instruction is invalidated. EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	rA,rB
icbt	Instruction Cache Block Touch	If the instruction specified by the effective address (EA) is cacheable and is not currently cached by the instruction cache, the cacheline containing that instruction is loaded into the instruction cache from system memory. EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	rA,rB
iccci	Instruction Cache Congruence Class Invalidate	Invalidates the entire instruction cache.	—
icread	Instruction Cache Read	If the instruction specified by the effective address (EA) is cached by the instruction cache, the ICDBDR register is loaded with information from one of the two ways indexed by the EA. CCR0 fields specify the cache way, and whether the instruction tag or instruction word is loaded into the ICDBDR. See icread Instruction , page 504 for more information. EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	rA,rB

Data-Cache Control Instructions

Table 5-5 shows the *data-cache control* instructions supported by the PPC405. These instructions provide the ability to invalidate the entire cache array or a single cacheline, prefetch data into the cache, and debug the cache.

Table 5-5: Data-Cache Control Instructions

Mnemonic	Name	Operation	Operand Syntax
dcba	Data Cache Block Allocate	An effective address (EA) is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$ This instruction can be used as a hint that a program might soon store into EA. It allocates a data cacheline for the byte addressed by EA. A subsequent store to EA hits the cache, improving program performance.	rA,rB
dcbf	Data Cache Block Flush	If the byte specified by the effective address (EA) is cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated. EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	rA,rB
dcbi	Data Cache Block Invalidate	If the byte specified by the effective address (EA) is cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), those modifications are lost. EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	rA,rB
dcbst	Data Cache Block Store	If the byte specified by the effective address (EA) is cached by the data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty). EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	rA,rB
dcbt	Data Cache Block Touch	If the byte specified by the effective address (EA) is cacheable and is not currently cached by the data cache, the cacheline containing that byte is loaded into the data cache from system memory. EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	rA,rB
dcbtst	Data Cache Block Touch for Store	If the byte specified by the effective address (EA) is cacheable and is not currently cached by the data cache, the cacheline containing that byte is loaded into the data cache from system memory. EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	rA,rB

Table 5-5: Data-Cache Control Instructions (*Continued*)

Mnemonic	Name	Operation	Operand Syntax
dcbz	Data Cache Block Clear to Zero	An effective address (EA) is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$ If the byte referenced by EA is not cached, a cacheline is allocated for that address. The cacheline containing the byte referenced by EA is cleared to 0 and marked modified (dirty). If the EA is non-cacheable or write-through, an alignment exception occurs. The alignment-interrupt handler can emulate the operation by clearing the corresponding bytes in system memory to 0.	rA,rB
dccci	Data Cache Congruence Class Invalidate	Invalidates both data-cache ways in the congruence class specified by the effective address (EA). Any modified data is lost. EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	rA,rB
dcread	Data Cache Read	If the byte specified by the effective address (EA) is cached by the data cache, rD is loaded with information from one of the two ways indexed by the EA. CCR0 fields specify the cache way and whether the data tag or data word is loaded into rD. See dcread Instruction, page 505 for more information. EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	rD,rA,rB

The **dcbt** and **dcbtst** instructions are implemented identically on the PPC405. On some processor implementations, these instructions can cause separate bus operations to occur that differentiate data-cache touches for loads from data-cache touches for stores.

dcbz establishes a cacheline without accessing system memory. It is possible for software to erroneously use this instruction to establish a cacheline for unimplemented memory locations. A subsequent access that attempts to update unimplemented system memory (such as a cacheline replacement) can cause unpredictable results or system failure.

Core-Configuration Register

The core-configuration register (CCR0) is a 32-bit register used to configure memory-system features, including:

- Whether cache misses cause cacheline allocation.
- Whether instruction prefetching is permitted.
- The size of non-cacheable requests over the processor local bus.
- The priority given by the processor when it makes a request over the processor local bus on behalf of a cache unit.
- Enablement of the U0 storage-attribute exception.
- Cache-debug features.

[Figure 5-13](#) shows the format of the CCR0. The fields in CCR0 are defined as shown in [Table 5-6](#).

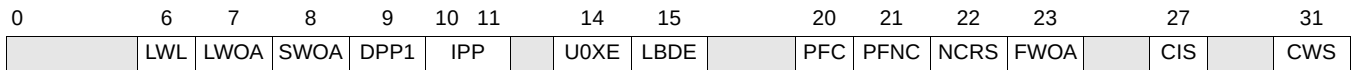


Figure 5-13: Core-Configuration Register (CCR0)

Table 5-6: Core-Configuration Register (CCR0) Field Definitions

Bit	Name	Function	Description
0:5		Reserved	
6	LWL	Load Word as Line 0—Load only requested data 1—Load entire cacheline	When this bit is set to 1, eight words are loaded into the fill buffer when a data-cache load-miss occurs, or when a load from non-cacheable memory occurs. The requested data is included in the eight words. When this bit is cleared to 0, only the requested data is loaded.
7	LWOA	Load Without Allocate 0—Allocate 1—Do not allocate	When this bit is set to 1, a load miss behaves like a non-cacheable load and does not allocate a data cacheline. When cleared to 0, load misses allocate a data cacheline.
8	SWOA	Store Without Allocate 0—Allocate 1—Do not allocate	When this bit is set to 1, a store miss behaves like a non-cacheable store and does not allocate a data cacheline. When cleared to 0, store misses to write-back memory allocate a data cacheline.
9	DPP1	DCU PLB-Priority Bit 1 0—DCU PLB priority 0 on bit 1 1—DCU PLB priority 1 on bit 1	Establishes the value of bit 1 in the 2-bit request-priority signal driven by the data-cache unit onto the processor local bus (PLB). Bit 0 is controlled by the processor and cannot be controlled by software. See PLB-Request Priority , page 497 for more information.
10:11	IPP	ICU PLB-Priority Bits 0:1 00—Lowest PLB req priority 01—Next-to-lowest priority 02—Next-to-highest priority 03—Highest PLB req priority	Establishes the value of the 2-bit request-priority signal driven by the instruction-cache unit onto the processor local bus (PLB). See PLB-Request Priority , page 497 for more information.
12:13		Reserved	
14	U0XE	Enable U0 Exception 0—Disabled 1—Enabled	Controls data-storage interrupts for memory with the U0 storage attribute set. A data-storage interrupt occurs when this bit is set to 1 and a store is performed to U0 memory. See Data-Storage Interrupt (0x0300) , page 542 for more information.
15	LBDE	Load-Debug Enable 0—Load data is not visible on the data-side OCM 1—Load data is visible on the data-side OCM.	
16:19		Reserved	
20	PFC	Prefetching for Cacheable Regions 0—Disabled. 1—Enabled.	When this bit is set to 1, the processor can prefetch instructions from cacheable memory regions into the instruction-prefetch buffers. Clearing this bit to 0 disables prefetching from cacheable memory regions, generally at a cost to performance.
21	PFNC	Prefetching for Non-Cacheable Regions 0—Disabled. 1—Enabled.	When this bit is set to 1, the processor can prefetch instructions from non-cacheable memory regions into the instruction-prefetch buffers. Clearing this bit to 0 disables prefetching from non-cacheable memory regions, generally at a cost to performance.

Table 5-6: Core-Configuration Register (CCR0) Field Definitions (Continued)

Bit	Name	Function	Description
22	NCRS	Non-Cacheable Request Size 0—Request size is four words. 1—Request size is eight words.	Specifies the number of instructions requested from non-cacheable memory when an instruction fetch or prefetch occurs. (Requests to cacheable memory are always eight words.)
23	FWOA	Fetch Without Allocate 0—Allocate. 1—Do not allocate.	When this bit is set to 1, an instruction-fetch miss behaves like a non-cacheable fetch and does allocate a data cacheline. When cleared to 0, fetch misses from cacheable memory allocate a data cacheline.
24:26		Reserved	
27	CIS	Cache-Information Select 0—Information is cache data. 1—Information is cache tag.	This bit is used by the dcread and icread instructions, and specifies whether cache-data or cache-tag information is loaded into the destination register. See Cache Debugging , page 504 for more information.
28:30		Reserved	
31	CWS	Cache-Way Select 0—Cache way is A. 1—Cache way is B.	This bit is used by the dcread and icread instructions, and identifies the cache way (A or B) from which the cache information specified by CCR0[CIS] is read. The information is loaded into the destination register. See Cache Debugging , page 504 for more information.

The CCR0 is a privileged SPR with an address of 947 (0x3B3) and can be read and written using the **mfspr** and **mtspr** instructions.

PLB-Request Priority

Table 5-7 shows the encoding of the 2-bit PLB-request priority signal. This signal is sent from a PLB master to a PLB arbiter indicating the priority of the master request. The arbiter uses these signals along with priority signals from other masters to determine which request should be granted. The PPC405 ICU and DCU are both PLB masters, and software can control their respective PLB-request priority using CCR0[IPP] and CCR0[DPP1].

Table 5-7: PLB-Request Priority Encoding

Bit 0	Bit 1	Definition
0	0	Lowest PLB-request priority.
0	1	Next-to-lowest PLB-request priority.
1	0	Next-to-highest PLB-request priority.
1	1	Highest PLB-request priority.

CCR0 Programming Guidelines

Several fields in CCR0 affect the instruction-cache and data-cache operation. Severe problems can occur—including a processor hang—if these fields are modified while the cache unit is involved in a PLB operation. To prevent problems, certain code sequences must be followed when modifying the CCR0 fields.

The first code example (Sequence 1) can be used to alter any field within CCR0. Use of this sequence is *required* when altering either CCR0[IPP] or CCR0[FWOA], both of which affect instruction-cache operation. In this and the following example, registers **rN**, **rM**, **rX**, and **rZ** are any available GPRs.

```

! SEQUENCE 1 - Required when altering CCR0[IPP, FWOA].
!
! Turn off interrupts.
mfmsr rM
addis rZ,r0,0x0002 ! CE bit
ori rZ,rZ,0x8000 ! EE bit
andc rZ,rM,rZ ! Turn off MSR[CE,EE]
mtmsr rZ
! Synchronize execution.
sync
! Touch the CCR0-altering function into the instruction cache.
addis rX,r0,seq1@h
ori rX,rX,seq1@l
icbt r0,rX

! Call the CCR0-altering function.
b seq1

back:
! Restore MSR to original value.
mtmsr rM
...

! The following function must be in cacheable memory so that it can be
touched into the instruction cache.

.align 5 ! Align the CCR0-altering function code on a cacheline
! boundary.

seq1:
! Repeat the instruction-cache touch and synchronize context to
! guarantee the most recent value of CCR0 is read. A total of eight
! instructions are touched into a single cacheline. This function
! example contains seven instructions. If more than eight instructions
! are required, additional lines must be touched into the cache.
icbt r0,rX
isync ! The CCR0-altering code has been completely
! fetched across the PLB.
mfmsr rN,CCR0 ! Read CCR0
! Use and/or instructions to modify any CCR0 bits. Because one cache
! line was touched in this example, up to two instructions can be used
! to modify CCR0.
andi/ori rN,rN,0xnnnn
mtspr CCR0,rN ! Update CCR0.
isync ! Refetch instructions under new processor context.
b back ! Branch back to initialization code.

```

The following code example (Sequence 2) can be used to alter either CCR0[DPP1] or CCR0[U0XE]. Sequence 1 can also be used to alter these fields.

```

! SEQUENCE 2 - Alter CCR0[DPP1, U0XE].
! Turn off interrupts.
mfmsr rM
addis rZ,r0,0x0002 ! CE bit
ori rZ,rZ,0x8000 ! EE bit
andc rZ,rM,rZ ! Turn off MSR[CE,EE]
mtmsr rZ
! Synchronize execution.
sync
! Modify CCR0.
mfmsr rN,CCR0 ! Read CCR0
! Use and/or instructions to modify any CCR0 bits.
andi/ori rN,rN,0xnnnn

```

```

mtspr CCR0,rN      ! Update CCR0.
isync              ! Refetch instructions under new processor context.
! Restore MSR to original value.
mtmsr rM

```

Modifications to CCR0[CIS] and CCR0[CWS] do not require special treatment.

Software Management of Cache Coherency

The PPC405 does not support memory-coherency management in hardware. This section describes the situations that can cause a loss of memory coherency and the steps software must take to prevent such loss.

How Coherency is Lost

Generally, coherency is lost when software shares cacheable memory with external devices. When a memory address is cached, the potential for losing memory coherency exists each time the address is accessed by any external device in the system. If a device reads cacheable system-memory, it can receive incorrect data. This occurs when modified data resides in write-back cachelines. Such data is not stored to system memory until the modified line is replaced by another line or until it is stored explicitly by a cache-control instruction. The use of write-through cachelines does not completely solve the problem. When an external device updates a cacheable system-memory location, copies present in the cache are not updated.

For example, when a DMA controller reads and writes cacheable system memory, coherency can be lost because:

- The processor does not automatically supply the DMA controller with the latest copy of data from the cache.
- The processor does not update cached locations with the latest copy written to system memory by the DMA controller.

To illustrate how coherency can be lost, consider the initial state of system memory and the contents of cache memory shown in the following table. For simplicity, the example uses a cacheline size of 16 bytes rather than 32 bytes. Each data element in the table represents a word (four bytes), although for clarity only byte values are shown. A row in the system-memory portion and cache-memory portion of the table each contain 16 data bytes. The “V” column indicates whether the cacheline is valid and the “D” column indicates whether the line data is dirty (modified). A “—” in the cache-memory portions indicates a don’t care.

System Memory					Cache Memory						
Address	Data (Words)				Address	V	D	Data (Words)			
1000	A9	2A	3A	EB	—	No	No	—	—	—	—
1010	0C	93	EE	A1	—	No	No	—	—	—	—
1020	EF	39	EB	A6	—	No	No	—	—	—	—
1030	3D	5F	8F	34	—	No	No	—	—	—	—

This example assumes write-back caching is enabled for all system-memory addresses represented in the above table (0x1000–0x103F). The following program is executed, updating the data words in addresses 0x1004–0x1030:

```

li    r1,0x1004-4    ! Start at address 0x1004.
li    r2,12           ! Fill 12 words.
mtctr r2              ! Initialize counter.
li    r3,0            ! Initialize data to zero.

```

```

loop:
stwu  r3,4(r1)      ! r1=r1+4, write (r3) to address in r1.
addi  r3,r3,1       ! Increment data (r3=r3+1).
bdnz  loop          ! Repeat until done.

```

As the program executes, cachelines are fetched from system memory into the cache and portions of the lines are overwritten with new data as specified by the program. The result is shown in the following table. Because the addresses are write-back cacheable, system memory is not updated. If an external device reads or writes the gray-shaded system-memory locations, a loss of coherency occurs. This can be prevented only if software flushes the affected lines from cache memory before the external device accesses system memory.

System Memory				
Address	Data (Words)			
1000	A9	2A	3A	EB
1010	0C	93	EE	A1
1020	EF	39	EB	A6
1030	3D	5F	8F	34

Cache Memory						
Address	V	D	Line Data (Words)			
1000	Yes	Yes	A9	00	01	02
1010	Yes	Yes	03	04	05	06
1020	Yes	Yes	07	08	09	0A
1030	Yes	Yes	0B	5F	8F	34

To further illustrate coherency loss, assume normal cache operations cause the first two cachelines to be replaced by unrelated data. Cacheline replacement updates system memory as shown below. Here, fewer system-memory locations are not coherent (shaded gray). An “x” indicates a replacement value in the cache unrelated to the program.

System Memory				
Address	Data (Words)			
1000	A9	00	01	02
1010	03	04	05	06
1020	EF	39	EB	A6
1030	3D	5F	8F	34

Cache Memory						
Address	V	D	Line Data (Words)			
x	Yes	x	x	x	x	x
x	Yes	x	x	x	x	x
1020	Yes	Yes	07	08	09	0A
1030	Yes	Yes	0B	5F	8F	34

Next, assume an external device updates the words at system-memory addresses 0x100C–0x1024, while at the same time a cacheline reload from 0x1010 occurs. This causes neither system memory nor the cache to contain data expected by the programmer (gray-shaded locations).

System Memory				
Address	Data (Words)			
1000	A9	00	01	FF
1010	FE	FD	FC	FB
1020	FA	F9	EB	A6
1030	3D	5F	8F	34

Cache Memory						
Address	V	D	Line Data (Words)			
x	Yes	x	x	x	x	x
1010	Yes	No	FE	FD	05	06
1020	Yes	Yes	07	08	09	0A
1030	Yes	Yes	0B	5F	8F	34

Coherency Loss Through Dual-Mapping

Some memory controllers support *dual-mapping* of physical-address ranges. With dual-mapping, two address ranges are resolved as a single address range. For example, assume

a memory controller is programmed to ignore the high-order physical-address bit (bit 0). Here, accesses to physical addresses 0x0000_0000 and 0x8000_0000 are resolved by the memory controller to the same physical address.

Software running on the PPC405 can specify address ranges as cacheable or non-cacheable using the cacheability registers (DCCR and ICCR) in real mode or using page translations in virtual mode. Using the above dual-mapping example, assume address 0x0000_0000 is cacheable and address 0x8000_0000 is non-cacheable. Software that reads data from address 0x0000_0000 does so using the cached copy, and reads from address 0x8000_0000 use the system-memory copy. Coherency is lost when the cached copy differs from the system-memory copy. To prevent this problem, dual-mapping should not be used to resolve cacheable address ranges and non-cacheable address ranges into a single address range.

Enforcing Coherency With Software

If a processor can cache shared-memory regions, access to those regions must be controlled by software. Software must ensure that addresses from a shared-memory region are not present in any of the processor caches before granting another device access to the region. Software must also avoid cacheable accesses into a shared-memory region until after the other device completes its access.

Cacheable accesses to non-shared-memory regions should not inadvertently cache information from adjacent, shared-memory regions. It is recommended that the alignment and size of shared-memory regions be a multiple of the cacheline size. By configuring all shared-memory regions to start on a cacheline boundary and span an integral number of cachelines, software can ensure that no cacheline contains a mixture of shared and non-shared memory.

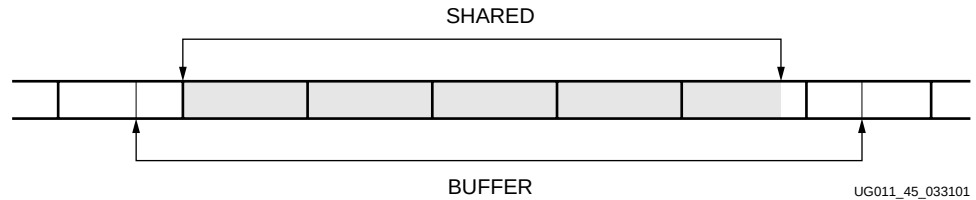
The instruction and data caches in the PPC405 have a cacheline size of 32 bytes. If a C program executing on a PPC405 requires 150 bytes of shared-buffer space, it should allocate the corresponding memory region as shown in the following programming example. In this example, *shared* represents the shared-memory region. However, system software controls the cacheability of *buffer* rather than *shared*.

```
#define LINE_LENGTH 32      ! Cacheline length in bytes.
#define BIT_MASK 0x1F      ! Address bits that select a byte in line.
char *buffer;              ! Buffer allocated by malloc.
char *shared;              ! Cacheline-aligned buffer.

! Obtain the buffer.
buffer = (char) malloc(150+2*LINE_LENGTH-2);

! If the buffer is not at the beginning of the cacheline,
! point to the start of the next cacheline.
if (buffer & BIT_MASK != 0)
    shared = buffer + LINE_LENGTH - (buffer & BIT_MASK);
else
    shared = buffer;        ! otherwise use as is
```

Figure 5-14 shows the placement of *buffer* and *shared* in memory after the above program is executed (cacheline boundaries are represented by heavy vertical lines). Because *malloc* does not necessarily allocate memory aligned on a cacheline boundary, the size of *buffer* is increased to account for alignment, and to span an integral number of cachelines. The second memory region, *shared*, is overlaid on *buffer*. The starting address of *shared* is adjusted to fall on the first cacheline boundary within *buffer*. The ending address of *shared* falls before a cacheline boundary, but that cacheline boundary falls within *buffer*.



UG011_45_033101

Figure 5-14: Example of Shared-Memory Allocation

Failure to allocate memory using this technique, or through compiler directives that align and pad variables in a similar manner, can cause coherency problems.

It is important that software control the cacheability of *buffer* when managing access to *shared*. The alignment and size of *buffer* is such that information in *shared* cannot be inadvertently cached by accesses to adjacent memory regions. If the cacheability of *shared* is managed instead, it is possible for data near the last address in *shared* to be cached inadvertently.

Cache Flushing

Before another device can access a shared-memory region, software must flush all shared-memory contents from the data cache. If the region contains executable code, all shared contents must be invalidated in the instruction cache. Data-cache flushing and instruction-cache invalidation are both required if software treats executable code as data (for example, moves executable code into or out of a shared-memory region). Invalidating shared-memory contents in the instruction cache keeps it coherent with system memory when executable code is relocated.

The method used to flush shared memory from the data cache depends on the size of the memory region relative to the data-cache size. Flushing shared memory address-by-address is most efficient when the region is smaller than the data cache. The following code sequence is an example of how shared-memory can be flushed from the data cache:

```
! r1 = start of shared-memory region.
! r2 = end of shared-memory region.
loop:
dcbf 0,r1                ! Flush cacheline at address r1.
addi r1,r1,32            ! Point to the next cacheline.
cmpw r1,r2               ! Check if finished.
ble  loop                ! If not, continue until done.
```

In the above example, the **dcbf** instruction invalidates all data cachelines containing shared-memory addresses. If a cacheline contains modified data, it is written back to system memory prior to invalidation. No action is taken if the cache does not contain addresses from the shared-memory region.

If the shared-memory region is larger than the data cache, flushing the entire data cache can often yield better performance than using the process shown above. However, the PPC405 does not provide a data-cache flush instruction. Instead, software must replace the data-cache contents, forcing writes of all modified lines to system memory.

The following code sequence uses the **dcbz** instruction in such a manner. **dcbz** can be used to establish a line in the data cache at an unused (and possibly non-existent) address without causing a load from system memory (and consuming PLB bandwidth). By executing two **dcbz** instructions using different addresses in the same congruence class, software can flush both cachelines in a set. Afterward, software can execute a **dccci** instruction to invalidate both of these new lines.

```
<Disable interrupts>
li  r1,<start of unused address range as large as data cache>
li  r2,16384                ! Cache size in bytes/2.
```



```

li    r3,256          ! Number of congruence classes in cache.
mtctr r3

loop:
dcbz  0,r1            ! Flush one way of the cache set.
dcbz  r2,r1           ! Flush the other way of the cache set.
dccci 0,r1            ! Invalidate the cache set.
addi  r1,r1,32        ! Point to the next cacheline.
bdnz  loop            ! Continue until all sets are flushed.
sync                      ! Ensure cache data has been written.

```

<Re-enable interrupts>

Interrupts are disabled during the flush procedure to prevent possible system-memory corruption occurring due to an unexpected system-memory access. These problems can arise if an interrupt occurs after a **dcbz** establishes a new cacheline but before the **dccci** invalidates that line. Executing the interrupt handler could cause a flush of the new line due to normal line replacement. This could corrupt system-memory or cause invalid memory accesses. Disabling interrupts eliminates the potential for unexpected cache activity.

Self-Modifying Code

Software that updates executable-memory locations is known as *self-modifying code*. If self-modifying code operates on cacheable-memory locations, cache-control instructions must be executed to maintain coherency between the instruction cache, system memory, and the data cache. Data-cache coherency is an issue because the instructions are treated as data when they are modified by other instructions.

Software that relocates executable code from one cacheable-memory location to another requires the same coherency treatment as self-modifying code. Although instructions are not changed, they are treated as data by the program that moves them, and can therefore be cached by the data cache.

The following code sequence can be used to enforce coherency between system memory and both the instruction and data caches. In this example, instructions are moved individually from one memory location to another while caching is enabled. Cache coherency is maintained throughout the process. Performance can be improved if software prohibits execution of the instructions while they are moved so that the caches are flushed and invalidated outside the loop.

```

! r1 = Instruction source address (word aligned).
! r2 = Instruction target address (word aligned).
! r3 = Number of instructions to move.
addi  r1,r1,-4        ! Initialize for use of lwzu and stwu
addi  r2,r2,-4
mtctr r3

loop:
lwzu  r4,4(r1)        ! Read source instruction.
stwu  r4,4(r2)        ! Write target instruction.
dcbf  0,r2            ! Remove target instruction from data cache.
icbi  0,r2            ! Remove target instruction from instruction cache.
bdnz  loop            ! Repeat until all instructions are moved.
sync                      ! Synchronize execution.
isync                   ! Synchronize context.

```

Coherency of self-modifying code can be maintained in a similar fashion. Instead of moving an instruction from one location to another, the source and target addresses are identical. A modifying instruction (or sequence of instructions) is inserted between the instruction load and instruction store. Below is a simple assembler-code sequence that can be used to maintain cache coherency during self-modifying code operations.


```

! rN contains a modified instruction.
stw    rN, addr1    ! Store the modified instruction.
dcbst  addr1        ! Force instruction to be written to system memory.
sync                   ! Wait for the system-memory update.
icbi   addr1        ! Invalidate unmodified instruction-cache entry.
isync                   ! The unmodified instruction might be in the
                        ! prefetch buffers. isync invalidates the prefetch
                        ! buffers.

```

Cache Debugging

The PPC405 provides two instructions that can read cache-tag and cache-data information for a specific cache congruence class. **icread** performs this function for the instruction cache and **dcread** performs this function for the data cache. These instructions operate under the control of certain bit fields in the CCR0 register (see **Core-Configuration Register**, page 495). The operation of each instruction is described in the following sections.

icread Instruction

The **icread** instruction reads instruction cacheline information for a specific effective address. A congruence class is selected from the instruction cache using the effective-address bits EA_{22:26}. A way is selected from the congruence class using the *cache-way select* field (CWS) in the CCR0 register. CCR0[CWS]=0 selects way A and CCR0[CWS]=1 selects way B. The cacheline information in the selected congruence-class and way is loaded into the 32-bit instruction-cache debug-data register (ICDBDR). Figure 5-15 shows the format of the ICDBDR. The fields in the ICDBDR are defined as shown in Table 5-8.

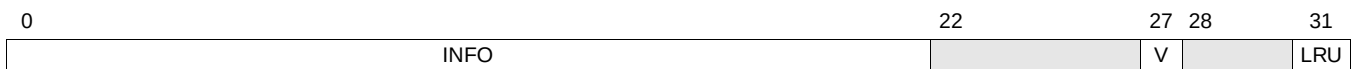


Figure 5-15: Instruction-Cache Debug-Data Register (ICDBDR)

Table 5-8: Instruction-Cache Debug-Data Register (ICDBDR) Field Definitions

Bit	Name	Function	Description
0:21	INFO	Instruction-Cache Information CCR0[CIS]=0—Instruction word. CCR0[CIS]=1—Instruction tag.	Contains either the cacheline tag or a single instruction word from the cacheline. If an instruction word is loaded, it is specified using effective-address bits EA _{27:29} . CCR0[CIS] controls the type of information loaded into this field.
22:26		Reserved	
27	V	Valid 0—Cacheline is not valid. 1—Cacheline is valid.	Contains a copy of the cacheline valid bit.
28:30		Reserved	
31	LRU	Least-Recently Used 0—Way A is least-recently used. 1—Way B is least-recently used.	Contains the LRU bit for the congruence class associated with the cacheline.

The ICDBDR is a privileged, read-only SPR with an address of 979 (0x3D3). It can be read using the **mfspr** instruction.

Synchronization is required between the **icread** instruction and the **mfspir** that reads the ICDBDR contents. This guarantees that the values read by **mfspir** are those loaded by the most-recent execution of **icread**. The following assembler-code sequence provides an example:

```
icread  rA, rB      ! Read instruction-cache information.
isync                    ! Ensure icread completes execution.
mficdbdr rD         ! Copy information to GPR.
```

dcread Instruction

The **dcread** instruction reads data cacheline information for a specific effective address. A congruence class is selected from the data cache using the effective-address bits EA_{19:26}. A way is selected from the congruence class using the *cache-way select* field (CWS) in the CCR0 register. CCR0[CWS]=0 selects way A and CCR0[CWS]=1 selects way B. The cacheline information in the selected congruence-class and way is loaded into the destination GPR, rD. **Figure 5-15** shows the format of the cache information loaded into rD. The information fields loaded in rD are defined as shown in **Table 5-8**.



Figure 5-16: Information Fields Loaded by dcread into rD

Table 5-9: dcread Information-Field Definitions

Bit	Name	Function	Description
0:18	INFO	Data-Cache Information CCR0[CIS]=0—Data word. CCR0[CIS]=1—Data tag.	Contains either the cacheline tag or a single data word from the cacheline. If a data word is loaded, it is specified using effective-address bits EA _{27:29} . CCR0[CIS] controls the type of information loaded into this field.
19:25		Reserved	
26	D	Dirty 0—Cacheline is not dirty. 1—Cacheline is dirty.	Contains a copy of the cacheline dirty bit, indicating whether the line contains modified data.
27	V	Valid 0—Cacheline is not valid. 1—Cacheline is valid.	Contains a copy of the cacheline valid bit.
28:30		Reserved	
31	LRU	Least-Recently Used 0—Way A is least-recently used. 1—Way B is least-recently used.	Contains the LRU bit for the congruence class associated with the cacheline.

Virtual-Memory Management

Programs running on the PPC405 use effective addresses to access a flat 4 GB address space. The processor can interpret this address space in one of two ways, depending on the translation mode:

- In *real mode*, effective addresses are used to directly access physical memory.
- In *virtual mode*, effective addresses are translated into physical addresses by the virtual-memory management hardware in the processor.

Virtual mode provides system software with the ability to relocate programs and data anywhere in the physical address space. System software can move inactive programs and data out of physical memory when space is required by active programs and data. Relocation can make it appear to a program that more memory exists than is actually implemented by the system. This frees the programmer from working within the limits imposed by the amount of physical memory present in a system. Programmers do not need to know which physical-memory addresses are assigned to other software processes and hardware devices. The addresses visible to programs are translated into the appropriate physical addresses by the processor.

Virtual mode provides greater control over memory protection. Blocks of memory as small as 1 KB can be individually protected from unauthorized access. Protection and relocation enable system software to support *multitasking*. This capability gives the appearance of simultaneous or near-simultaneous execution of multiple programs.

In the PPC405, virtual mode is implemented by the memory-management unit (MMU). The MMU controls effective-address to physical-address mapping and supports memory protection. Using these capabilities, system software can implement demand-paged virtual memory and other memory management schemes.

The MMU features are summarized as follows:

- Translates effective addresses into physical addresses.
- Controls page-level access during address translation.
- Provides additional virtual-mode protection control through the use of zones.
- Provides independent control over instruction-address and data-address translation and protection.
- Supports eight page sizes: 1 KB, 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, and 16 MB. Any combination of page sizes can be used by system software.
- Software controls the page-replacement strategy.

Real Mode

The processor references memory when it fetches an instruction and when it accesses data with a load, store, or cache-control instruction. Programs reference memory locations using a 32-bit effective address (EA) calculated by the processor based on the address mode (see **Effective-Address Calculation**, [page 380](#)). When real mode is enabled, the

physical address is identical to the effective address and the processor uses the EA to access physical memory. After a processor reset, the processor operates in real mode. Real mode can also be enabled independently for instruction fetches and data accesses by clearing the appropriate bits in the MSR:

- Clearing the *instruction-relocate* bit (MSR[IR]) to 0 disables instruction-address translation. Instruction fetches from physical memory are performed in real mode using the effective address.
- Clearing the *data-relocate* bit (MSR[DR]) to 0 disables data-address translation. Physical-memory data accesses (loads and stores) are performed in real mode using the effective address.

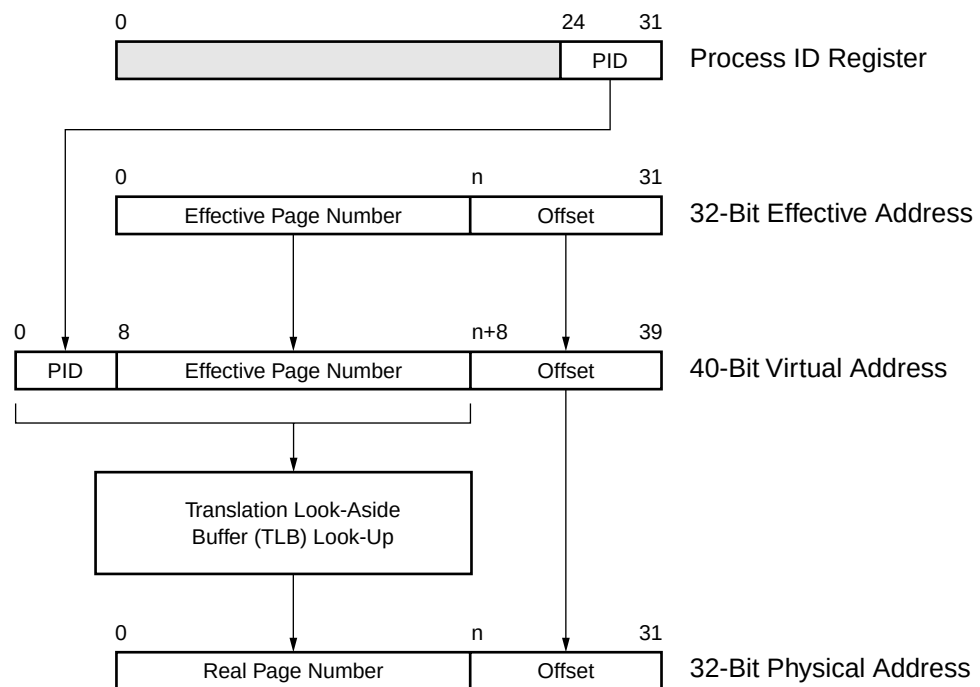
Real mode does not provide system software with the level of memory-management flexibility available in virtual mode. Storage attributes are associated with real-mode memory but access protection is limited (the U0 storage attribute can be used for write protection). Implementation of a real-mode memory manager is more straightforward than a virtual-mode memory manager. Real mode is often an appropriate solution for memory management in simple embedded environments.

See **Storage-Attribute Control Registers**, [page 488](#), for more information on real-mode memory control.

Virtual Mode

In virtual mode, the processor translates an EA into a physical address using the process shown in [Figure 6-1](#). Virtual mode can be enabled independently for instruction fetches and data accesses by setting the appropriate bits in the MSR:

- Setting the instruction-relocate bit (MSR[IR]) to 1 enables address translation (virtual mode) for instruction fetches.
- Setting the data-relocate bit (MSR[DR]) to 1 enables address translation (virtual mode) for data accesses (loads and stores).



UG011_37_021302

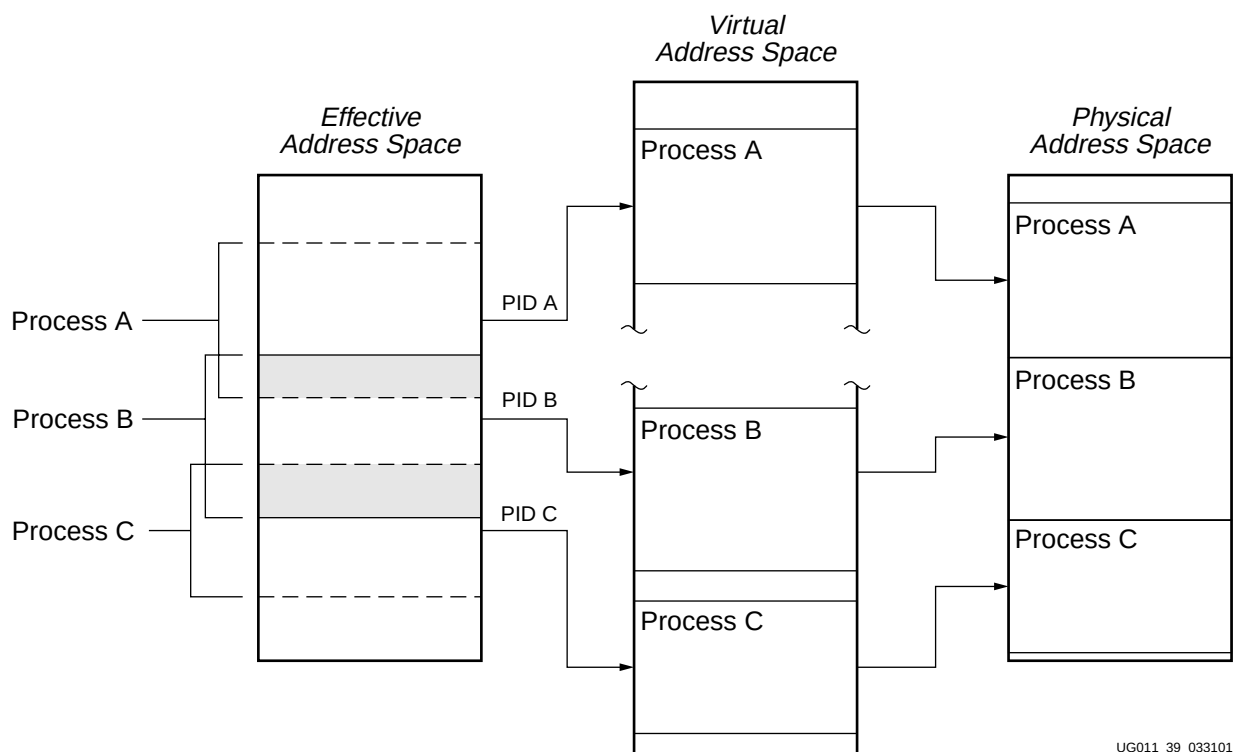
Figure 6-1: Virtual-Mode Address Translation

Each address shown in [Figure 6-1](#) contains a page-number field and an offset field. The page number represents the portion of the address translated by the MMU. The offset represents the byte offset into a page and is not translated by the MMU. The virtual address consists of an additional field, called the process ID (PID), which is taken from the PID register (see [Process-ID Register](#), [page 510](#)). The combination of PID and effective page number (EPN) is referred to as the virtual page number (VPN). The value n is determined by the page size, as shown in [Table 6-2](#), [page 514](#).

System software maintains a page-translation table that contains entries used to translate each virtual page into a physical page (see [page 510](#)). The page size defined by a page-translation entry determines the size of the page number and offset fields. For example, when a 4 KB page size is used, the page-number field is 20 bits and the offset field is 12 bits. The VPN in this case is 28 bits. See [Table 6-2](#), [page 514](#), for more information on page size.

Then the most frequently used page translations are stored in the translation look-aside buffer (TLB). When translating a virtual address, the MMU examines the page-translation entries for a matching VPN (PID and EPN). Rather than examining all entries in the table, only entries contained in the processor TLB are examined (see [page 511](#), for information on the TLB). When a page-translation entry is found with a matching VPN, the corresponding physical-page number is read from the entry and combined with the offset to form the 32-bit physical address. This physical address is used by the processor to reference memory.

System software can use the PID to uniquely identify software processes (tasks, subroutines, threads) running on the processor. Independently compiled processes can operate in effective-address regions that overlap each other. This overlap must be resolved by system software if multitasking is supported. Assigning a PID to each process enables system software to resolve the overlap by relocating each process into a unique region of virtual-address space. The virtual-address space mappings enable independent translation of each process into the physical-address space. [Figure 6-2](#) shows an example of how the PID is used in virtual-memory mapping (overlapping areas are shaded gray).



UG011_39_033101

Figure 6-2: Process-Mapping Example

Process-ID Register

The process-ID register (PID) is a 32-bit register used in virtual-address translation.

Figure 6-3 shows the format of the PID register. The fields in the PID are defined as shown in Table 6-1.



Figure 6-3: Process-ID Register (PID)

Table 6-1: Process-ID Register (PID) Field Definitions

Bit	Name	Function	Description
0:23		Reserved	
24:31	PID	Process Identifier	Used to uniquely identify a software process during address translation.

The PID is a privileged SPR with an address of 945 (0x3B1) and is read and written using the **mf spr** and **mt spr** instructions.

Page-Translation Table

The page-translation table is a software-defined and software-managed data structure containing page translations. The requirement for software-managed page translation represents an architectural trade-off targeted at embedded-system applications. Embedded systems tend to have a tightly controlled operating environment and a well-defined set of application software. That environment enables virtual-memory management to be optimized for each embedded system in the following ways:

- The *page-translation table* can be organized to maximize page-table search performance (also called *table walking*) so that a given page-translation entry is located quickly. Most general-purpose processors implement either an indexed page table (simple search method, large page-table size) or a hashed page table (complex search method, small page-table size). With software table walking, any hybrid organization can be employed that suits the particular embedded system. Both the page-table size and access time can be optimized.
- Independent *page sizes* can be used for application modules, device drivers, system-service routines, and data. Independent page-size selection enables system software to more efficiently use memory by reducing fragmentation (unused memory). For example, a large data structure can be allocated to a 16 MB page and a small I/O device-driver can be allocated to a 1 KB page.
- *Page replacement* can be tuned to minimize the occurrence of missing page-translations. As described in the following section, the most-frequently used page translations are stored in the translation look-aside buffer (TLB). Software is responsible for deciding which translations are stored in the TLB and which translations are replaced when a new translation is required. The replacement strategy can be tuned to avoid *thrashing*, whereby page-translation entries are constantly being moved in and out of the TLB. The replacement strategy can also be tuned to prevent replacement of critical-page translations, a process sometimes referred to as *page locking*.

The unified 64-entry TLB, managed by software, caches a subset of instruction and data page-translation entries accessible by the MMU. Software uses the unified TLB to cache a subset of instruction and data page-translation entries for use by the MMU. Software is

responsible for reading entries from the page-translation table in system memory and storing them in the TLB. The following section describes the unified TLB in more detail. Internally, the MMU also contains a 4-entry shadow TLB for instructions and an 8-entry shadow TLB for data. These shadow TLBs are managed entirely by the processor (transparent to software) and are used to minimize access conflicts with the unified TLB. **Figure 6-4** shows the relationship of the page-translation tables and the TLBs.

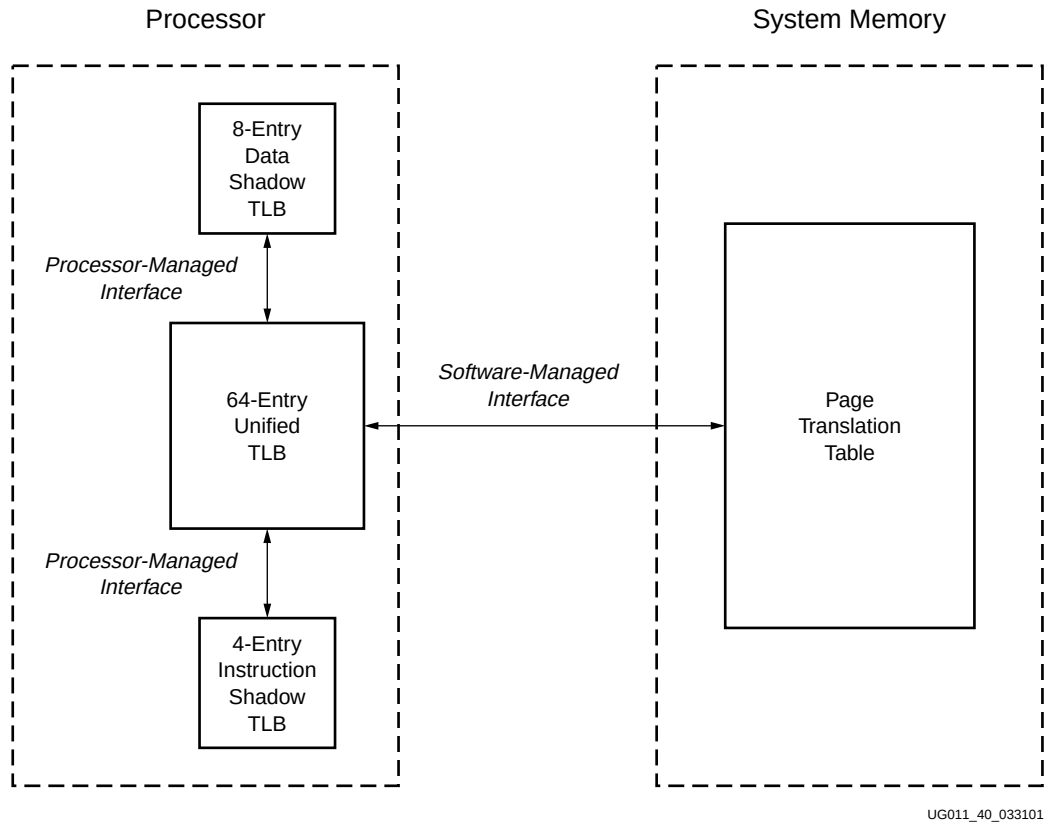


Figure 6-4: Page-Translation Table and TLB Organization

Translation Look-Aside Buffer

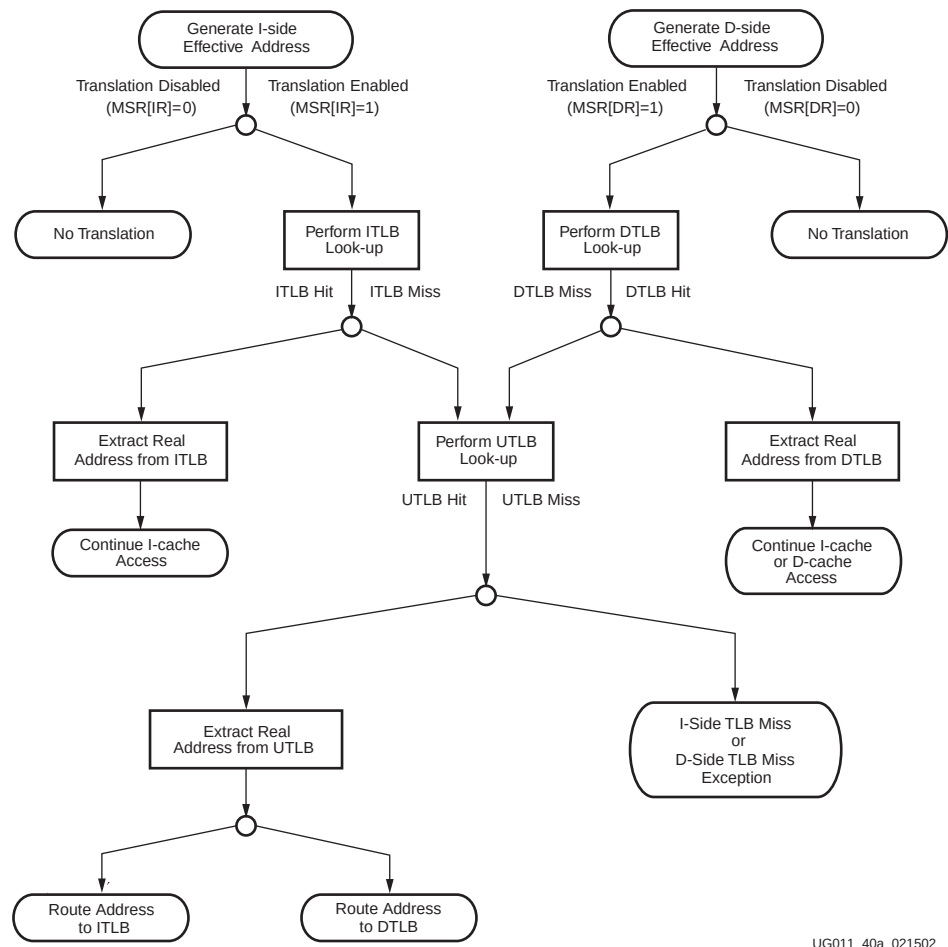
The translation look-aside buffer (TLB) is used by the PPC405 MMU for address translation, memory protection, and storage control when the processor is running in virtual mode. Each entry within the TLB contains the information necessary to identify a virtual page (PID and effective page number), specify its translation into a physical page, determine the protection characteristics of the page, and specify the storage attributes associated with the page.

The PPC405 TLB is physically implemented as three separate TLBs:

- **Unified TLB**—The UTLB contains 64 entries and is fully associative. Instruction-page and data-page translation can be stored in any UTLB entry. The initialization and management of the UTLB is controlled completely by software.
- **Instruction Shadow TLB**—The ITLB contains four instruction page-translation entries and is fully associative. The page-translation entries stored in the ITLB represent the four most-frequently accessed instruction-page translations from the UTLB. The ITLB is used to minimize contention between instruction translation and UTLB-update operations. The initialization and management of the ITLB is controlled completely by hardware and is transparent to software.

- **Data Shadow TLB**—The DTLB contains eight data page-translation entries and is fully associative. The page-translation entries stored in the DTLB represent the eight most-frequently accessed data-page translations from the UTLB. The DTLB is used to minimize contention between data translation and UTLB-update operations. The initialization and management of the DTLB is controlled completely by hardware and is transparent to software.

Figure 6-5 shows the address translation flow through the three TLBs.



UG011_40a_021502

Figure 6-5: ITLB/DTLB/UTLB Address Translation Flow

Although software is not responsible for managing the shadow TLBs, software must make sure the shadow TLBs are invalidated when the UTLB is updated. See **Maintaining Shadow-TLB Consistency**, page 523, for more information.

TLB Entries

Figure 6-6 shows the format of a TLB entry. Each TLB entry is 68 bits and is composed of two portions: TLBHI (also referred to as the *tag entry*), and TLBLO (also referred to as the *data entry*). The fields within a TLB entry are categorized as follows:

- **Virtual-page identification**—These fields identify the page-translation entry. They are compared with the virtual-page number during the translation process.
- **Physical-page identification**—These fields identify the translated page in physical memory.
- **Access control**—These fields specify the type of access allowed in the page and are used to protect pages from improper accesses.

- *Storage attributes*—These fields specify the storage-control attributes, such as whether a page is cacheable and how bytes are ordered (endianness).

The following sections describe the fields within each category.

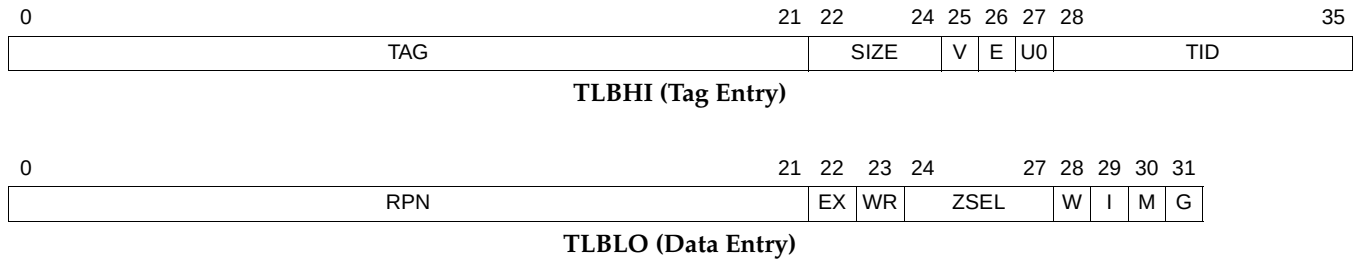


Figure 6-6: TLB-Entry Format

Virtual-Page Identification Fields

The virtual-page identification portion of a TLB entry contains the following fields:

- **TAG** (TLB-entry tag)—TLBHI, bits 0:21. This field is compared with the EPN portion of the EA (EA[EPN]) under the control of the SIZE field. [Table 6-2, page 514](#), shows the bit ranges used in comparing the TAG with EA[EPN]. In this table, TAG_{x:y} represents the bit range from the TAG field in TLBHI and EA_{x:y} represents the bit range from EA[EPN].
- **SIZE** (Page size)—TLBHI, bits 22:24. This field specifies the page size as shown in [Table 6-2, page 514](#). The SIZE field controls the bit range used in comparing the TAG field with EA[EPN].
- **V** (Valid)—TLBHI, bit 25. When this bit is set to 1, the TLB entry is valid and contains a page-translation entry. When cleared to 0, the TLB entry is invalid.
- **TID** (Process Tag)—TLBHI, bits 28:35. This 8-bit field is compared with the PID field in the process-ID register. When TID is clear (0x00), the field is ignored and not compared with the PID field. A clear TID indicates the TLB entry is used by all processes.

Physical-Page Identification Fields

The physical-page identification portion of a TLB entry contains the following field:

- **RPN** (Physical-page number, or real-page number)—TLBLO, bits 0:21. When a TLB hit occurs, this field is read from the TLB entry and is used to form the physical address. Depending on the value of the SIZE field, some of the RPN bits are not used in the physical address. *Software must clear unused bits in this field to 0.* See [Table 6-2, page 514](#), for information on which bits must be cleared.

Access-Control Fields

The access-control portion of a TLB entry contains the following fields:

- **EX** (Executable)—TLBLO, bit 22. When this bit is set to 1, the page contains executable code and instructions can be fetched from the page. When this bit is cleared to 0, instructions cannot be fetched from the page. Attempts to fetch instructions from a page with a clear EX bit cause an instruction-storage exception.
- **WR** (Writable)—TLBLO, bit 23. When this bit is set to 1, the page is writable and store instructions can be used to store data at addresses within the page. When this bit is cleared to 0, the page is read only (not writable). Attempts to store data into a page with a clear WR bit cause a data-storage exception.
- **ZSEL** (Zone select)—TLBLO, bits 24:27. This field selects one of 16 zone fields (Z0–Z15) from the zone-protection register (ZPR). For example, if ZSEL=0b0101, zone field

Z5 is selected. The selected ZPR field is used to modify the access protection specified by the TLB entry EX and WR fields. It is also used to prevent access to a page by overriding the TLB V (valid) field. See **Zone Protection**, [page 518](#), for more information.

Storage-Attribute Fields

The storage-attribute portion of a TLB entry contains the following fields:

- **E (Endian)**—TLBHI, bit 26. When this bit is set to 1, the page is accessed as a little-endian page. When cleared to 0, the page is accessed as a big-endian page. See **Byte Ordering**, [page 385](#), for information on little-endian and big-endian byte accesses.
- **U0 (User defined)**—TLBHI, bit 27. When this bit is set to 1, access to the page is governed by a user-defined storage attribute. When cleared to 0, the user-defined storage attribute does not govern accesses to the page. See **User Defined (U0)**, [page 488](#), for more information.
- **W (Write Through)**—TLBLO, bit 28. When this bit is set to 1, accesses to the page are cached using a write-through caching policy. When cleared to 0, accesses to the page are cached using a write-back caching policy. See **Write Through (W)**, [page 487](#), for more information.
- **I (Caching inhibited)**—TLBLO, bit 29. When this bit is set to 1, accesses to the page are not cached (caching is inhibited). When cleared to 0, accesses to the page are cacheable, under the control of the W attribute (write-through caching policy). See **Caching Inhibited (I)**, [page 487](#), for more information.
- **M (Memory coherent)**—TLBLO, bit 30. Setting and clearing this bit does not affect memory accesses in the PPC405. In implementations that support multi-processing, this bit can be used to improve the performance of hardware that manages memory coherency.
- **G (Guarded)**—TLBLO, bit 31. When this bit is set to 1, speculative page accesses are not allowed (memory is guarded). When cleared to 0, speculative page accesses are allowed. The G attribute is often used to protect memory-mapped I/O devices from inappropriate accesses. See **Guarded (G)**, [page 488](#), for more information.

In real mode, the storage-attribute control registers are used to define storage attributes. See **Storage-Attribute Control Registers**, [page 488](#) for more information.

Table 6-2 shows the relationship between the TLB-entry SIZE field and the translated page size. This table also shows how the page size determines which address bits are involved in a tag comparison, which address bits are used as a page offset, and which bits in the physical page number are used in the physical address. The final column, “*n*”, refers to a bit position shown in [Figure 6-1](#), [page 508](#).

When assigning sizes to instruction pages, software must be careful to avoid creating the opportunity for instruction-cache synonyms. See **Instruction-Cache Synonyms**, [page 478](#), for more information.

Table 6-2: Page-Translation Bit Ranges by Page Size

Page Size	SIZE (TLB Field)	Tag Comparison Bit Range	Page Offset	Physical-Page Number	RPN Bits Clear to 0	<i>n</i> (Figure 6-1)
1 KB	0b000	TAG _{0:21} ↔ EA _{0:21}	EA _{22:31}	RPN _{0:21}	—	22
4 KB	0b001	TAG _{0:19} ↔ EA _{0:19}	EA _{20:31}	RPN _{0:19}	20:21	20
16 KB	0b010	TAG _{0:17} ↔ EA _{0:17}	EA _{18:31}	RPN _{0:17}	18:21	18
64 KB	0b011	TAG _{0:15} ↔ EA _{0:15}	EA _{16:31}	RPN _{0:15}	16:21	16
256 KB	0b100	TAG _{0:13} ↔ EA _{0:13}	EA _{14:31}	RPN _{0:13}	14:21	14

Table 6-2: Page-Translation Bit Ranges by Page Size

Page Size	SIZE (TLB Field)	Tag Comparison Bit Range	Page Offset	Physical-Page Number	RPN Bits Clear to 0	<i>n</i> (Figure 6-1)
1 MB	0b101	TAG _{0:11} ↔ EA _{0:11}	EA _{12:31}	RPN _{0:11}	12:21	12
4 MB	0b110	TAG _{0:9} ↔ EA _{0:9}	EA _{10:31}	RPN _{0:9}	10:21	10
16 MB	0b111	TAG _{0:7} ↔ EA _{0:7}	EA _{8:31}	RPN _{0:7}	8:21	8

TLB Access

When the MMU translates a virtual address (the combination of PID and effective address) into a physical address, it first examines the appropriate shadow TLB for the page-translation entry. If an entry is found, it is used to access physical memory. If an entry is not found, the MMU examines the UTLB for the entry. A delay occurs each time the UTLB must be accessed due to a shadow TLB miss. For the ITLB, the miss latency is four cycles. The DTLB has a miss latency of three cycles. The DTLB has priority over the ITLB if both simultaneously access the UTLB.

Figure 6-7 shows the logical process the MMU follows when examining a page-translation entry in one of the shadow TLBs or the UTLB. All valid entries in the TLB are checked. In the PPC405, all entries in a specific TLB (shadow or unified) are examined simultaneously. A *TLB hit* occurs when all of the following conditions are met by a TLB entry:

- The entry is valid.
- The TAG field in the entry matches the EA[EPN] under the control of the SIZE field in the entry.
- The TID field in the entry matches the PID.

If any of the above conditions are not met, a *TLB miss* occurs. A TLB miss causes an exception, as described in **TLB-Access Failures**, page 516.

A TID value of 0x00 causes the MMU to ignore the comparison between the TID and PID. Only the TAG and EA[EPN] are compared. A TLB entry with TID=0x00 represents a process-independent translation. Pages that are accessed globally by all processes should be assigned a TID value of 0x00.

A PID value of 0x00 *does not* identify a process that can access any page. When PID=0x00, a page-translation hit only occurs when TID=0x00.

It is possible for software to load the TLB with multiple entries that match an EA[EPN] and PID combination. However, this is considered a programming error and results in undefined behavior.

When a hit occurs, the MMU reads the RPN field from the corresponding TLB entry. Some or all of the bits in this field are used, depending on the value of the SIZE field (see **Table 6-2**, page 514). For example, if the SIZE field specifies a 256 KB page size, RPN_{0:13} represents the physical page number and is used to form the physical address. RPN_{14:21} is not used, and software *must* clear those bits to 0 when initializing the TLB entry. The remainder of the physical address is taken from the page-offset portion of the EA. If the page size is 256 KB, the 32-bit physical address is formed by concatenating RPN_{0:13} with EA_{14:31}.

Prior to accessing physical memory, the MMU examines the TLB-entry access-control fields. These fields indicate whether the currently executing program is allowed to perform the requested memory access. See **Virtual-Mode Access Protection**, page 518, for more information.

If access is allowed, the MMU checks the storage-attribute fields to determine how to access the page. The storage-attribute fields specify the caching policy and byte ordering for memory accesses. See **Storage-Attribute Fields**, page 514, for more information.

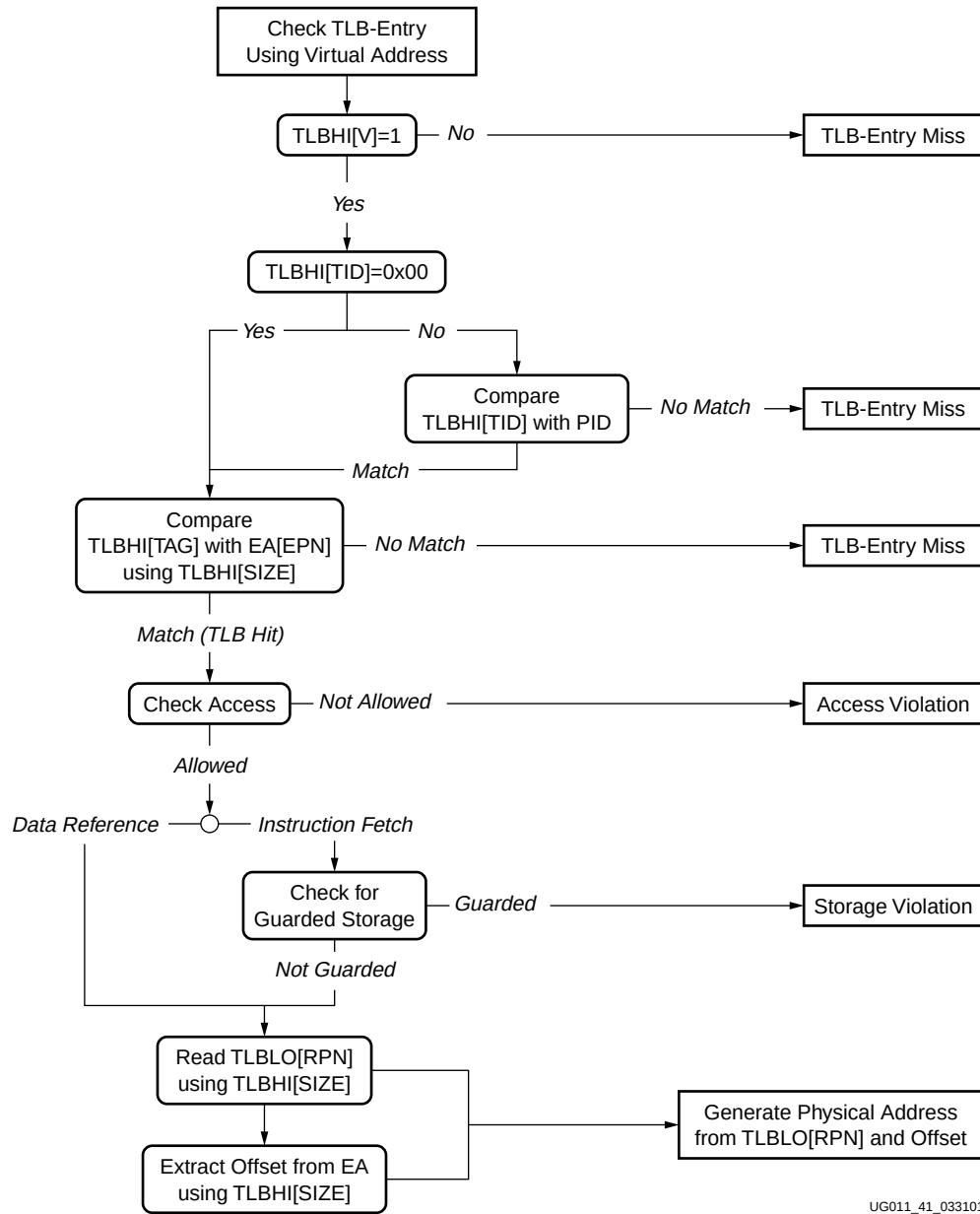


Figure 6-7: General Process for Examining a TLB Entry

TLB-Access Failures

A TLB-access failure causes an exception to occur. This interrupts execution of the instruction that caused the failure and transfers control to an interrupt handler to resolve the failure. A TLB access can fail for two reasons:

- A matching TLB entry was not found, resulting in a TLB miss.
- A matching TLB entry was found, but access to the page was prevented by either the storage attributes or zone protection.

When an interrupt occurs, the processor enters real mode by clearing MSR[IR, DR] to 0. In real mode, all address translation and memory-protection checks performed by the MMU are disabled. After system software initializes the UTLB with page-translation entries, management of the PPC405 UTLB is usually performed using interrupt handlers running in real mode.

The following sections describe the conditions under which exceptions occur due to TLB-access failures.

Data-Storage Exception

When data-address translation is enabled (MSR[DR]=1), a data-storage exception occurs when access to a page is not permitted for any of the following reasons:

- From user mode:
 - The TLB entry specifies a zone field that prevents access to the page (ZPR[Zn]=00). This applies to load, store, **dcbf**, **dcbst**, **dcbz**, and **icbi** instructions.
 - The TLB entry specifies a read-only page (TLBLO[WR]=0) that is not otherwise overridden by the zone field (ZPR[Zn]≠ 11). This applies to store and **dcbz** instructions.
 - The TLB entry specifies a U0 page (TLBHI[U0]=1) and U0 exceptions are enabled (CCR0[U0XE]=1). This applies to store and **dcbz** instructions.
- From privileged mode:
 - The TLB entry specifies a read-only page (TLBLO[WR]=0) that is not otherwise overridden by the zone field (ZPR[Zn]≠ 10 and ZPR[Zn]≠ 11). This applies to store, **dcbi**, **dcbz**, and **dccci** instructions.
 - The TLB entry specifies a U0 page (TLBHI[U0]=1) and U0 exceptions are enabled (CCR0[U0XE]=1). This applies to store, **dcbi**, **dcbz**, and **dccci** instructions.

See **Data-Storage Interrupt (0x0300)**, [page 542](#), for more information on this exception and **Zone Protection**, [page 518](#), for more information on zone protection.

Instruction-Storage Exception

When instruction-address translation is enabled (MSR[IR]=1), an instruction-storage exception occurs when access to a page is not permitted for any of the following reasons:

- From user mode:
 - The TLB entry specifies a zone field that prevents access to the page (ZPR[Zn]=00).
 - The TLB entry specifies a non-executable page (TLBLO[EX]=0) that is not otherwise overridden by the zone field (ZPR[Zn]≠ 11).
 - The TLB entry specifies a guarded-storage page (TLBLO[G]=1).
- From privileged mode:
 - The TLB entry specifies a non-executable page (TLBLO[EX]=0) that is not otherwise overridden by the zone field (ZPR[Zn]≠ 10 and ZPR[Zn]≠ 11).
 - The TLB entry specifies a guarded-storage page (TLBLO[G]=1).

See **Instruction-Storage Interrupt (0x0400)**, [page 544](#), for more information on this exception, **Guarded (G)**, [page 488](#), for more information on guarded storage, and **Zone Protection**, [page 518](#), for more information on zone protection.

Data TLB-Miss Exception

When data-address translation is enabled (MSR[DR]=1), a data TLB-miss exception occurs if a valid, matching TLB entry was not found in the TLB (shadow and UTLB). Any load, store, or cache instruction (excluding cache-touch instructions) can cause a data TLB-miss exception. See **Data TLB-Miss Interrupt (0x1100)**, [page 555](#), for more information.

Instruction TLB-Miss Exception

When instruction-address translation is enabled (MSR[IR]=1), an instruction TLB-miss exception occurs if a valid, matching TLB entry was not found in the TLB (shadow and UTLB). Any instruction fetch can cause an instruction TLB-miss exception. See **Instruction TLB-Miss Interrupt (0x1200)**, [page 556](#), for more information.

Virtual-Mode Access Protection

System software uses access protection to protect sensitive memory locations from improper access. System software can restrict memory accesses for both user-mode and privileged-mode software. Restrictions can be placed on reads, writes, and instruction fetches. Access protection is available only when instruction or data address translation is enabled.

Virtual-mode access control applies to instruction fetches, data loads, data stores, and cache operations. The TLB entry for a virtual page specifies the type of access allowed to the page. The TLB entry also specifies a zone-protection field in the zone-protection register that is used to override the access controls specified by the TLB entry.

TLB Access-Protection Controls

Each TLB entry controls three types of access:

- *Process*—Processes are protected from unauthorized access by assigning a unique process ID (PID) to each process. When system software starts a user-mode application, it loads the PID for that application into the PID register. As the application executes, memory addresses are translated using only TLB entries with a TLBHI[TID] field that matches the PID. This enables system software to restrict accesses for an application to a specific area in virtual memory.

A TLB entry with TID=0x00 represents a process-independent translation. Pages that are accessed globally by all processes should be assigned a TID value of 0x00.

- *Execution*—The processor executes instructions only if they are fetched from a virtual page marked as executable (TLBLO[EX]=1). Clearing TLBLO[EX] to 0 prevents execution of instructions fetched from a page, instead causing an instruction-storage interrupt (ISI) to occur. The ISI does not occur when the instruction is fetched, but instead occurs when the instruction is executed. This prevents speculatively fetched instructions that are later discarded (rather than executed) from causing an ISI.

The zone-protection register can override execution protection.

- *Read/Write*—Data is written only to virtual pages marked as writable (TLBLO[WR]=1). Clearing TLBLO[WR] to 0 marks a page as read-only. An attempt to write to a read-only page causes a data-storage interrupt (DSI) to occur.

The zone-protection register can override write protection.

TLB entries cannot be used to prevent programs from reading pages. In virtual mode, zone protection is used to read-protect pages. This is done by defining a *no-access-allowed* zone (ZPR[Zn] = 00) and using it to override the TLB-entry access protection. Only programs running in user mode can be prevented from reading a page. Privileged programs always have read access to a page. See **Zone Protection** below.

Zone Protection

Zone protection is used to override the access protection specified in a TLB entry. Zones are an arbitrary grouping of virtual pages with common access protection. Zones can contain any number of pages specifying any combination of page sizes. There is no requirement for a zone to contain adjacent pages.

The zone-protection register (ZPR) is a 32-bit register used to specify the type of protection override applied to each of 16 possible zones. The protection override for a zone is encoded in the ZPR as a 2-bit field. The 4-bit zone-select field in a TLB entry (TLBLO[ZSEL]) selects one of the 16 zone fields from the ZPR (Z0–Z15). For example, zone Z5 is selected when ZSEL = 0b0101.

Changing a zone field in the ZPR applies a protection override across all pages in that zone. Without the ZPR, protection changes require individual alterations to each page-translation entry within the zone.

Figure 6-8 shows the format of the ZPR register. The protection overrides encoded by the zone fields are shown in Table 6-3.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Z0	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Z9	Z10	Z11	Z12	Z13	Z14	Z15																

Figure 6-8: Zone-Protection Register (ZPR)

Table 6-3: Zone-Protection Register (ZPR) Bit Definitions

Bit	Name	Function	Description	
0:1	Z0	Zone 0 Protection	User Mode (MSR[PR]=1) 00—Override V in TLB entry. No access to the page is allowed. 01—No override. Use V, WR, and EX from TLB entry. 10—No override. Use V, WR, and EX from TLB entry. 11—Override WR and EX. Access the page as writable and executable.	Privileged Mode (MSR[PR]=0) 00—No override. Use V, WR, and EX from TLB entry. 01—No override. Use V, WR, and EX from TLB entry. 10—Override WR and EX. Access the page as writable and executable. 11—Override WR and EX. Access the page as writable and executable.
2:3	Z1	Zone 1 Protection		
4:5	Z2	Zone 2 Protection		
6:7	Z3	Zone 3 Protection		
8:9	Z4	Zone 4 Protection		
10:11	Z5	Zone 5 Protection		
12:13	Z6	Zone 6 Protection		
14:15	Z7	Zone 7 Protection		
16:17	Z8	Zone 8 Protection		
18:19	Z9	Zone 9 Protection		
20:21	Z10	Zone 10 Protection		
22:23	Z11	Zone 11 Protection		
24:25	Z12	Zone 12 Protection		
26:27	Z13	Zone 13 Protection		
28:29	Z14	Zone 14 Protection		
30:31	Z15	Zone 15 Protection		

The ZPR is a privileged SPR with an address of 944 (0x3B0) and is read and written using the **mfspr** and **mtspr** instructions.

Effect of Access Protection on Cache-Control Instructions

The access-protection mechanisms apply to certain cache-control instructions, depending on how those instructions affect data. Cache-control instructions—including those that affect the instruction cache—are treated as data loads or data stores by the access-protection mechanism. If an access-protection violation occurs, the resulting interrupt is a data-storage interrupt. The following summarizes how access protection is applied to cache-control instructions:

- Cache-control instructions that can modify data are treated as stores (writes) by the access-protection mechanism. Instructions that can cause loss of data by invalidating cachelines are also treated as stores. TLB write-protection and zone protection are used to restrict access by these instructions as follows:
 - dcbi**—Affected by TLBLO[WR] only. Because this is a privileged instruction, access cannot be denied by zone protection.
 - dcbz**—Affected by TLBLO[WR] and (in user mode only) ZPR[Zn]=00.
- Other cache-control instructions can invalidate an entire cache-congruence class.

These instructions are not address-specific and can affect multiple pages with different access protections. Because they are privileged instructions, access cannot be denied by zone protection.

- **dccci**—Affected by TLBLO[WR] only. This instruction can cause data loss by invalidating modified data in the cache-congruence class.
- **iccci**—Not affected by TLBLO[WR]. The instruction cache cannot hold modified data.

Both **dccci** and **iccci** can cause TLB-miss interrupts. Because these instructions are not address-specific, it is recommended that software does not execute them when data-relocation is enabled (MSR[DR]=1).

- Some cache-control instructions update system memory with data already present in the cache. These instructions are treated as loads (reads) by the access-protection mechanism rather than as stores. The reason is that stores were already used to place the modified data into the cache and passed the access-protection check. Therefore, these instructions are not affected by TLBLO[WR].
 - **dcbf**—Affected by ZPR[Zn]=00 in user mode only.
 - **dcbst**—Affected by ZPR[Zn]=00 in user mode only.
- Speculative cache-control instructions are restricted by TLB write-protection access control and by zone protection. However, if these instructions fail access protection checks they do not cause an exception and are instead treated as a “no operation”.
 - **dcba**—Affected by TLBLO[WR] and (in user mode only) ZPR[Zn]=00.
 - **dcbt**—Affected by ZPR[Zn]=00 in user mode only. This instruction is treated as a load and is therefore not affected by TLBLO[WR].
 - **dcbtst**—Affected by ZPR[Zn]=00 in user mode only. This instruction is treated as a load and is therefore not affected by TLBLO[WR].
 - **icbt**—Affected by ZPR[Zn]=00 in user mode only. This instruction is treated as a load and is therefore not affected by TLBLO[WR].
- Certain privileged cache-control instructions are treated as loads and are therefore unaffected by TLBLO[WR]. Because they are privileged instructions, access cannot be denied when ZPR[Zn]=00. These instructions are:
 - **dcread**.
 - **icbi**.
 - **icread**.

Table 6-4 summarizes the effect of access violations that occur when a cache-control instruction is executed. In this table, the “Read-Only Page” column applies to the execution of an instruction in privileged mode and (for the non-privileged instructions) user mode. The “No-Access Allowed Page” column applies to the execution of instructions only in user mode (no-access allowed protection is not available in supervisor mode).

Table 6-4: Effect of Cache-Control Instruction Access Violations

Instruction	Read-Only Page (TLBLO[WR]=0)	No-Access Allowed Page (ZPR[Zn]=00)
dcba	No operation.	No operation.
dcbf	No violation—treated as load.	Data-storage interrupt.
dcbi	Data-storage interrupt.	No violation—privileged instruction.
dcbst	No violation—treated as load.	Data-storage interrupt.
dcbt	No violation—treated as load.	No operation.
dcbtst	No violation—treated as load.	No operation.

Table 6-4: Effect of Cache-Control Instruction Access Violations

Instruction	Read-Only Page (TLBLO[WR]=0)	No-Access Allowed Page (ZPR[Zn]=00)
dcbz	Data-storage interrupt.	Data-storage interrupt.
dccci	Data-storage interrupt.	No violation—privileged instruction.
dcread	No violation—treated as load.	No violation—privileged instruction.
icbi	No violation—treated as load.	No violation—privileged instruction.
icbt	No violation—treated as load.	No operation.
iccci	Data-storage interrupt.	No violation—privileged instruction.
icread	No violation—treated as load.	No violation—privileged instruction.

UTLB Management

The UTLB serves as the interface between the processor MMU and memory-management software. System software manages the UTLB to tell the MMU how to translate virtual addresses into physical addresses. When a problem occurs due to a missing translation or an access violation, the MMU communicates the problem to system software using the exception mechanism. System software is responsible for providing interrupt handlers to correct these problems so that the MMU can proceed with memory translation.

Table 6-5 lists the PowerPC *TLB-management* instructions that enable system software to manage UTLB entries. These instructions are used to search the UTLB for specific entries, read entries, invalidate entries, and write entries. All of these instructions are privileged.

Table 6-5: TLB-Management Instructions

Mnemonic	Name	Operation	Operand Syntax
tlbia	TLB Invalidate All	Invalidates all UTLB entries by clearing their valid bits (TLBHI[V]) to 0. No other fields in the UTLB entries are modified.	—
tlbre	TLB Read Entry	rA contains an index value ranging from 0 to 63. Part of the UTLB entry specified by the index in rA is loaded into rD . If WS=0 , the tag portion (TLBHI) is loaded into rD and the PID is updated with the TLBHI[TID] field. If WS=1 , the data portion (TLBLO) is loaded into rD .	rD,rA,WS

Table 6-5: TLB-Management Instructions

Mnemonic	Name	Operation	Operand Syntax
tlbsx	TLB Search Indexed	If a translation is found, rD is loaded with the <i>index</i> of the UTLB entry for the page specified by EA. If a translation is not found, rD is undefined. The index is used by the tlbre and tlbre instructions. EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	rD,rA,rB
tlbsx.	TLB Search Indexed and Record	If a translation is found, rD is loaded with the <i>index</i> of the UTLB entry for the page specified by EA, and CR0[EQ] is set to 1. If a translation is not found, rD is undefined and CR0[EQ] is cleared to 0. The index is used by the tlbre and tlbre instructions. EA is calculated using register-indirect with index addressing: $EA = (rA \mid 0) + (rB)$	
tlbsync	TLB Synchronize	On the PPC405, this instruction performs no operation.	—
tlbwe	TLB Write Entry	rA contains an index value ranging from 0 to 63. Part of the UTLB entry specified by the index in rA is loaded with the value in rS . If WS=0 , the tag portion (TLBHI) is loaded from rS and the TLBHI[TID] field is updated with the PID. If WS=1 , the data portion (TLBLO) is loaded from rS .	rS,rA,WS

Software reads and writes UTLB entries using the **tlbre** and **tlbwe** instructions, respectively. These instructions specify an index (numbered 0 to 63) corresponding to one of the 64 entries in the UTLB. The tag and data portions are read and written separately, so software must execute two **tlbre** or **tlbwe** instructions to completely access an entry. The UTLB is searched for a specific translation using the **tlbsx** instruction. **tlbsx** locates a translation using an effective address and loads the corresponding UTLB index into a register.

Simplified mnemonics are defined for the TLB read and write instructions. See **TLB-Management Instructions**, [page 868](#), for more information.

The **tlbia** instruction invalidates the entire contents of the UTLB. Individual entries are invalidated using the **tlbwe** instruction to clear the valid bit in the tag portion of a TLB entry (TLBHI[V]).

The **tlbsync** instruction performs no operation on the PPC405 because the processor does not provide hardware support for multiprocessor memory coherency.

Recording Page Access and Page Modification

Software management of virtual-memory poses several challenges:

- In a virtual-memory environment, software and data often consume more memory than is physically available. Some of the software and data pages must be stored outside physical memory, such as on a hard drive, when they are not used. Ideally, the most-frequently used pages stay in physical memory and infrequently used pages are stored elsewhere.
- When pages in physical-memory are replaced to make room for new pages, it is important to know whether the replaced (old) pages were modified. If they were

modified, they must be saved prior to loading the replacement (new) pages. If the old pages were not modified, the new pages can be loaded without saving the old pages.

- A limited number of page translations are kept in the UTLB. The remaining translations must be stored in the page-translation table. When a translation is not found in the UTLB (due to a miss), system software must decide which UTLB entry to discard so that the missing translation can be loaded. It is desirable for system software to replace infrequently used translations rather than frequently used translations.

Solving the above problems in an efficient manner requires keeping track of page accesses and page modifications. The PPC405 does not track page access and page modification in hardware. Instead, system software can use the TLB-miss exceptions and the data-storage exception to collect this information. As the information is collected, it can be stored in a data structure associated with the page-translation table.

Page-access information is used to determine which pages should be kept in physical memory and which are replaced when physical-memory space is required. System software can use the valid bit in the TLB entry (TLBHI[V]) to monitor page accesses. This requires page translations be initialized as not valid (TLBHI[V]=0) to indicate they have not been accessed. The first attempt to access a page causes a TLB-miss exception, either because the UTLB entry is marked not valid or because the page translation is not present in the UTLB. The TLB-miss handler updates the UTLB with a valid translation (TLBHI[V]=1). The set valid bit serves as a record that the page and its translation have been accessed. The TLB-miss handler can also record the information in a separate data structure associated with the page-translation entry.

Page-modification information is used to indicate whether an old page can be overwritten with a new page or the old page must first be stored to a hard disk. System software can use the write-protection bit in the TLB entry (TLBLO[WR]) to monitor page modification. This requires page translations be initialized as read-only (TLBLO[WR]=0) to indicate they have not been modified. The first attempt to write data into a page causes a data-storage exception, assuming the page has already been accessed and marked valid as described above. If software has permission to write into the page, the data-storage handler marks the page as writable (TLBLO[WR]=1) and returns. The set write-protection bit serves as a record that a page has been modified. The data-storage handler can also record this information in a separate data structure associated with the page-translation entry.

Tracking page modification is useful when virtual mode is first entered and when a new process is started.

Maintaining Shadow-TLB Consistency

The PPC405 TLBs are maintained by two different mechanisms: software manages the UTLB and the processor manages the shadow TLBs. Software must ensure the shadow TLBs remain consistent with the UTLB when updates are made to entries in the UTLB. If software updates any field in a UTLB entry, it *must* synchronize that update with the shadow TLBs. Failure to properly synchronize the shadow TLBs can cause unexpected behavior.

Synchronization occurs when the processor hardware replaces a shadow-TLB entry with an updated entry from the UTLB. To force a replacement, software must invalidate the shadow-TLB entry. This forces the MMU to read the modified entry from the UTLB the next time it is accessed. The processor invalidates *all* shadow-TLB entries when any of the following context-synchronizing events occur:

- An **isync** instruction is executed.
- An **sc** instruction is executed.
- An interrupt occurs.
- An **rfi** or **rfci** instruction is executed.

TLB entries are normally modified by interrupt handlers. The shadow TLB is automatically invalidated when an interrupt occurs. The interrupt also disables address translation, placing the processor in real mode. The MMU does not access the UTLB or update the shadow TLBs when address translation is disabled. If the interrupt handler updates the UTLB and returns from the interrupt handler (using **rfi**) without enabling virtual mode, no additional context synchronization is required.

However, if virtual mode is enabled by the interrupt handler and the UTLB is updated, those updates are not synchronized with the shadow TLBs until an **rfi** is executed to exit the handler. If UTLB updates must be reflected in the shadow TLB while the interrupt handler is executing, **isync** must be executed after updating the UTLB.

As a general rule, software manipulation of UTLB entries should always be followed by a context-synchronizing operation, typically an **isync** instruction.

Exceptions and Interrupts

The PowerPC embedded-environment architecture extends the base PowerPC exception and interrupt mechanism in the following ways:

- A dual-level interrupt structure is defined supporting critical and noncritical interrupts.
- New save/restore registers are defined in support of the dual-level interrupt structure.
- A new interrupt-return instruction is defined in support of the dual-level interrupt structure.
- New special-purpose registers are defined for recording exception information.
- New exceptions and interrupts are defined.

This chapter describes the exceptions recognized by the PPC405D5 and how the interrupt mechanism responds to those exceptions.

Overview

Exceptions are events detected by the processor that often require action by system software. Most exceptions are unexpected and are the result of error conditions. A few exceptions can be programmed to occur through the use of exception-causing instructions. Some exceptions are generated by external devices and communicated to the processor using external signalling. Still other exceptions can occur when pre-programmed conditions are recognized by the processor.

Interrupts are automatic control transfers that occur as a result of an exception. An interrupt occurs when the processor suspends execution of a program after detecting an exception. The processor saves the suspended-program machine state and a return address into the suspended program. This information is stored in a pair of special registers, called *save/restore registers*. A predefined machine state is loaded by the processor, which transfers control to an *interrupt handler*. An interrupt handler is a system-software routine that responds to the interrupt, often by correcting the condition causing the exception. System software places interrupt handlers at predefined addresses in physical memory and the interrupt mechanism automatically transfers control to the appropriate handler based on the exception condition.

An interrupt places the processor in both privileged mode and real mode (instruction-address and data-address relocation are disabled). Interrupts are context-synchronizing events. All instructions preceding the interrupted instruction are guaranteed to have completed execution when the interrupt occurs. All instructions following the interrupted instruction (in the program flow) are discarded.

Returning from an interrupt handler to an interrupted program requires that the old machine state and program return address be restored from the save/restore register pair. This is accomplished using a *return-from-interrupt* instruction. Like interrupts, return-from-interrupt instructions are context synchronizing.

Certain interrupts can be disabled (masked) or enabled (unmasked). Disabling an interrupt prevents it from occurring when the corresponding exception condition is detected by the processor.

Synchronous and Asynchronous Exceptions

Exceptions (and the corresponding interrupt) can be synchronous or asynchronous. Synchronous exceptions are directly caused by the execution or attempted execution of an instruction. Asynchronous exceptions occur independently of instruction execution. The cause of an asynchronous exception is generally not related to the instruction executing at the time the exception occurs.

Precise and Imprecise Interrupts

Most interrupts are precise. A precise interrupt occurs in program order and on the instruction boundary where the exception is recognized. A precise interrupt causes the following to occur:

- The return address points to the excepting instruction. For synchronous exceptions, the return address points to either the instruction causing the exception or the instruction that immediately follows, depending on the exception condition. For asynchronous exceptions, the return address points to the instruction boundary where the exception is recognized by the processor.
- All instructions preceding the excepting instruction complete execution before the interrupt occurs. However, it is possible that some storage accesses initiated by those instructions are not complete with respect to external devices.
- Depending on the exception condition, it is possible for the excepting instruction to have completed execution, partially completed execution, or not have begun execution.
- No instructions following the excepting instruction are executed prior to transferring control to the interrupt handler.

When an imprecise interrupt occurs, the excepting instruction is unrelated to the exception condition. Here, there is a delay between the point where the exception is recognized by the processor and the time when the interrupt occurs. An imprecise interrupt causes the following to occur:

- The excepting instruction follows (in program order) the instruction boundary where the exception is recognized by the processor. The delay can span several instructions.
- All instructions preceding the excepting instruction complete execution before the interrupt occurs. However, it is possible that some storage accesses initiated by those instructions are not complete with respect to external devices.
- It is possible for the excepting instruction to have completed execution, partially completed execution, or not have begun execution.
- No instructions following the excepting instruction are executed prior to transferring control to the interrupt handler.

On the PPC405, only the machine-check interrupt is imprecise. A machine check can be caused indirectly by the execution of an instruction. In this case, it is possible for the processor to execute additional instructions before recognizing the occurrence of a machine check.

Partially-Executed Instructions

Certain instructions can cause an alignment exception or data-storage exception part-way through their execution. When an interrupt occurs, some software-visible state can be updated to reflect the partial execution of the excepting instruction. The instructions and the effect interrupts have on partial execution are as follows:

- Load-multiple and load-string instructions.

It is possible that some of the target registers are updated when a data-storage exception or an alignment exception occurs. When the instruction is restarted, the modified registers are updated again.

- Store-multiple and store-string instructions.

It is possible that some of the target bytes in memory are updated when a data-storage exception or an alignment exception occurs. When the instruction is restarted, the modified memory locations are updated again.

- Scalar load instructions that cross a word boundary.

It is possible that some memory bytes have been accessed (read) when a data-storage exception or alignment exception occurs. However, no registers are updated.

- Scalar store instructions that cross a word boundary.

It is possible that some of the target bytes in memory are updated when a data-storage exception or alignment exception occurs. If the instruction is an update form, the update register is *not* updated. When the instruction is restarted, the modified memory locations are updated again.

In the above cases, memory protection is never violated by the partial execution of an instruction. No other instruction updates software-visible state if an exception occurs part-way through execution.

To prevent load and store instructions from being interrupted and restarted, only scalar instructions (not string or multiple) should be used to reference memory. Also, one of the following two rules must be followed:

- The memory operand must be aligned on the operand-size boundary (see [Table 2-1, page 389](#)).
- The accessed memory location must be protected by the guarded storage attribute (see [Guarded \(G\), page 488](#)).

If a properly-aligned scalar load or store is interrupted, a memory-access request does not appear on the processor local bus (PLB). Conversely, the processor does not interrupt a properly-aligned scalar load or store once its corresponding memory-access request appears on the PLB. Thus, the guarded storage attribute is not required to prevent interruption of properly-aligned loads and stores.

PPC405D5 Exceptions and Interrupts

[Table 7-1](#) lists the exceptions supported by the PPC405D5. Included is the exception-vector offset into the interrupt-handler table, the exception classification, and a brief description of the cause. Gray-shaded rows indicate exceptions that are not supported by the PPC405D5 but can occur on other implementations of the PowerPC 405 processor. Refer to [Interrupt Reference, page 538](#), for a detailed description of each exception and its resulting interrupt.

Table 7-1: Exceptions Supported by the PPC405D5

Exception	Vector Offset	Classification			Cause
Critical Input	0x0100	Critical	Asynchronous	Precise	External critical-interrupt signal.
Machine Check	0x0200	Critical	Asynchronous	Imprecise	External bus error.
Data Storage	0x0300	Noncritical	Synchronous	Precise	Data-access violation.
Instruction Storage	0x0400	Noncritical	Synchronous	Precise	Instruction-access violation.
External	0x0500	Noncritical	Asynchronous	Precise	External noncritical-interrupt signal.

Table 7-1: Exceptions Supported by the PPC405D5 (Continued)

Exception	Vector Offset	Classification			Cause
Alignment	0x0600	Noncritical	Synchronous	Precise	Unaligned operand of dcread , lwarx , stwcx . dcbz to non-cacheable or write-through memory.
Program	0x0700	Noncritical	Synchronous	Precise	Improper or illegal instruction execution. Execution of trap instructions.
FPU Unavailable	0x0800	Noncritical	Synchronous	Precise	Attempt to execute an FPU instruction when FPU is disabled.
System Call	0x0C00	Noncritical	Synchronous	Precise	Execution of sc instruction.
APU Unavailable	0x0F20	Noncritical	Synchronous	Precise	Attempt to execute an APU instruction when APU is disabled.
Programmable-Interval Timer	0x1000	Noncritical	Asynchronous	Precise	Time-out on the programmable-interval timer.
Fixed-Interval Timer	0x1010	Noncritical	Asynchronous	Precise	Time-out on the fixed-interval timer.
Watchdog Timer	0x1020	Critical	Asynchronous	Precise	Time-out on the watchdog timer.
Data TLB Miss	0x1100	Noncritical	Synchronous	Precise	No data-page translation found.
Instruction TLB Miss	0x1200	Noncritical	Synchronous	Precise	No instruction-page translation found.
Debug	0x2000	Critical	Asynchronous and synchronous	Precise	Occurrence of a debug event.

Critical and Noncritical Exceptions

The PPC405 supports critical and noncritical exceptions. Generally, the processor responds to critical exceptions before noncritical exceptions (certain debug exceptions are handled at a lower priority). Four exceptions and their associated interrupts are critical:

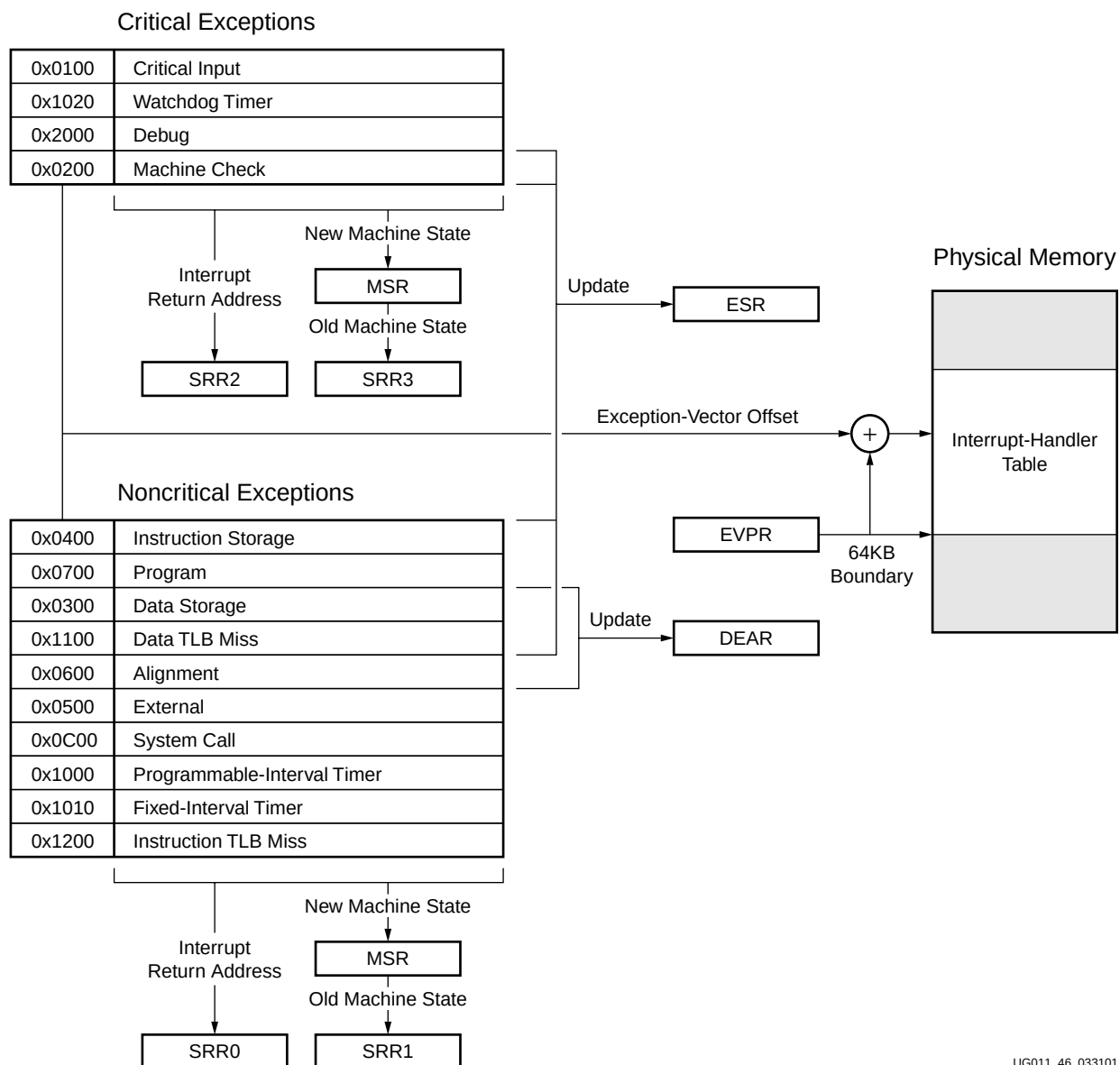
- Critical-input exception.
- Machine-check exception.
- Watchdog-timer exception.
- Debug exception.

Critical interrupts use a different save/restore register pair (SRR2 and SRR3) than is used by noncritical interrupts (SRR0 and SRR1). This enables a critical interrupt to interrupt a noncritical-interrupt handler. The state saved by the noncritical interrupt is not overwritten by the critical interrupt.

Because a different register pair is used for saving processor state, a different instruction is used to return from critical interrupt handlers—**rfci**.

Transferring Control to Interrupt Handlers

Figure 7-1 shows how the components of the PPC405 exception mechanism interact when transferring program control to an interrupt handler.



UG011_46_033101

Figure 7-1: PPC405 Exception Mechanism

Referring to **Figure 7-1**, the actions performed by the processor when an interrupt occurs are:

1. Save the interrupt-return address (effective address).

Generally, the return address is either that of the instruction that caused the exception, or the next-sequential instruction that would have executed had no exception occurred. It is saved in one of two save/restore registers, depending on the type of interrupt:

- Critical interrupts load SRR2 with the return address.
- Noncritical interrupts load SRR0 with the return address.

Refer to the specific interrupt description in **Interrupt Reference**, **page 538** for information on the saved return address.

2. Save the interrupted-program state.

The contents of the machine-state register (MSR) are copied into one of two save/restore registers, depending on the type of interrupt:

- Critical interrupts load SRR3 with a copy of the MSR.
- Noncritical interrupts load SRR1 with a copy of the MSR.

3. Update the exception-syndrome register (ESR), if applicable.

Five exceptions report status information in the ESR when control is transferred to the interrupt handler (ESR is not modified by the remaining exceptions):

- Machine check.
- Data storage.
- Instruction storage.
- Program.
- Data TLB miss.

Interrupt handlers use the ESR to determine the cause of an exception.

4. Update the data exception-address register (DEAR), if applicable.

Three exceptions report the address of a failed data access in the DEAR when control is transferred to the interrupt handler (DEAR is not modified by the remaining exceptions):

- Data storage.
- Alignment.
- Data TLB miss.

5. Load the new program state into the MSR.

All interrupts load new program state into the MSR. The new state places the processor in privileged mode. Instruction-address and data-address translation are disabled, placing the processor in real mode. Certain interrupts are disabled, depending on the exception.

6. Synchronize the processor context.

All interrupts are context synchronizing. The processor fetches and executes the first instruction in the interrupt handler in the context established by the new MSR contents.

7. Transfer control to the interrupt handler.

An exception-vector offset is associated with each exception. The offset is added to a 64KB-aligned base address located in the exception-vector prefix register (EVPR). The sum represents a physical address that points to the first instruction of the interrupt handler.

Interrupt handlers are located in an interrupt-handler table. The available space in this table is generally insufficient to hold entire interrupt handlers. Instead, system software typically places “glue code” in the table for transferring control to the full handler, located elsewhere in memory.

Returning from Interrupt Handlers

System software exits an interrupt handler using one of two privileged instructions. Noncritical-interrupt handlers return to an interrupted program using the *return-from-interrupt* instruction (**rfi**). Critical-interrupt handlers return to an interrupted program using the *return-from-critical-interrupt* instruction (**rfci**). Both instructions operate in a similar fashion, with the only difference being the save/restore register pair used to restore the interrupted-program state. **rfi** and **rfci** perform the following functions:

1. All previous instructions complete execution in the context they were issued (privilege, protection, and address-translation mode).
2. All previous instructions are completed to a point where they can no longer cause an exception.
3. The processor loads the MSR with the interrupted-program state from one of two save/restore registers, depending on the instruction:
 - **rfi** copies SRR1 into the MSR.
 - **rfdi** copies SRR3 into the MSR.
4. Processor context is synchronized.
Both instructions are context synchronizing. The processor fetches and executes the instruction at the return address in the interrupted-program context.
5. The processor begins fetching and executing instructions from the interrupted program:
 - Instructions are fetched from the address in SRR0 following completion of the **rfdi**.
 - Instructions are fetched from the address in SRR2 following completion of the **rfdi**.

Simultaneous Exceptions and Interrupt Priority

The PPC405 interrupt mechanism responds to exceptions serially. If multiple exceptions are pending simultaneously, the associated interrupts occur in a consistent and predictable order. Even though critical and noncritical exceptions use different save/restore register pairs, simultaneous occurrences of these exceptions are also processed serially.

The PPC405 uses the interrupt priority shown in [Table 7-2](#) for handling simultaneous exceptions. Lower-priority interrupts occur ahead of *masked* higher-priority interrupts.

Table 7-2: Interrupt Priority for Simultaneous Exceptions

Priority	Exception	Cause
1	Machine check—Data.	External bus error during data access.
2	Debug—Instruction-address compare.	Instruction-address compare (IAC) debug event.
3	Machine check—Instruction.	Attempted execution of an instruction for which an external bus error occurred during instruction fetch.
4	Debug—Exception.	Exception (EDE) debug event.
	Debug—Unconditional.	Unconditional (UDE) debug event.
5	Critical input	Critical-interrupt input signal is asserted.
6	Watchdog timer	Watchdog timer time-out.
7	Instruction TLB Miss	Attempted execution of an instruction from a memory address with no valid, matching page translation loaded in the TLB (virtual mode only).
8	Instruction storage—No access.	In user mode, attempted execution of an instruction from a memory address with no-access-allowed zone protection (virtual mode only).
9	Instruction storage—Non-executable.	Attempted execution of an instruction from a non-executable memory address (virtual mode only).
	Instruction storage—Guarded.	Attempted execution of an instruction from a guarded memory address.

Table 7-2: Interrupt Priority for Simultaneous Exceptions (Continued)

Priority	Exception	Cause
10	Program	Attempted execution of: - An illegal instruction. - Unimplemented floating-point instructions. - Unimplemented auxiliary-processor instructions. - A privileged instruction from user mode. - Execution of a trap instruction that satisfies the trap conditions.
	System call	Execution of the sc instruction.
	FPU unavailable	Attempted execution of an implemented floating-point instruction when MSR[FP]=0. Not implemented by the PPC405D5.
	APU unavailable	Attempted execution of an implemented auxiliary-processor instruction when MSR[AP]=0. Not implemented by the PPC405D5.
11	Data TLB Miss	Attempted access of data from an address with no valid, matching page translation loaded in the TLB (virtual mode only).
12	Data storage—No access.	In user mode, attempted access of data from a memory address with no-access-allowed zone protection (virtual mode only).
13	Data storage—Read-only.	Attempted data write (store) into a read-only memory address (virtual mode only).
	Data storage—User defined.	Attempted data write (store) into a memory address with the U0 storage attribute set to 1, when U0 exceptions are enabled.
14	Alignment	Attempted execution of: dcbz to a non-cacheable or write-through cacheable address. lwarx or stwcx. to an address that is not word aligned. dcread to an address that is not word aligned (privileged mode only).
15	Debug—Branch taken.	Branch taken (BT) debug event.
	Debug—Data-address compare.	Data-address compare (DAC) debug event.
	Debug—Data-value compare.	Data-value compare (DVC) debug event.
	Debug—Instruction completion.	Instruction completion (IC) debug event.
	Debug—Trap instruction.	Trap instruction (TDE) debug event.
16	External	Noncritical-interrupt input signal is asserted.
17	Fixed-interval timer	Fixed-interval timer time-out.
18	Programmable-interval timer	Programmable-interval timer time-out.

Persistent Exceptions and Interrupt Masking

When certain exceptions are recognized by the processor, system software can prevent the corresponding interrupt from occurring by disabling, or *masking*, the interrupt. In general, disabling an interrupt only delays its occurrence. The processor continues to recognize the exception. When software re-enables (unmasks) the interrupt, the interrupt occurs. Such exceptions are referred to as *persistent* exceptions.

An persistent exception normally sets a status bit in a specific register associated with the exception mechanism. The only way for software to prevent the interrupt from occurring is to clear the exception-status bit before unmasking (enabling) the interrupt. Likewise, the interrupt handler must clear the exception-status bit to prevent the interrupt from reoccurring, should it be re-enabled upon exiting the interrupt handler.

The following exceptions are persistent and their corresponding interrupts can be disabled:

- Critical input—Exception status is recorded in a device control register (DCR) associated with the external interrupt controller. The MSR[CE] bit is used to enable and disable the interrupt.
- External—Exception status is recorded in a device control register (DCR) associated with the external interrupt controller. The MSR[EE] bit is used to enable and disable the interrupt.
- Programmable-interval timer—Exception status is recorded in the PIT-status bit of the timer-status register, TSR[PIS]. The MSR[EE] bit is used to enable and disable the interrupt.
- Fixed-interval timer—Exception status is recorded in the FIT-status bit of the timer-status register, TSR[FIS]. The MSR[EE] bit is used to enable and disable the interrupt.
- Debug—Imprecise exception status is recorded in the imprecise-debug exception bit of the debug-status register, DBSR[IDE]. This indicates that a debug event occurred while debug interrupts were disabled. Other bits in the DBSR can be set, indicating which debug events occurred while the interrupt was disabled. The MSR[DE] bit is used to enable and disable the interrupt.

The watchdog-timer exception is also persistent, but its persistence *prevents* further interrupts from occurring. This function causes an interrupt to occur on a watchdog time-out but prevents interrupts on subsequent time-outs. Exception status is recorded in the watchdog-status bit of the timer-status register, TSR[WIS]. Once the status bit is set, further watchdog-timer time-outs do not cause an interrupt. Clearing the bit enables time-out interrupts to occur. The MSR[CE] bit is used to enable and disable the interrupt.

The machine-check interrupt can be disabled but the exception is not persistent. Machine-check exception status is recorded in the machine-check interrupt status bit of the exception-syndrome register, ESR[MCI]. However, enabling machine-check interrupts when the status bit is set does not necessarily cause the interrupt to occur. Instead, the interrupt occurs when the appropriate external bus-error signal is asserted. The error signal persists only for the duration of the bus cycle that causes the error.

Interrupt-Handling Registers

When an exception occurs and an interrupt is taken, the interrupt-handling mechanism uses the following registers:

- *Save/restore register 0 (SRR0)*—Contains the return address for noncritical interrupts.
- *Save/restore register 1 (SRR1)*—Contains a copy of the MSR for noncritical interrupts.
- *Save/restore register 2 (SRR2)*—Contains the return address for critical interrupts.
- *Save/restore register 3 (SRR3)*—Contains a copy of the MSR for critical interrupts.
- *Exception-vector prefix register (EVPR)*—Contains the base address of the interrupt-handler table.
- *Exception-syndrome register (ESR)*—Identifies the cause of an exception. ESR is used by five exceptions.
- *Data exception-address register (DEAR)*—Contains the memory-operand effective address of the data-access instruction that caused the exception. DEAR is used by three exceptions.

The machine-state register is also updated, placing the processor in privileged and real mode. The following sections describe the effect of the interrupt-handling mechanism on the interrupt-handling registers.

Machine-State Register Following an Interrupt

During an interrupt, the contents of the MSR (see [page 467](#)) are loaded into either SRR1 (noncritical interrupts) or SRR3 (critical interrupts). Depending on the interrupt, the MSR is updated with the values shown in [Table 7-3](#).

Table 7-3: Effect of Interrupts on Machine-State Register Contents

Bit	Name	Interrupt	Value	Description
0:5		All	0	Reserved
6	AP	All	0	This unsupported bit is cleared, but otherwise ignored.
7:11		All	0	Reserved
12	APE	All	0	This unsupported bit is cleared, but otherwise ignored.
13	WE	All	0	Processor wait state is disabled.
14	CE	Critical-Input Interrupt Machine-Check Interrupt Watchdog-Timer Interrupt Debug Interrupt	0	Critical-input interrupts are disabled (masked).
		All Others	No Change	Critical-input interrupts are enabled or disabled.
15		All	0	Reserved
16	EE	All	0	External interrupts are disabled (masked).
17	PR	All	0	Processor is in privileged mode.
18	FP	All	0	This unsupported bit is cleared, but otherwise ignored.
19	ME	Machine-Check Interrupt	0	Machine-check interrupts are disabled (masked).
		All Others	No Change	Machine-check interrupts are enabled or disabled.
20	FE0	All	0	This unsupported bit is cleared, but otherwise ignored.
21	DWE	All	0	Debug wait-mode is disabled.
22	DE	Critical-Input Interrupt Machine-Check Interrupt Watchdog-Timer Interrupt Debug Interrupt	0	Debug interrupts are disabled (masked).
		All Others	No Change	Debug interrupts are enabled or disabled.
23	FE1	All	0	This unsupported bit is cleared, but otherwise ignored.
24:25		All	0	Reserved
26	IR	All	0	Instruction-address translation is disabled (real mode).
27	DR	All	0	Data-address translation is disabled (real mode).
28:31		All	0	Reserved

Save/Restore Registers 0 and 1

The save/restore registers 0 and 1 (SRR0 and SRR1) are 32-bit registers used to save machine state when a noncritical interrupt occurs. The format of each register is shown in Figure 7-2.

0	30	31
Interrupted-Instruction Effective Address		0 0
SRR0		

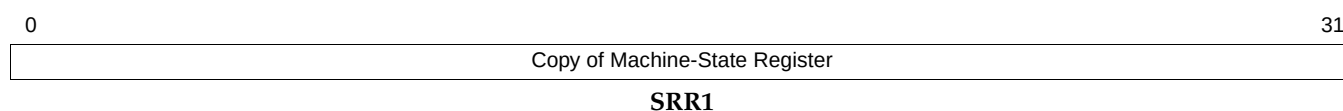


Figure 7-2: Save/Restore Registers 0 and 1

During a noncritical interrupt, SRR0 is loaded by the processor with the effective address of the interrupted instruction (bits 30:31 are always 0, because instruction addresses are word aligned). An **rfi** instruction is used to return from the noncritical-interrupt handler to the instruction address stored in SRR0. Depending on the exception, this effective address represents either:

- The instruction that caused the exception.
- The instruction that would have executed had no exception occurred. For example, when an **sc** instruction is executed SRR0 is loaded with the instruction effective address following the **sc**.

See the specific instruction for details.

SRR1 is loaded with a copy of the MSR when a noncritical interrupt occurs. An **rfi** instruction restores the machine state by copying the contents of SRR0 into the MSR (defined and reserved MSR fields are updated).

SRR0 is a privileged SPR with an address of 26 (0x01A) and SRR1 is a privileged SPR with an address of 27 (0x01B). Both registers are read and written using the **mfspr** and **mtspr** instructions.

Save/Restore Registers 2 and 3

The save/restore registers 2 and 3 (SRR2 and SRR3) are 32-bit registers used to save machine state when a critical interrupt occurs. Interrupts defined as critical are:

- Critical-Input Interrupt.
- Machine-Check Interrupt.
- Watchdog-Timer Interrupt.
- Debug Interrupt.

The format of each register is shown in **Figure 7-3**.

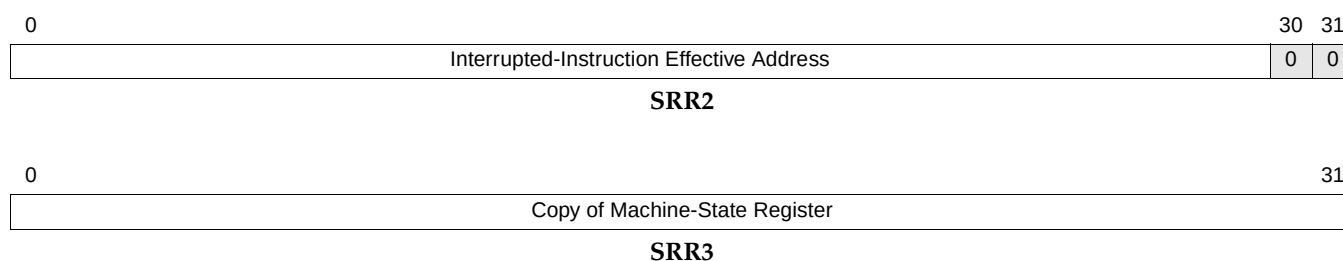


Figure 7-3: Save/Restore Registers 2 and 3

During a critical interrupt, SRR2 is loaded by the processor with the effective address of the interrupted instruction (bits 30:31 are always 0, because instruction addresses are word aligned). An **rfci** instruction is used to return from the critical-interrupt handler to the instruction address stored in SRR2. Depending on the exception, this effective address represents either:

- The instruction that caused the exception.
- The instruction that would have executed had no exception occurred. For example, when a watchdog-timer interrupt occurs SRR2 is loaded with the effective address of the next-sequential instruction.

See the specific instruction for details.

SRR3 is loaded with a copy of the MSR when a critical interrupt occurs. An **rfci** instruction restores the machine state by copying the contents of SRR3 into the MSR (defined and reserved MSR fields are updated).

SRR2 is a privileged SPR with an address of 990 (0x3DE) and SRR3 is a privileged SPR with an address of 991 (0x3DF). Both registers are read and written using the **mf spr** and **mt spr** instructions.

Exception-Vector Prefix Register

The exception-vector prefix register (EVPR) is a 32-bit register that contains the base address of the interrupt-handler table. Software can locate the interrupt-handler table anywhere in physical-address space, with a base address that falls on a 64KB-aligned boundary. When an exception occurs, the high-order 16 bits in EVPR are concatenated on the left with the 16-bit exception-vector offset (the low-order 16 reserved bits in the EVPR are ignored by the processor). The resulting 32-bit exception-vector physical address is used by the interrupt mechanism to transfer control to the appropriate interrupt handler. **Figure 7-4** shows the format of the EVPR register. The fields in the EVPR are defined as shown in **Table 7-4**.

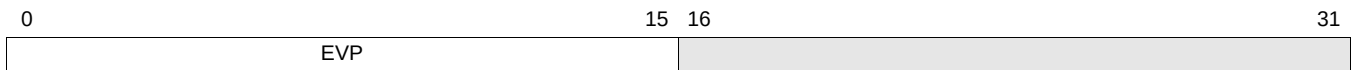


Figure 7-4: Exception-Vector Prefix Register (EVPR)

Table 7-4: Exception-Vector Prefix Register (EVPR) Field Definitions

Bit	Name	Function	Description
0:15	EVP	Exception-Vector Prefix	Used to locate the interrupt-handler table base address on an arbitrary 64KB physical-address boundary.
16:31		Reserved	

The EVPR is a privileged SPR with an address of 982 (0x3D6) and is read and written using the **mf spr** and **mt spr** instructions.

Exception-Syndrome Register

The exception-syndrome register (ESR) is a 32-bit register used to identify the cause of the following exceptions:

- Program exception.
- Instruction machine-check exception.
- Instruction-storage exception.
- Data-storage exception.
- Data TLB-miss exception.

Figure 7-5 shows the format of the ESR register. The fields in the ESR are defined as shown in **Table 7-5**.

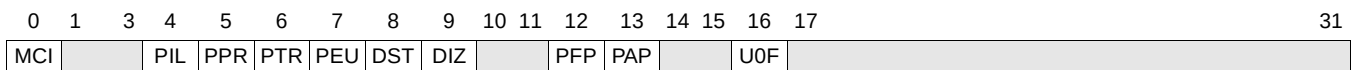


Figure 7-5: Exception-Syndrome Register (ESR)

Table 7-5: Exception-Syndrome Register (ESR) Field Definitions

Bit	Name	Function	Description
0	MCI	Machine Check—Instruction 0—Did not occur. 1—Occurred.	When set to 1, indicates an instruction machine-check exception occurred.
1:3		Reserved	
4	PIL	Program—Illegal Instruction 0—Did not occur. 1—Occurred.	When set to 1, indicates an illegal-instruction program exception occurred.
5	PPR	Program—Privileged Instruction 0—Did not occur. 1—Occurred.	When set to 1, indicates a privileged-instruction program exception occurred.
6	PTR	Program—Trap Instruction 0—Did not occur. 1—Occurred.	When set to 1, indicates a successful trap-instruction compare occurred, resulting in a trap-instruction program exception.
7	PEU	Program—Unimplemented Instruction 0—Did not occur. 1—Occurred.	Not supported by the PPC405D5.
8	DST	Data Storage—Store Instruction 0—Did not occur. 1—Occurred.	When set to 1, indicates a store instruction (including dcbi , dcbz , and dccci) caused an exception to occur (data-storage exception or data TLB-miss exception).
9	DIZ	Data and Instruction Storage—Zone Protection 0—Did not occur. 1—Occurred.	When set to 1, indicates a zone-protection violation caused a data-storage or instruction-storage exception to occur. For instruction-storage exceptions, DIZ is cleared to 0 when the exception is caused by a fetch from a non-executable address or from guarded storage.
10:11		Reserved	
12	PFP	Program—Floating-Point Instruction 0—Did not occur. 1—Occurred.	Not supported by the PPC405D5.
13	PAP	Program—Auxiliary-Processor Instruction 0—Did not occur. 1—Occurred.	Not supported by the PPC405D5.
14:15		Reserved	
16	U0F	Data Storage—U0 Protection 0—Did not occur. 1—Occurred.	When set to 1, indicates a U0-protection violation caused a data-storage exception to occur.
17:31		Reserved	

In general, an exception sets its corresponding ESR bit and clears all other bits. However, if machine-check interrupts are not enabled ($MSR[ME]=0$), a previously set $ESR[MCI]$ bit is not cleared when other exceptions occur. This is true whether or not the other exception occurs simultaneously with the instruction machine-check exception that sets $ESR[MCI]$. Handling $ESR[MCI]$ in this manner prevents losing a record of an instruction machine-

check exception when machine-check interrupts are disabled. It is recommended that instruction machine-check interrupt handlers clear the ESR[MSI] bit prior to returning to the interrupted program.

If machine-check interrupts are enabled (MSR[ME]=1), an instruction machine-check exception sets ESR[MCI] and clears all other ESR bits.

The ESR is a privileged SPR with an address of 980 (0x3D4) and is read and written using the **mf spr** and **mt spr** instructions.

Data Exception-Address Register

The data exception-address register (DEAR) is a 32-bit register that contains the memory-operand effective address of the data-access instruction that caused one of the following exceptions:

- Alignment exception.
- Data-storage exception.
- Data TLB-miss exception.

Figure 7-6 shows the format of the DEAR register.

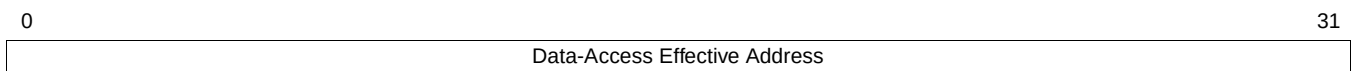


Figure 7-6: Data Exception-Address Register (DEAR)

The DEAR is a privileged SPR with an address of 981 (0x3D5) and is read and written using the **mf spr** and **mt spr** instructions.

Interrupt Reference

This section describes each interrupt, using the following outline:

- The name of each interrupt is shown, followed by its exception-vector offset.
- Interrupts are classified based on whether they are critical or noncritical, synchronous or asynchronous, and precise or imprecise.
- The conditions that cause the exception for which the interrupt occurs are described.
- The methods used to enable and disable (mask) the interrupt are described, if applicable.
- The values of the registers affected by taking the interrupt are shown.

Critical-Input Interrupt (0x0100)

Interrupt Classification

- Critical—return using the **rfei** instruction.
- Asynchronous.
- Precise.

Description

A critical-input exception is caused by an external device (usually the external-interrupt controller) asserting the critical-interrupt input signal to the processor.

This exception is persistent. To prevent repeated interrupts from occurring, the interrupt handler must clear the exception status in the appropriate device control register (DCR) associated with the external-interrupt controller before returning, and before re-enabling critical interrupts.

This interrupt is enabled using the critical-interrupt enable bit (CE) in the MSR. When MSR[CE]=1, the processor recognizes exceptions caused by asserting the critical-interrupt input signal and forces a critical-input interrupt to occur. When MSR[CE]=0, the processor does not recognize the critical-interrupt input signal and critical-input interrupts cannot occur.

All maskable interrupts, except those caused by machine-check exceptions, are disabled when a critical-input interrupt occurs. The critical-input interrupt handler should not re-enable MSR[CE] until it has cleared the exception and saved SRR2 and SRR3. Saving these registers avoids potential corruption of the interrupt handler should a watchdog-timer interrupt or another critical-input interrupt occur.

In some PowerPC implementations, this exception-vector offset corresponds to a system-reset interrupt.

Affected Registers

Register	Value After Interrupt
SRR0	Not used.
SRR1	
SRR2	Loaded with the effective address of the next-sequential instruction to be executed at the point the interrupt occurs.
SRR3	Loaded with a copy of the MSR at the point the interrupt occurs.
ESR	Not used.
DEAR	
MSR	[AP, APE, WE, CE, EE, PR, FP, FE0, DWE, DE, FE1, IR, DR] ← 0. [ME] ← Unchanged.

Machine-Check Interrupt (0x0200)

Interrupt Classification

- Critical—return using the **rfci** instruction.
- Asynchronous (not guaranteed to be synchronous).
- Imprecise (not guaranteed to be precise).

Description

A machine-check exception is caused by an error detected on the processor-local bus (PLB). External devices assert an error signal to the processor when a machine-check error is recognized. The processor supports two external PLB-error signals, one for instructions and one for data. This enables the processor to differentiate between machine checks due to instruction fetching and those caused by data access.

This interrupt is enabled using the machine-check enable bit (ME) in the MSR. When MSR[ME]=1, the processor recognizes exceptions caused by asserting one of the PLB-error input signals and forces a machine-check interrupt to occur. When MSR[ME]=0, the processor continues to recognize the PLB-error input signals, but an associated machine-check interrupt does not occur. The exception is not persistent.

All maskable interrupts, including those caused by machine-check exceptions, are disabled when a machine-check interrupt occurs. The machine-check interrupt handler should not re-enable MSR[ME] until it has saved SRR2 and SRR3. Saving these registers avoids potential corruption of the interrupt handler should another machine-check interrupt occur.

Instruction Machine-Check Interrupt

Instruction machine-check errors are reported to the processor by an external device during an instruction fetch. However, the exception and subsequent interrupt do not occur until the processor attempts to *execute* the instruction that caused the error. If the erroneous instruction fetch results in a cache-line fill, any instruction later executed from the cacheline can cause the exception to occur. Machine-check exceptions associated with cached instructions always invalidate the corresponding instruction-cacheline.

ESR[MCI] is set to 1 by all instruction machine-check exceptions. This is true regardless of whether the machine-check interrupt is enabled or not. If machine-check interrupts are disabled (MSR[ME]=0), software can periodically examine ESR[MCI] to determine if any instruction machine-check exceptions have occurred. Software should clear ESR[MCI] to 0 before returning from the machine-check interrupt handler to avoid any ambiguity when handling subsequent machine-check interrupts.

Data Machine-Check Interrupt

Data machine-check errors are reported to the processor by an external device during a data access. Determining the cause is dependent on the system implementation. Generally the data machine-check interrupt handler must examine the error-reporting registers located in the external-PLB devices to determine the exception cause.

Affected Registers

Instruction Machine-Check Interrupt

Register	Value After Interrupt
SRR0	Not used.
SRR1	
SRR2	Loaded with the effective address of the instruction that caused the machine-check exception.
SRR3	Loaded with a copy of the MSR at the point the interrupt occurs.
ESR	[MCI] $\leftarrow$ 1 All remaining bits are cleared to 0.
DEAR	Not used.
MSR	[AP, APE, WE, CE, EE, PR, FP, ME, FE0, DWE, DE, FE1, IR, DR] $\leftarrow$ 0.

Data Machine-Check Interrupt

Register	Value After Interrupt
SRR0	Not used.
SRR1	
SRR2	Loaded with the effective address of the next-sequential instruction to be executed at the point the interrupt occurs.
SRR3	Loaded with a copy of the MSR at the point the interrupt occurs.
ESR	Not used.
DEAR	
MSR	[AP, APE, WE, CE, EE, PR, FP, ME, FE0, DWE, DE, FE1, IR, DR] $\leftarrow$ 0.

Data-Storage Interrupt (0x0300)

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Synchronous.
- Precise.

Description

Data-storage exceptions are associated with the execution of an instruction that accesses memory, including certain cache-control instructions. A data-storage exception occurs when a data access fails for any of the following reasons:

- An access is made to an address with *no-access-allowed* zone protection (the corresponding zone-field value is 0b00). Any load, store, **dcbf**, **dcbst**, **dcbz**, or **icbi** instruction can cause an exception for this reason. No-access-allowed zone protection is possible only in user mode with data virtual-mode enabled (MSR[DR]=1).
- A store is made to a *read-only* address. Read-only addresses can only be specified when data virtual-mode is enabled (MSR[DR]=1). Read-only addresses have the write-enable bit (TLBLO[WR]) in the corresponding TLB entry cleared to zero. The cause of this exception further depends on the privilege mode:
 - In user mode, any store or **dcbz** instruction can cause an exception for this reason. No zone-protection override can be specified (the corresponding zone-field value is *not* equal to 0b11).
 - In privileged mode, any store, **dcbi**, **dcbz**, or **dccci** instruction can cause an exception for this reason. No zone-protection override can be specified (the corresponding zone-field value is *not* equal to 0b10 or 0b11).
- A store is made to an address with the corresponding U0 storage attribute set to 1 and U0 exceptions are enabled (CCR0[U0XE]=1). In real mode, the U0 storage attribute is specified by the SU0R register. In virtual mode, the U0 storage attribute is specified by the TLB entry (TLBHI[U0]) used to translate the address. The instructions that can cause an exception for this reason are:
 - In user mode, any store or **dcbz** instruction.
 - In privileged mode, any store, **dcbi**, **dcbz**, or **dccci** instruction.

System software can use this exception condition to implement real-mode write protection.

Software cannot disable data-storage interrupts.

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the instruction that caused the data-storage exception.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	

Register	Value After Interrupt
ESR	[DST] ← 1 if the operation is a store, dcbi , dcbz , or dccci , otherwise 0. [DIZ] ← 1 if the exception was caused by a zone-protection violation, otherwise 0. [U0F] ← 1 if the exception was caused by a U0 violation, otherwise 0. [MCI] ← Unchanged. All remaining bits are cleared to 0.
DEAR	Loaded with the effective address of the failed data-access.
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] ← 0. [CE, ME, DE] ← Unchanged.

Instruction-Storage Interrupt (0x0400)

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Synchronous.
- Precise.

Description

Instruction-storage exceptions are associated with the *fetching* of an instruction from memory. However, an instruction-storage interrupt occurs only if an attempt is made to *execute* the instruction as required by the sequential-execution model. Speculative fetches that are later discarded do not cause instruction-storage interrupts. An instruction-storage exception occurs when an instruction fetch fails for any of the following reasons:

- An instruction is fetched from an address with *no-access-allowed* zone protection (the corresponding zone-field value is 0b00). No-access-allowed zone protection is possible only in user mode with instruction virtual-mode enabled (MSR[IR]=1).
- An instruction is fetched from a *non-executable* address. Non-executable addresses can only be specified when instruction virtual-mode is enabled (MSR[IR]=1). Non-executable addresses have the write-executable bit (TLBLO[EX]) in the corresponding TLB entry cleared to zero. No zone-protection override can be specified:
 - In user mode, the corresponding zone-field value is *not* equal to 0b11.
 - In privileged mode, the corresponding zone-field value is *not* equal to 0b00 or 0b11.
- An instruction is fetched from guarded storage (G attribute set to 1) regardless of privilege. In real mode, guarded storage is specified by the SGR register. In virtual mode, guarded storage is specified by the TLB entry (TLBLO[G]) used to translate the address.

Software cannot disable instruction-storage interrupts.

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the instruction that caused the instruction-storage exception.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	
ESR	[DIZ] ← 1 if the exception was caused by a zone-protection violation. [DIZ] ← 0 if the exception was caused by fetching from a non-executable address or from guarded storage. [MCI] ← Unchanged. All remaining bits are cleared to 0.
DEAR	Not used.
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] ← 0. [CE, ME, DE] ← Unchanged.

External Interrupt (0x0500)

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Asynchronous.
- Precise.

Description

An external exception is caused by an external device (usually the external-interrupt controller) asserting the noncritical-interrupt input signal to the processor.

This exception is persistent. To prevent repeated interrupts from occurring, the interrupt handler must clear the exception status in the appropriate device control register (DCR) associated with the external-interrupt controller before returning.

This interrupt is enabled using the external-interrupt enable bit (EE) in the MSR. When MSR[EE]=1, the processor recognizes exceptions caused by asserting the noncritical-interrupt input signal and forces an external interrupt to occur. When MSR[EE]=0, the processor does not recognize the noncritical-interrupt input signal and external interrupts cannot occur.

External interrupts are disabled when an external interrupt occurs. The external interrupt handler should not re-enable MSR[EE] until it has cleared the exception and saved SRR0 and SRR1. Saving these registers avoids potential corruption of the interrupt handler should an external interrupt, programmable-interval timer interrupt, or fixed-interval timer interrupt occur.

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the next-sequential instruction to be executed at the point the interrupt occurs.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	
ESR	
DEAR	
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] ← 0. [CE, ME, DE] ← Unchanged.

Alignment Interrupt (0x0600)

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Synchronous.
- Precise.

Description

Alignment exceptions are caused by the following memory accesses:

- Executing a **dcbz** instruction with an operand located in non-cacheable or write-through memory.
- Executing an **lwarx** instruction with an operand that is not aligned on a word boundary.
- Executing an **stwcx.** instruction with an operand that is not aligned on a word boundary.
- From privileged mode (MSR[PR]=0), executing a **dcread** instruction with an operand that is not aligned on a word boundary.

Software cannot disable alignment interrupts.

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the instruction that caused the alignment exception.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	
ESR	
DEAR	Loaded with the effective address of the operand that caused the alignment exception.
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] ← 0. [CE, ME, DE] ← Unchanged.

Program Interrupt (0x0700)

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Synchronous.
- Precise.

Description

Program exceptions are caused by any of the following conditions:

- Attempted execution of an illegal instruction. Floating-point instructions are considered illegal instructions in the PPC405D5.
- Attempted execution of a privileged instruction from user mode.
- Execution of a trap instruction that satisfies the trap conditions. Following execution of a trap instruction, SRR0 contains the address of the trap instruction. To avoid repeated program interrupts as a result of returning from the trap handler, software should either:
 - Replace the trap instruction with a non-trapping instruction.
 - Modify the trap conditions to prevent a program interrupt.
 - Modify the address in SRR0 to point to the next-sequential instruction in the interrupted program prior to executing the **rfi**.

The following exception conditions *do not* occur on the PPC405D5 but can occur on other versions of the PowerPC 405 processor:

- Exceptions caused by attempting to execute an unimplemented FPU or APU instruction. This exception condition sets ESR[PEU]=1.
- Exceptions caused by FPU-instruction errors. This exception condition sets ESR[PFP]=1.
- Exceptions caused by APU-instruction errors. This exception condition sets ESR[PAP]=1.

Software cannot disable program interrupts.

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the instruction that caused the program exception.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	

Register	Value After Interrupt
ESR	[PIL] $\leftarrow$ 1 for attempted execution of an illegal instruction, otherwise 0. This bit is set if software attempts to execute a floating-point instruction. [PPR] $\leftarrow$ 1 for attempted execution of a privileged instruction in user mode, otherwise 0. [PTR] $\leftarrow$ 1 for exceptions due to trap instructions, otherwise 0. [MCI] $\leftarrow$ Unchanged. All remaining bits are cleared to 0.
DEAR	Not used.
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] $\leftarrow$ 0. [CE, ME, DE] $\leftarrow$ Unchanged.

FPU-Unavailable Interrupt (0x0800)

Programs running on the PPC405D5 cannot cause this interrupt to occur because the floating-point unit is not implemented. It is shown for completeness to assist in porting software between systems containing different implementations of the PowerPC 405 processor.

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Synchronous.
- Precise.

Description

FPU-unavailable exceptions occur when a program attempts to execute an implemented floating-point instruction when the FPU is disabled (MSR[FP]=0).

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the instruction that caused the FPU-unavailable exception.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	
ESR	
DEAR	
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] ← 0. [CE, ME, DE] ← Unchanged.

System-Call Interrupt (0x0C00)

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Synchronous.
- Precise.

Description

System-call exceptions occur as a result of executing the system-call instruction (**sc**). The **sc** instruction provides a means for a user-level program to call a privileged system-service routine. It is assumed that the appropriate linkage information expected by the system-call handler is initialized prior to executing the **sc** instruction.

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the instruction following the system-call instruction.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	
ESR	
DEAR	
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] ← 0. [CE, ME, DE] ← Unchanged.

APU-Unavailable Interrupt (0x0F20)

Programs running on the PPC405D5 cannot cause this interrupt to occur because the auxiliary-processor unit is not implemented. It is shown for completeness to assist in porting software between systems containing different implementations of the PowerPC 405 processor.

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Synchronous.
- Precise.

Description

APU-unavailable exceptions occur when a program attempts to execute an implemented auxiliary-processor instruction when the APU is disabled (MSR[AP]=0).

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the instruction that caused the APU-unavailable exception.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	
ESR	
DEAR	
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] ← 0. [CE, ME, DE] ← Unchanged.

Programmable-Interval Timer Interrupt (0x1000)

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Asynchronous.
- Precise.

Description

A programmable-interval timer exception is caused by a time-out on the programmable-interval timer (PIT). A time-out occurs when:

1. The current PIT contents are 1.
2. The PIT is decremented. Decrementing the PIT when the current value is 1 can cause the PIT to be loaded either with a value of 0, or cause a new non-zero value to be automatically loaded.

When a time-out is detected, the processor sets the PIT-status bit in the timer-status register (TSR[PIS]) to 1. At the beginning on the next clock cycle, the set TSR[PIS] bit causes the PIT interrupt to occur. Using the **mtspr** instruction to clear the PIT to 0 *does not* cause a PIT interrupt.

This exception is persistent. To prevent repeated interrupts from occurring, the interrupt handler must clear the exception status in TSR[PIS] before returning.

This interrupt is enabled only by setting both of the following:

- The PIT-interrupt enable bit in the timer-control register (TCR[PIE]) must be set to 1.
- The external-interrupt enable bit in the machine-state register (MSR[EE]) must be set to 1.

If either TCR[PIE]=0 or MSR[EE]=0, a PIT interrupt does not occur. See [Chapter 8, Timer Resources](#), for more information on the PIT, TCR, and TSR.

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the next-sequential instruction to be executed at the point the interrupt occurs.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	
ESR	
DEAR	
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] ← 0. [CE, ME, DE] ← Unchanged.

The timer-status register (TSR) is also updated as a result of a PIT exception.

Register	Value After Exception
TSR	[PIS] ← 1. All others are unchanged.

Fixed-Interval Timer Interrupt (0x1010)

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Asynchronous.
- Precise.

Description

A fixed-interval timer exception is caused by a time-out on the fixed-interval timer (FIT). The processor detects a time-out when a 0 to 1 transition occurs on the time-base bit corresponding to the fixed-interval time period.

When a time-out is detected, the processor sets the FIT-status bit in the timer-status register (TSR[FIS]) to 1. At the beginning on the next clock cycle, the set TSR[FIS] bit causes the FIT interrupt to occur.

This exception is persistent. To prevent repeated interrupts from occurring, the interrupt handler must clear the exception status in TSR[FIS] before returning.

This interrupt is enabled only by setting both of the following:

- The FIT-interrupt enable bit in the timer-control register (TCR[FIE]) must be set to 1.
- The external-interrupt enable bit in the machine-state register (MSR[EE]) must be set to 1.

If either TCR[FIE]=0 or MSR[EE]=0, a FIT interrupt does not occur. See [Chapter 8, Timer Resources](#), for more information on the FIT, TCR, and TSR.

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the next-sequential instruction to be executed at the point the interrupt occurs.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	
ESR	
DEAR	
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] ← 0. [CE, ME, DE] ← Unchanged.

The timer-status register (TSR) is also updated as a result of a FIT exception.

Register	Value After Exception
TSR	[FIS] ← 1. All others are unchanged.

Watchdog-Timer Interrupt (0x1020)

Interrupt Classification

- Critical—return using the **rfei** instruction.
- Asynchronous.
- Precise.

Description

A watchdog-timer exception is caused by a time-out on the watchdog timer. For a watchdog-timer interrupt to occur, the interrupt must be enabled and the processor must be enabled to detect the watchdog-timer exception, as follows:

- The watchdog-timer interrupt is enabled only by setting both of the following:
 - The watchdog-interrupt enable bit in the timer-control register (TCR[WIE]) must be set to 1.
 - The critical-interrupt enable bit in the machine-state register (MSR[CE]) must be set to 1.

If either TCR[WIE]=0 or MSR[CE]=0, a watchdog-timer interrupt does not occur.

- The processor detects a watchdog-timer exception when:
 - The enable-next-watchdog bit in the timer-status register (TSR[ENW]) is set to 1.
 - The watchdog-interrupt status bit in the timer-status register (TSR[WIS]) is cleared to 0.
 - A 0 to 1 transition occurs on the time-base bit corresponding to the watchdog time period.

During the cycle following detection of the watchdog time-out, the processor sets TSR[WIS] to 1. At the beginning of the *next* cycle, the processor detects TSR[WIS]=1 and causes the watchdog-timer interrupt to occur.

This exception is persistent, but the persistence *prevents* further interrupts from occurring. This function causes an interrupt to occur on the first watchdog time-out, but prevents interrupts on subsequent time-outs. To enable additional interrupts, the interrupt handler must clear the exception status in TSR[WIS] before returning.

See **Chapter 8, Watchdog-Timer Events**, for more information on the watchdog timer and its relationship to the TCR and TSR.

Affected Registers

Register	Value After Interrupt
SRR0	Not used.
SRR1	
SRR2	Loaded with the effective address of the next-sequential instruction to be executed at the point the interrupt occurs.
SRR3	Loaded with a copy of the MSR at the point the interrupt occurs.
ESR	Not used.
DEAR	
MSR	[AP, APE, WE, CE, EE, PR, FP, FE0, DWE, DE, FE1, IR, DR] ← 0. [ME] ← Unchanged.

The timer-status register (TSR) is also updated as a result of a watchdog-timer interrupt.

Register	Value After Interrupt
TSR	[WIS] $\leftarrow$ 1. All others are unchanged.

Data TLB-Miss Interrupt (0x1100)

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Synchronous.
- Precise.

Description

Data TLB-miss exceptions can occur only when data translation is enabled (MSR[DR]=1). They are associated with the execution of an instruction that accesses memory, including certain cache-control instructions. A data TLB-miss exception occurs when no valid TLB entry is found with both:

- A TAG field that matches the data effective-address page number (EA[EPN]).
- A TID field that matches the current process ID (PID).

Software cannot disable data TLB-miss interrupts.

See **TLB Access**, [page 515](#), for more information on how TLB hits and misses are determined.

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the instruction that caused the data TLB-miss exception.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	
ESR	[DST] $\leftarrow$ 1 if the operation is a store, dcbi , dcbz , or dccci , otherwise 0. [MCI] $\leftarrow$ Unchanged. All remaining bits are cleared to 0.
DEAR	Loaded with the effective address of the failed data-access.
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] $\leftarrow$ 0. [CE, ME, DE] $\leftarrow$ Unchanged.

Instruction TLB-Miss Interrupt (0x1200)

Interrupt Classification

- Noncritical—return using the **rfi** instruction.
- Synchronous.
- Precise.

Description

Instruction TLB-miss exceptions can occur only when instruction translation is enabled (MSR[IR]=1). An instruction TLB-miss exception occurs when no valid TLB entry is found with both:

- A TAG field that matches the instruction effective-address page number (EA[EPN]).
- A TID field that matches the current process ID (PID).

Instruction TLB-miss exceptions are associated with the *fetching* of an instruction from memory. However, an instruction TLB-miss interrupt occurs only if an attempt is made to *execute* the instruction as required by the sequential-execution model. Speculative fetches that are later discarded do not cause instruction TLB-miss interrupts.

Software cannot disable instruction TLB-miss interrupts.

See **TLB Access**, page 515, for more information on how TLB hits and misses are determined.

Affected Registers

Register	Value After Interrupt
SRR0	Loaded with the effective address of the instruction that caused the instruction TLB-miss exception.
SRR1	Loaded with a copy of the MSR at the point the interrupt occurs.
SRR2	Not used.
SRR3	
ESR	
DEAR	
MSR	[AP, APE, WE, EE, PR, FP, FE0, DWE, FE1, IR, DR] ← 0. [CE, ME, DE] ← Unchanged.

Debug Interrupt (0x2000)

Interrupt Classification

- Critical—return using the **rfei** instruction.
- The debug interrupt can be synchronous or asynchronous, depending on the debug event:

Synchronous debug events:

- BT—Branch taken.
- DAC—Data-address compare.
- DVC—Data-value compare.
- IAC—Instruction-address compare.
- IC—Instruction completion.
- TDE—Trap instruction.

Asynchronous debug events:

- EDE—Exception taken.
- UDE—Unconditional.

- Precise.

Description

A debug exception is caused by an *enabled* debug event. Debug events are enabled and disabled using the debug-control registers (DBCR0 and DBCR1). A debug event occurs when a predefined debug condition is met, such as a data-address match.

This exception is persistent. If a debug exception occurs when debug interrupts are disabled, the imprecise-debug exception-status bit in the debug-status register is set, DBSR[IDE]. This bit is set in *addition to* other debug-status bits. When debug interrupts are later enabled, the set IDE bit causes a debug interrupt to occur immediately. When exiting an interrupt handler using an **rfei** instruction, the interrupt handler must clear DBSR[IDE] to prevent repeated interrupts from occurring. To prevent ambiguity in reporting debug status, all other DBSR bits should be cleared as well.

This interrupt is enabled using the debug-interrupt enable bit (DE) in the MSR. When MSR[DE]=1, the processor recognizes exceptions caused by enabled debug events. When MSR[DE]=0, the processor does not cause a debug interrupt when an enabled debug event occurs.

All maskable interrupts, except those caused by machine-check exceptions, are disabled when a debug interrupt occurs. The debug-interrupt handler should not re-enable MSR[DE] until it has cleared the exception and saved SRR2 and SRR3. Saving these registers avoids potential corruption of the interrupt handler should a subsequent debug interrupt occur.

See [Chapter 9, Debugging](#), for more information on debug events.

Affected Registers

Register	Value After Interrupt
SRR0	Not used.
SRR1	

Register	Value After Interrupt	
SRR2	Loaded based on the debug event, as follows:	
	BT DAC IAC TDE	Loaded with the effective address of the instruction that caused the debug exception.
	DVC IC	Loaded with the effective address of the instruction <i>following</i> the instruction that caused the debug exception.
	EDE	Loaded with the 32-bit exception-vector physical address of the <i>exception</i> that caused the debug interrupt. This corresponds to the first instruction in the interrupt handler.
	UDE	Loaded with the effective address of the next-sequential instruction to be executed at the point the debug interrupt occurs.
SRR3	Loaded with a copy of the MSR at the point the interrupt occurs.	
ESR	Not used.	
DEAR		
MSR	[AP, APE, WE, CE, EE, PR, FP, FE0, DWE, DE, FE1, IR, DR] ← 0. [ME] ← Unchanged.	

The debug-status register (DBSR) is also updated as a result of a debug interrupt. See **Debug-Status Register**, [page 577](#), for more information on the DBSR.

Register	Value After Interrupt
DBSR	Updated to reflect the debug event.

Timer Resources

The PPC405 supports several timer resources that can be used for a variety of time-keeping functions. Possible uses of these timer resources include:

- Time-of-day computation.
- Data-logging for system-service routines.
- Periodic servicing of time-sensitive external devices.
- Preemptive multitasking.

The timer resources supported by the PPC405 consist of:

- Two timer registers:
 - A 64-bit incrementing timer, called the *time-base*.
 - A 32-bit decrementing timer, called the *programmable-interval timer*.
- Three timer-event interrupts:
 - A *watchdog-timer interrupt* that provides the ability to set critical interrupts that can aid in recovery from system failures.
 - A *programmable-interval timer interrupt* that provides the ability to set noncritical variable-time interrupts.
 - A *fixed-interval timer interrupt* that provides the ability to set noncritical interrupts with a fixed, repeatable time period.
- A *timer-control register* for setting up and controlling the timer events.
- A *timer-status register* for recording timer-event status.

Figure 8-1 shows the relationship of the two timers and three timer-event interrupts. The two timers are clocked at the same frequency. This frequency is determined using external input signals to the processor. Refer to the *PPC405 Processor Block Manual* for more information on setting the timer frequency.

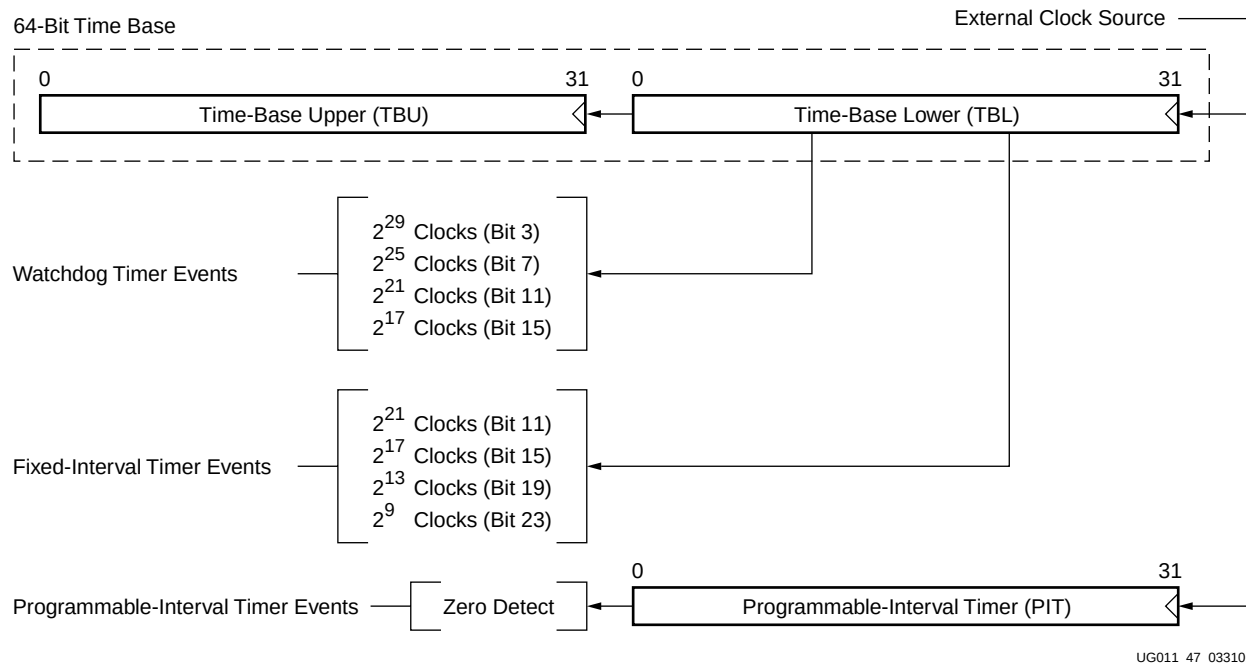


Figure 8-1: PPC405 Timer Resources

Time Base

The time base is a 64-bit incrementing counter supported by all PowerPC processors. 64 bits provide a long time period before *rolling over* from 0xFFFF_FFFF_FFFF_FFFF to 0x0000_0000_0000_0000. At a clock rate of 300 MHz, for example, the time base increments for about 1,950 *years* before rolling over. This makes it suitable for certain long-term timing functions, such as time-of-day calculation. A time-base rollover is silent—it does not cause an exception to timer event.

The 64-bit time base is implemented as two 32-bit registers. The time-base upper register (TBU) holds time-base bits 0:31, and the time-base lower register (TBL) holds time-base bits 32:63. Figure 8-2 shows the format of the time base.

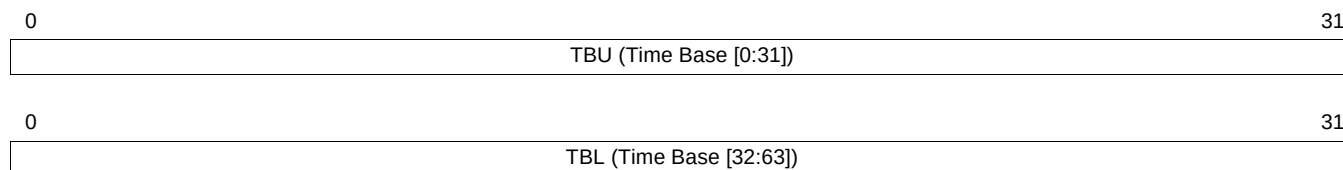


Figure 8-2: Time-Base Register

The TBU and TBL registers are SPRs with user-mode read access and privileged-mode write access. Reading the time-base registers requires use of the *move from time-base register* instruction. This instruction, shown in Table 8-1, is similar to the *move from SPR* instruction. The TBR number (TBRN) shown in the operand syntax column can be specified as a decimal or hexadecimal value in the assembler listing. Within the instruction opcode, this number is encoded using a *split-field notation* (see **Split-Field Notation**, page 607).

Table 8-1: Time-Base Register Instructions

Mnemonic	Name	Operation	Operand Syntax
mftb	Move from Time Base Register	This instruction provides read-only access from the time base for user and privileged software. rD is loaded with the contents of the time-base register specified by TBRN.	rD,TBRN
mtspr	Move to Special Purpose Register	This instruction provides write-only access to the time base for privileged software. The time-base register specified by SPRN is loaded with the contents of rS.	SPRN,rS

Table 8-2 summarizes the time-base numbers and SPR numbers used by the above instructions to access the time base registers. Simplified instruction mnemonics are available for reading and writing the time base. See **Special-Purpose Registers**, [page 866](#), for more information.

Table 8-2: Time-Base Register Numbers

Register	Decimal	Hex	Access
TBL	268	0x10C	User and privileged read-only— mftb .
TBU	269	0x10D	
TBL	284	0x11C	Privileged write-only— mtspr .
TBU	285	0x11D	

Reading and Writing the Time Base

The 64-bit time-base cannot be read or written using a single instruction. Software must access the upper and lower portions separately. During the time it takes to execute the instructions necessary to access the time base, it is possible for the TBU to be incremented. This occurs when TBL rolls over from 0xFFFF_FFFF to 0x0000_0000 (at 300 MHz, this happens every 14.3 seconds). If there is a rollover, the values read from or written to TBU and TBL can be inconsistent.

Following is a code example for reading the time base. The comparison of old and new TBU values within the loop ensures that a consistent pair of TBU and TBL values are read, avoiding problems with TBL rollover.

```

loop:
mftbu rx      ! Read TBU.
mftb  ry      ! Read TBL.
mftbu rz      ! Read TBU again.
cmpw  rz,rx    ! Check for TBU rollover by comparing old and new.
bne   loop     ! Read the time base again if a rollover occurred.

```

Following is a code example for writing the time base (simplified mnemonics are used for writing the time-base registers). Clearing TBL to 0 before writing it with a non-zero value ensures TBL rollover does not occur in the brief time required to update both TBU and TBL.

```

lwz   rx,upper_value ! Load upper 32-bit time-base value into rx.
lwz   ry,lower_value ! Load lower 32-bit time-base value into ry.
li    rz, 0           ! Clear rz.
mttbl rz              ! Clear TBL to avoid rollover after writing TBU.
mttbu rx              ! Update TBU.
mttbl ry              ! Update TBL.

```

Computing Time of Day

Calculating the time-of-day from the current time-base value requires the following information:

- A fixed-reference time.
- The equivalent time-base value corresponding to the fixed reference time.
- The system-dependent time-base update frequency.

Following is an algorithm that calculates the time-of-day. Awkward 64-bit division is avoided by assuming the algorithm is initiated by a time-keeping interrupt *at least* once per second. This periodic interrupt can be triggered by the fixed-interval timer or some external-interrupt device. The algorithm uses the following variables:

- *billion*—one billion (1,000,000,000).
 - *posix_tb*—A 64-bit variable containing the last value read from the time base.
 - *posix_sec*—A 32-bit variable containing the number of seconds that have elapsed since the fixed-reference time. When timekeeping actually begins, this variable must be initialized with the number of seconds that have elapsed from the fixed-reference time. For example, assume:
 - The fixed-reference time is 12:00:00 AM, January 1, 2001
 - The equivalent time-base value for the fixed-reference time is 0.
 - Timekeeping begins at 12:00:00 AM, July 1, 2001.
- Using these parameters, this variable is initialized with 0x00EE_9F80, which represents the number of seconds that have elapsed since the fixed-reference time.
- *posix_ns*—A 32-bit variable containing the number of nanoseconds that have elapsed since the last time-of-day calculation.
 - *ticks_per_sec*—The number of times the time base increments per second. In this example, the processor clock is 300 MHz and the time base is incremented once every 32 processor clocks. Thus, the variable is set to 0x8F_0D18 (300 MHz ÷ 32 = 9,375,000).
 - *ns_adj*—The number of nanoseconds per increment of the time base. In this example, the variable is set to 0x6B (*billion* ÷ *ticks_per_sec* = 107).

The following code sequence implements the algorithm:

```

loop:
mftbu rx          ! Read TBU.
mftb  ry          ! Read TBL.
mftbu rz          ! Read TBU again.
cmpw  rz, rx      ! Check for TBU rollover by comparing old and new.
bne   loop        ! Read the time base again if a rollover occurred.

! We now have a consistent 64-bit time base in rx and ry.

lwz   rz, posix_tb+4 ! Load rz with the low-32 bits of posix_tb.
sub   rz, ry, rz      ! rz = change in TB since last read.
lwz   rw, ns_adj      ! Load the number of ns per time-base increment.
mullw rz, rz, rw      ! rz = number of elapsed ns since TB last read.
lwz   rw, posix_ns    ! Load elapsed ns since last computation.
add   rz, rz, rw      ! rz = new ns since last computation.
lwz   rw, billion     ! A billion nanoseconds is 1 second.
cmpw  rz, rw          ! Are new elapsed ns more than 1 second?
blt   nochange        ! Branch if not.
sub   rz, rz, rw      ! Subtract 1 second from elapsed nanoseconds.
lwz   rw, posix_sec   ! Load the number of elapsed seconds.
addi  rw, rw, 1       ! Add 1 second.
stw   rw, posix_sec   ! Store the number of elapsed seconds.
nochange:
stw   rz, posix_ns    ! Update elapsed ns.

```

```
stw    rx, posix_tb    ! Update record of last time-base value.
stw    ry, posix_tb+4
```

Timekeeping software can use the `posix_sec` value to compute the current date and time by adding it to the fixed reference time.

Varying the Update Frequency

Time-of-day computations require a comparison between the current time-base value and a fixed-reference time. This reference time is valid only when the time-base update frequency remains fixed. Many embedded systems change the time-base update frequency periodically. Changes are often initiated by system software, but hardware can also cause a frequency change (for example, a low-power mode that is initiated by a sudden power failure). When the frequency changes, a mechanism must be provided to the time-of-day calculation routine notifying it of the change. If the change is software initiated, a system call to the calculation routine can be used. If the change is hardware initiated, an external interrupt can be used.

When the time-of-day calculation routine is called, it must compute new reference values. This involves the following:

- Saving the time-base value at the point the frequency is changed.
- Computing and saving the current time-of-day using the old update frequency and the saved time-base value.
- Computing and saving a new value for `ticks_per_sec`.

Later calls to compute the time-of-day can use the updated variables along with the current time-base value to calculate the correct time.

Timer-Event Registers

Three PPC405 registers are defined for managing timer-event interrupts:

- Programmable-interval timer register.
- Timer-control register.
- Timer-status register.

A description of each register is provided in the following sections.

Programmable-Interval Timer Register

The programmable-interval timer (PIT) register is a 32-bit decrementing counter that is clocked at the same frequency as the time-base register. It can be used by software to cause a PIT interrupt after a variable-length time period elapses. [Figure 8-3](#) shows the format of the PIT register.

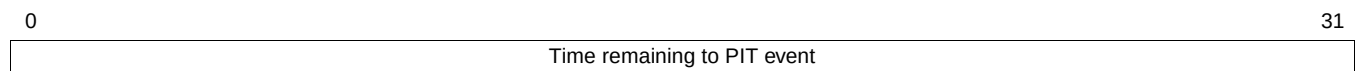


Figure 8-3: Programmable-Interval Timer Register (PIT)

The PIT is a privileged SPR with an address of 987 (0x3DB). It is read and written using the `mf spr` and `mt spr` instructions.

When the PIT contains a value of 1 and is decremented, a PIT event occurs. A PIT event can be used to cause a PIT interrupt as described in [Programmable-Interval Timer Events, page 568](#). Auto-reload mode controls the state of the PIT register when it contains a value of 1 and is decremented, as follows:

- In auto-reload mode, the PIT is reloaded with the last value loaded by an `mt spr` instruction. In this mode, the PIT never contains a value of 0. Auto-reload mode is enabled by setting the auto-reload enable bit in the timer-control register

(TCR[ARE]=1).

- If auto-reload mode is disabled (TCR[ARE]=0), the PIT decrements from 1 to 0. When the PIT contains a value of 0, it stops decrementing until software loads it with a non-zero value.

Auto-reload mode is disabled after a reset.

Timer-Control Register

The timer-control register (TCR) is a 32-bit register used to control the PPC405 timer events. **Figure 8-4** shows the format of the TCR. The fields in TCR are defined as shown in **Table 8-3**.

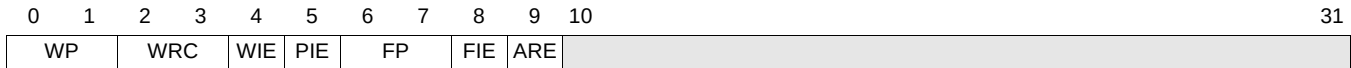


Figure 8-4: Timer-Control Register (TCR)

The TCR is a privileged SPR with an address of 986 (0x3DA). It is read and written using the **mfspr** and **mtspr** instructions.

Table 8-3: Timer-Control Register (TCR) Field Definitions

Bit	Name	Function	Description
0:1	WP	Watchdog Period 00— 2^{17} clocks 01— 2^{21} clocks 10— 2^{25} clocks 11— 2^{29} clocks	Specifies the period for a watchdog-timer event.
2:3	WRC	Watchdog Reset Control 00—No reset 01—Processor reset 10—Chip reset 11—System reset	Specifies the type of reset that occurs as a result of a watchdog-timer event. After a bit is set in the WRC field, it cannot be cleared by software. Only a reset can clear the bit. This prevents errant code from disabling watchdog resets.
4	WIE	Watchdog-Interrupt Enable 0—Disabled 1—Enabled	Enables and disables watchdog interrupts.
5	PIE	PIT-Interrupt Enable 0—Disabled 1—Enabled	Enables and disables programmable-interval timer interrupts.
6:7	FP	FIT Period 00— 2^9 clocks 01— 2^{13} clocks 10— 2^{17} clocks 11— 2^{21} clocks	Specifies the period for a fixed-interval timer event.
8	FIE	FIT-Interrupt Enable 0—Disabled 1—Enabled	Enables and disables fixed-interval timer interrupts.
9	ARE	Auto-Reload Enable 0—Disabled 1—Enabled	Enables and disables the programmable-interval timer auto-reload mode.
10:31		Reserved	

Timer-Status Register

The timer-status register (TSR) is a 32-bit register used to report status for the PPC405 timer events. [Figure 8-5](#) shows the format of the TSR. The fields in TSR are defined as shown in [Table 8-4](#).



Figure 8-5: Timer-Status Register (TSR)

The TSR is a privileged SPR with an address of 984 (0x3D8). Hardware sets the status bits. Software is responsible for reading and clearing the bits. It is read using the **mfsprr** instruction. The register is cleared, but not directly written, using the **mtspr** instruction. Values in the source register, **rS**, behave as a mask when clearing the TSR. Here, a value of 0b1 in any bit position of **rS** *clears* the corresponding bit in the TSR. A value of 0b0 in an **rS** bit position does not alter the corresponding bit in the TSR.

Table 8-4: Timer-Status Register (TSR) Field Definitions

Bit	Name	Function	Description
0	ENW	Enable Next Watchdog 0—Next watchdog time-out sets TSR[ENW]=1 1—Next watchdog time-out determined by TSR[WIS]	Enables the watchdog-timer event. See Watchdog-Timer Events , page 566 , for more information.
1	WIS	Watchdog-Interrupt Status 0—No interrupt occurred 1—Interrupt occurred	Specifies whether a watchdog interrupt occurred, or could have occurred had it been enabled.
2:3	WRS	Watchdog Reset Status 00—No reset 01—Processor reset 10—Chip reset 11—System reset	Specifies the type of reset that occurred as a result of a watchdog-timer event, if the event caused a reset.
4	PIS	PIT-Interrupt Status 0—No interrupt pending 1—Interrupt is pending	If programmable-interval timer interrupts are disabled, this bit specifies whether a PIT interrupt is pending. If they are enabled, the bit specifies whether a PIT interrupt occurred.
5	FIS	FIT-Interrupt Status 0—No interrupt pending 1—Interrupt is pending	If fixed-interval timer interrupts are disabled, this bit specifies whether a FIT interrupt is pending. If they are enabled, the bit specifies whether a FIT interrupt occurred.
6:31		Reserved	

Timer-Event Interrupts

Three timer-event interrupts are defined by the PPC405. Each interrupt transfers control to a unique exception-vector offset (see [Interrupt Reference](#), [page 538](#), for more information):

- Watchdog-timer (WDT) interrupt. This critical interrupt is assigned to exception-vector offset 0x1020.
- Programmable-interval timer (PIT) interrupt. This noncritical interrupt is assigned to exception-vector offset 0x1000.
- Fixed-interval timer (FIT) interrupt. This noncritical interrupt is assigned to exception-vector offset 0x1010.

The following sections describe the use of the timer-event registers in managing the interrupts.

Watchdog-Timer Events

The watchdog timer can aid in recovery from software or hardware failure. It can be programmed to cause a watchdog time-out (also called the watchdog event) after a fixed time-period elapses. Watchdog time-outs can be further programmed to cause a critical interrupt called the watchdog interrupt. Normally, the watchdog-interrupt handler clears the watchdog event before returning. However, if a software or hardware failure prevents the interrupt handler from clearing the event, a subsequent watchdog time-out can be programmed to force a reset.

Watchdog interrupts are enabled when *both* of the following bits are set to 1:

- The watchdog-interrupt enable bit in the timer-control register, TCR[WIE].
- The critical-interrupt enable bit in the machine-state register, MSR[CE].

If either TCR[WIE]=0 or MSR[CE]=0, watchdog-timer interrupts are disabled. However, watchdog time-outs can be programmed to force a reset whether or not the watchdog interrupt is enabled.

A watchdog time-out occurs when a selected bit in the time-base lower register (TBL) changes from 0 to 1. The watchdog-period bit in the timer-control register (TCR[WP]) is used to select the TBL bit that controls the time-out, as shown in Table 8-5.

Table 8-5: Watchdog Time-Out Periods

TCR[WP]	Selected TBL Bit	Time-Base Clock Period	Watchdog Period (300 MHz Clock)
00	15	2^{17}	0.437 msec
01	11	2^{21}	6.99 msec
10	7	2^{25}	0.112 sec
11	3	2^{29}	1.79 sec

Software cannot disable watchdog time-outs. This is because the time-base register is always incrementing and a valid watchdog interval is always specified by TCR[WP]. Instead of preventing watchdog time-outs, software controls the action taken by the processor when a time-out occurs by managing the watchdog-event state machine. A timer-control register field and two timer-status register bits are used to control the state machine:

- *Watchdog-reset control*, TCR[WRC]—This field specifies the type of reset to be performed when the state machine enters the reset state:
 - 00—No reset. The processor ignores the watchdog time-out.
 - 01—A processor-only reset occurs. No external devices are reset.
 - 10—A chip reset occurs. The processor and all external devices on the same chip are reset. No other system components are reset.
 - 11—The entire system, including the processor and chip, are reset.

Each bit in TCR[WRC] is sticky. Software can set these bits but cannot clear them. After a bit is set only a reset can clear it.

- *Enable next watchdog*, TSR[ENW]—This bit performs the following functions:
 - When cleared to 0, the TSR[WIS] bit is not updated or used by the processor. Watchdog time-outs cannot cause an interrupt or reset. The next watchdog time-out sets this bit to 1.

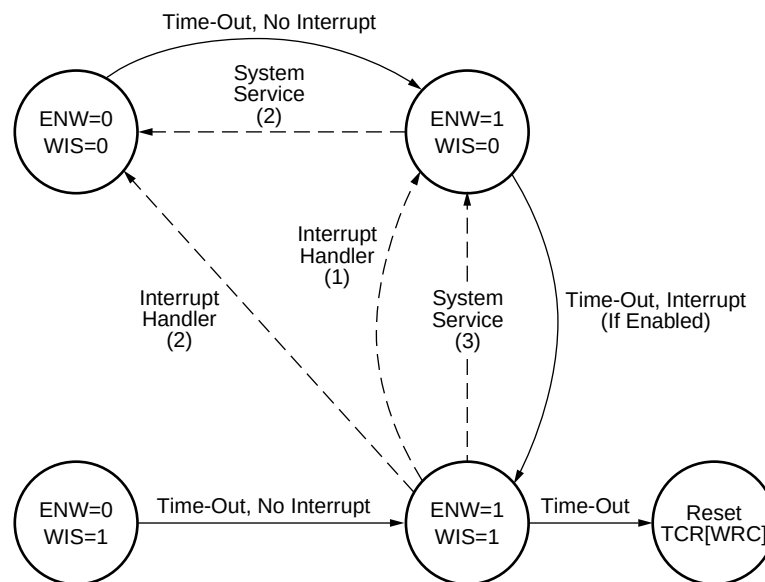
- When set to 1, the TSR[WIS] bit can be updated and is used by the processor as described below. When TSR[ENW]=1, the next watchdog time-out causes a watchdog interrupt (if enabled) or forces a reset (if a reset is specified). The value of the TSR[WIS] bit determines whether the action taken is an interrupt or a reset. In this case, the watchdog time-out that causes an interrupt is often referred to as the *second watchdog time-out*.

The processor sets the TSR[ENW] bit but never clears it. Only software can clear the bit.

- *Watchdog-interrupt status, TSR[WIS]*—This bit is used by the processor only when TSR[ENW]=1. It indicates whether or not a watchdog interrupt occurred and controls further watchdog interrupts and reset, as follows:
 - When cleared to 0, no watchdog interrupt occurred. The next watchdog time-out can cause a watchdog interrupt to occur, if the interrupt is enabled. When TSR[ENW]=1, the next time-out sets this bit to 1.
 - When set to 1, a watchdog interrupt occurred or would have occurred if enabled. The next watchdog time-out forces a reset if a reset condition is specified by TCR[WRC].

The processor sets the TSR[WIS] bit but never clears it. Only software can clear the bit.

Figure 8-6 shows the watchdog-event state machine and the transitions described in the previous paragraphs. The transitions for the interrupt handler and system service routines (both shown as dashed lines) are described in the following paragraphs.



UG011_48_033101

Figure 8-6: Watchdog-Event State Machine

Watchdog time-outs can be used to recover from otherwise unrecoverable errors. In the absence of software intervention, consecutive watchdog time-outs can cause a reset under the control of TCR[WRC]. This happens when the watchdog-event state machine enters the “Reset” state shown in Figure 8-6. After a reset, system software can determine the cause of the unrecoverable error and take appropriate action.

If no errors occur, software must periodically update the state of the state machine to prevent a reset. Figure 8-6, shows three possible methods for properly managing the state machine:

- Method (1)—an interrupt handler manages the state machine.

This method uses the watchdog interrupt. The watchdog interrupt handler clears TSR[WIS]=0 before returning. TSR[ENW] is never cleared and is always set to 1. If an error prevents watchdog interrupts, consecutive watchdog time-outs force a reset.

- Method (2)—the combination of a system-service routine and an interrupt handler manages the state machine.

This method attempts to avoid watchdog interrupts. Here, a system-service routine periodically clears the TSR[ENW] bit to 0, preventing watchdog interrupts. The system service routine must run more frequently than the watchdog time-out period. The fixed-interval timer can be used to initiate this routine at the proper time interval.

If an error prevents the system-service routine from clearing TSR[ENW], the next watchdog time-out causes a watchdog interrupt. The interrupt handler can attempt to correct the problem and clear both TSR[ENW] and TSR[WIS]. If an error prevents watchdog interrupts, another watchdog time-out forces a reset.

- Method (3)—a system-service routine manages the state machine.

This method avoids the watchdog interrupt entirely and requires that the interrupt be disabled. A system-service routine periodically clears the TSR[WIS] bit to 0 and leaves TSR[ENW] set to 1. If an error prevents the system-service routine from clearing TSR[WIS], the next watchdog time-out causes a reset. As with method (2), the system service routine must run more frequently than the watchdog time-out period. The fixed-interval timer can be used to initiate this routine at the proper time interval.

Disabling Watchdog Time-outs

After a reset (including power-on reset), watchdog interrupts are disabled because MSR[CE]=0. However, watchdog time-outs continue to occur because the time-base register is always incrementing, and a valid watchdog interval is always specified by TCR[WP].

Unless prevented by software, consecutive watchdog time-outs cause the state machine to enter the “Reset” state shown in [Figure 8-6](#). If the state machine enters the “Reset” state and TCR[WRC]=00 (the value following a reset), watchdog time-outs become silent, causing neither an interrupt or reset. This effectively disables the event.

Programmable-Interval Timer Events

The programmable-interval timer (PIT) is a 32-bit decrementing register that is clocked at the same frequency as the time-base register. The PIT begins decrementing when it is loaded with a non-zero value and it stops decrementing when the contents reach 0. When the PIT contains a value of 1 and is decremented, a PIT event occurs. The value in the PIT following a PIT event depends on whether auto-reload mode is enabled:

- If auto-reload is not enabled (TCR[ARE]=0), the next PIT value is 0 and decrementing is halted. Loading the PIT with a value of 0 does not cause a PIT event.
- If auto-reload is enabled (TCR[ARE]=1), the PIT is loaded with the last value written to it. Decrementing continues from that value.

A PIT event causes a PIT interrupt when *both* of the following bits are set to 1:

- The PIT interrupt-enable bit in the timer-control register, TCR[PIE].
- The external-enable bit in the machine-state register, MSR[EE].

PIT events always set the PIT-interrupt status bit in the timer-status register (TSR[PIS]=1). This happens whether or not PIT interrupts are enabled. If TSR[PIS]=1 and the PIT interrupt is disabled, the PIT interrupt is pending. A PIT interrupt occurs if the status bit is set and the interrupt is enabled.

PIT events are disabled as follows:

- Disable PIT interrupts by clearing TCR[PIE]=0.
- Clear TSR[PIS] to 0 to remove pending PIT interrupts.

- Halt PIT decrementing by loading the PIT with 0. Alternatively, auto-reload mode can be disabled by clearing TCR[ARE]=0. When the PIT reaches to 0, decrementing is halted.

Fixed-Interval Timer Events

A fixed-interval timer (FIT) event occurs when a selected bit in the time-base lower register (TBL) changes from 0 to 1. The FIT-period bit in the timer-control register (TCR[FP]) is used to select the TBL bit controlling the FIT event, as shown in [Table 8-6](#).

Table 8-6: Fixed-Interval Timer-Event Periods

TCR[FP]	Selected TBL Bit	Time-Base Clock Period	FIT Period (300 MHz Clock)
00	23	2^9	1.71 μ sec
01	19	2^{13}	27.3 μ sec
10	15	2^{17}	0.437 msec
11	11	2^{21}	6.99 msec

Software cannot prevent FIT events from occurring. This is because the time-base register is always incrementing and a valid fixed interval is always specified by TCR[FP].

A FIT event causes a FIT interrupt when *both* of the following bits are set to 1:

- The FIT interrupt-enable bit in the timer-control register, TCR[FIE].
- The external-enable bit in the machine-state register, MSR[EE].

FIT events always set the FIT-interrupt status bit in the timer-status register (TSR[FIS]=1). This happens whether or not FIT interrupts are enabled. If TSR[FIS]=1 and the FIT interrupt is disabled, the interrupt is considered pending. A FIT interrupt occurs if the status bit is set and the interrupt is enabled.

To disable FIT interrupts, software must clear TCR[FIE]=0. TSR[FIS] should be cleared to 0 to remove pending FIT interrupts.

Debugging

The PPC405 debugging resources can be used by system software and external hardware to implement software debug and trace-capture tools (collectively referred to as *debuggers*). These resources provide the following capabilities:

- Debug modes that support various debug tools and debug tasks commonly used in embedded-systems development.
- A debug exception (vector offset 0x2000) for use by debuggers when debug events occur.
- A variety of debugging functions (not all functions are available from all debug modes):
 - *Debug Events*—Several types of debug events are available from the various debug modes. When detected, debug events can cause an interrupt or stop the processor, depending on the debug mode.
 - *Trap Instructions*—The trap instructions (**tw** and **twi**) can be used to set software breakpoints that cause debug events rather than program interrupts.
 - *Halt*—An external debug signal can be used to *halt* (stop) the processor. No instructions are executed during a halt, but processor registers can be read and written using the JTAG port. Execution resumes when the external halt signal is de-asserted.
 - *Stop*—Stop can be used to halt the processor using the JTAG port rather than the external halt signal. No instructions are executed during a halt, but processor registers can be read and written using the JTAG port.
 - *Instruction Step*—Using the JTAG port, the processor can be stopped and *single-stepped* one instruction at a time.
 - *Instruction Stuff*—Using the JTAG port, the processor can be stopped and instructions can be inserted (stuffed) into the processor and executed. The instructions do not replace existing instruction.
 - *Freeze Timers*—The JTAG port or a debug-control register can be used to control the PPC405 timer resources. The timers can be frozen (stopped) completely, frozen only for the duration of debug events, or left running.
 - *Reset*—A processor, chip, or system reset can be forced using the JTAG port, a debug-control register, or external signalling.
- Control registers used to manage the debug modes and functions.
- Status registers used to report debug information.
- Status reporting through the JTAG port, including:
 - *Execution Status*—Indicates whether the processor is stopped, waiting, or running.
 - *Exception Status*—Indicates the status of pending synchronous exceptions.
 - *Most Recent Reset*—Indicates the cause of the most-recent reset.
- A debug interface (JTAG) and a trace interface for connecting external hardware and software debug tools.

Debug Modes

The PPC405 supports the following four debug modes:

- Internal-debug mode for use by software debuggers.
- External-debug mode for use by JTAG debuggers.
- Debug-wait mode for interrupt servicing when a JTAG debugger is in use.
- Real-time trace mode for use by instruction-trace tools.

The internal-debug and external-debug modes can be enabled simultaneously. Debug-wait mode and real-time trace mode are available only when both the internal-debug and external-debug modes are disabled.

Internal-Debug Mode

Internal-debug mode is used during normal program execution and provides an effective means for debugging system software and application programs. The mode supports setting breakpoints and monitoring processor status. In this mode, debug events can cause debug interrupts. The debug-interrupt handler is used to collect status information and to alter software-visible resources.

Internal-debug mode is enabled by setting the internal-debug mode bit in debug-control register 0, DBCR0[IDM]=1. Debug interrupts are enabled by setting MSR[DE]=1. An internal debug event can cause a debug interrupt only when both DBCR0[IDM]=1 and MSR[DE]=1.

External-Debug Mode

External-debug mode can be used to alter normal program execution. It provides the ability to debug system hardware as well as software. The mode supports starting and stopping the processor, single-stepping instruction execution, setting breakpoints, and monitoring processor status. Access to processor resources is provided through the JTAG port.

External-debug events stop the processor, halting instruction execution. External-bus activity continues when the processor is stopped. Processor resources are accessed through the JTAG port when the processor is stopped. External-debug mode also enables instructions to be stuffed (inserted) into the processor through the JTAG port and executed. This capability does not cause privileged (program) exceptions, so privileged instructions can be stuffed when the processor is in user mode.

Instructions stuffed into the processor can provide access to a variety of system resources, including DCRs and system memory. However, memory-protection mechanisms continue to operate in external-debug mode. Debug software can modify the MSR or TLB entries as necessary to enable access into protected memory locations.

External-debug mode is enabled by setting the external-debug mode bit in debug-control register 0, DBCR0[EDM]=1.

Debug events in external-debug mode can cause debug interrupts if internal-debug mode is also enabled. Here, the processor stops with a debug-interrupt pending. The external debugger can perform debug operations and restart the processor. When the processor is restarted the debug interrupt occurs, transferring control to the debug-interrupt handler. The handler can be used to collect processor-status information and to alter software-visible resources. An external debug event can cause a debug interrupt only when both DBCR0[IDM]=1 and MSR[DE]=1.

Debug-Wait Mode

Debug-wait mode causes the processor to enter a state in which interrupts can be handled when the processor appears to be stopped. The mode operates in a fashion similar to external-debug mode. It supports starting and stopping the processor, single-stepping instruction execution, setting breakpoints, and monitoring processor status. Access to processor resources is provided through the JTAG port.

External-debug events stop the processor, halting instruction execution. External-bus activity continues when the processor is stopped. Processor resources are accessed through the JTAG port when the processor is stopped. External-debug mode also enables instructions to be stuffed (inserted) into the processor through the JTAG port and executed. This capability does not cause privileged (program) exceptions, so privileged instructions can be stuffed when the processor is in user mode.

Unlike external-debug mode, debug-wait mode enables external devices to interrupt the processor when it is stopped. The processor transfers control to the critical-input interrupt handler (0x0100) or the external-interrupt handler (0x0500), as appropriate. After the interrupt handler completes and executes a return-from-interrupt instruction, the processor re-enters the stopped state.

Debug-wait mode is enabled by setting the debug-wait mode bit in the MSR, MSR[DWE]=1. Internal-debug mode and external debug mode must both be disabled (DBCR0[IDM]=0 and DBCR0[EDM]=0).

Real-Time Trace-Debug Mode

Real-time trace-debug mode supports real-time tracing of the instruction stream executed by the processor. In this mode, debug events are used to cause external trigger events. An external trace tool uses the trigger events to control the collection of trace information. The broadcast of trace information occurs independently of external trigger events (trace information is always supplied by the processor). Real-time trace-debug does not affect processor performance.

Real-time trace-debug mode is always enabled. However, the trigger events occur only when both internal-debug mode and external debug mode are disabled (DBCR0[IDM]=0 and DBCR0[EDM]=0). Most trigger events are blocked when either of those two debug modes are enabled.

Information on the trace-debug capabilities, how trace-debug works, and how to connect an external trace tool is available in the *RISCWatch Debugger User's Guide*.

Debug Registers

The PPC405 debug resources include the following registers:

- Debug-control registers (DBCR0 and DBCR1).
- Debug-status register (DBSR).
- Instruction address-compare registers (IAC1–IAC4).
- Data address-compare registers (DAC1–DAC2).
- Data value-compare registers (DVC1–DVC2).

A description of each register is provided in the following sections.

Debug-Control Registers

Two debug-control registers are supported by the PPC405: DBCR0 and DBCR1.

Debug-control register 0 (DBCR0) is used to enable the debug modes. It also is used to enable instruction-complete, branch-taken, exception-taken, and trap-instruction debug events. It controls the various features of the instruction address-compare debug event. DBCR0 is also used to freeze the timers during a debug event. **Figure 9-1** shows the format of the DBCR0 register. The fields in the DBCR0 are defined as shown in **Table 9-1**.

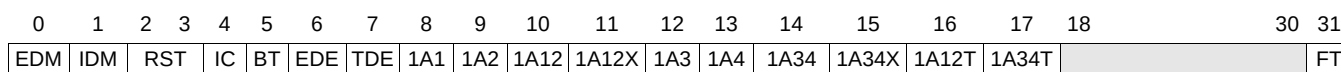


Figure 9-1: Debug-Control Register 0 (DBCR0)

Table 9-1: Debug-Control Register 0 (DBCR0) Field Definitions

Bit	Name	Function	Description
0	EDM	External-Debug Mode 0—Disabled 1—Enabled	Specifies whether or not external-debug mode is enabled.
1	IDM	Internal-Debug Mode 0—Disabled 1—Enabled	Specifies whether or not internal-debug mode is enabled.
2:3	RST	Reset 00—No reset 01—Processor reset 10—Chip reset 11—System reset	Causes the specified reset to occur when written. The reset occurs immediately after the processor recognizes the value written to the register.
4	IC	Instruction-Complete Debug Event 0—Disabled 1—Enabled	Specifies whether or not the instruction-complete debug event is enabled.
5	BT	Branch-Taken Debug Event 0—Disabled 1—Enabled	Specifies whether or not the branch-taken debug event is enabled.
6	EDE	Exception-Taken Debug Event 0—Disabled 1—Enabled	Specifies whether or not the exception debug event is enabled.
7	TDE	Trap-Instruction Debug Event 0—Disabled 1—Enabled	Specifies whether or not the trap debug event is enabled.
8	IA1	Instruction Address-Compare 1 Debug Event 0—Disabled 1—Enabled	Specifies whether or not the instruction address-compare 1 (IAC1) debug event is enabled.
9	IA2	Instruction Address-Compare 2 Debug Event 0—Disabled 1—Enabled	Specifies whether or not the instruction address-compare 2 (IAC2) debug event is enabled.

Table 9-1: Debug-Control Register 0 (DBCR0) Field Definitions (Continued)

Bit	Name	Function	Description
10	IA12	Instruction-Address Range-Compare 1-2 0—Disabled 1—Enabled	Instruction address-compare registers IAC1 and IAC2 specify an address range used by either the IAC1 or IAC2 debug events. If address-range comparison is disabled, exact-address comparison is enabled.
11	IA12X	IA12 Range-Compare Exclusive 0—Inclusive 1—Exclusive	Specifies whether the 1A12 address range (enabled by bit 10) is an inclusive range or an exclusive range.
12	IA3	Instruction Address-Compare 3 Debug Event 0—Disabled 1—Enabled	Specifies whether or not the instruction address-compare 3 (IAC3) debug event is enabled.
13	IA4	Instruction Address-Compare 4 Debug Event 0—Disabled 1—Enabled	Specifies whether or not the instruction address-compare 4 (IAC4) debug event is enabled.
14	IA34	Instruction-Address Range-Compare 3-4 0—Disabled 1—Enabled	Instruction address-compare registers IAC3 and IAC4 specify an address range used by either the IAC3 or IAC4 debug events. If address-range comparison is disabled, exact-address comparison is enabled.
15	IA34X	IA34 Range-Compare Exclusive 0—Inclusive 1—Exclusive	Specifies whether the 1A34 address range (enabled by bit 14) is an inclusive range or an exclusive range.
16	IA12T	IA12 Range-Compare Toggle 0—No toggle 1—Toggle.	Toggles the value of the 1A12X bit (bit 11) from 1 to 0 or 0 to 1 when a debug event caused by a IA12 range comparison (bit 10) occurs.
17	IA34T	IA34 Range-Compare Toggle 0—No toggle 1—Toggle.	Toggles the value of the 1A34X bit (bit 15) from 1 to 0 or 0 to 1 when a debug event caused by a IA34 range comparison (bit 14) occurs.
18:30		Reserved	
31	FT	Freeze Timers on Debug Event 0—Do not freeze 1—Freeze	Specifies whether the timers are frozen when a debug event occurs.

The DBCR0 is a privileged SPR with an address of 1010 (0x3F2) and is read and written using the **mfspr** and **mtspr** instructions.

Debug-control register 1 (DBCR1) is used to enable the parameters governing the various data address-compare and data value-compare debug events.

Figure 9-2 shows the format of the DBCR1 register. The fields in the DBCR1 are defined as shown in **Table 9-2**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	19	20	23	24	31
D1R	D2R	D1W	D2W	D1S	D2S	DA12	DA12X			DV1M	DV2M	DV1BE	DV2BE								

Figure 9-2: Debug-Control Register 1 (DBCR1)

Table 9-2: Debug-Control Register 1 (DBCR1) Field Definitions

Bit	Name	Function	Description
0	D1R	Data Address-Compare 1 Read Debug Event 0—Disabled 1—Enabled	Specifies whether or not the data address-compare 1 (DAC1) debug event is enabled for reads.
1	D2R	Data Address-Compare 2 Read Debug Event 0—Disabled 1—Enabled	Specifies whether or not the data address-compare 2 (DAC2) debug event is enabled for reads.
2	D1W	Data Address-Compare 1 Write Debug Event 0—Disabled 1—Enabled	Specifies whether or not the data address-compare 1 (DAC1) debug event is enabled for writes.
3	D2W	Data Address-Compare 2 Write Debug Event 0—Disabled 1—Enabled	Specifies whether or not the data address-compare 2 (DAC2) debug event is enabled for writes.
4:5	D1S	Data Address-Compare 1 Size 00—Compare all bits 01—Ignore least-significant bit 10—Ignore least-significant two bits 11—Ignore least-significant five bits	Specifies the granularity of DAC1 exact-address comparisons: 00—Byte granular 01—Halfword granular 10—Word granular 11—Cache-line (8-byte) granular
6:7	D2S	Data Address-Compare 2 Size 00—Compare all bits 01—Ignore least-significant bit 10—Ignore least-significant two bits 11—Ignore least-significant five bits	Specifies the granularity of DAC2 exact-address comparisons: 00—Byte granular 01—Halfword granular 10—Word granular 11—Cache-line (8-byte) granular
8	DA12	Data-Address Range-Compare 1-2 0—Disabled 1—Enabled	Data address-compare registers DAC1 and DAC2 specify an address range used by either the DAC1 or DAC2 debug events. If address-range comparison is disabled, exact-address comparison is enabled.
9	DA12X	DA12 Range-Compare Exclusive 0—Inclusive 1—Exclusive	Specifies whether the DA12 address range (enabled by bit 8) is an inclusive range or an exclusive range.
10:11		Reserved	
12:13	DV1M	Data-Value Compare 1 Mode 00—Undefined 01—All selected bytes must match 10—At least one selected byte must match 11—At least one selected halfword must match	Specifies the conditions under which a data value-comparison with the DVC1 register causes a debug event (DVC1 event). The comparison is made using the bytes selected by DV1BE.
14:15	DV2M	Data-Value Compare 2 Mode 00—Undefined 01—All selected bytes must match 10—At least one selected byte must match 11—At least one selected halfword must match	Specifies the conditions under which a data value-comparison with the DVC2 register causes a debug event (DVC2 event). The comparison is made using the bytes selected by DV2BE.

Table 9-2: Debug-Control Register 1 (DBCR1) Field Definitions (Continued)

Bit	Name	Function	Description
16:19	DV1BE	Data-Value Compare 1 Byte Enables	Specifies which bytes in the DVC1 register are used in the comparison. Each DV1BE bit corresponds to a byte in the DVC1 register. DVC1 events are <i>disabled</i> when DV1BE=0b0000.
20:23	DV2BE	Data-Value Compare 2 Byte Enables	Specifies which bytes in the DVC2 register are used in the comparison. Each DV2BE bit corresponds to a byte in the DVC2 register. DVC2 events are <i>disabled</i> when DV2BE=0b0000.
24:31		Reserved	

The DBCR1 is a privileged SPR with an address of 957 (0x3BD) and is read and written using the **mfspir** and **mtspir** instructions.

Debug-Status Register

The PPC405 contains a 32-bit debug-status register (DBSR). Fields within the register are set by the various debug events to report debug status. The DBSR can be updated by a debug event even when all debug modes are disabled. DBSR[MRR] is updated by a reset, not by a debug event. **Figure 9-3** shows the format of the DBSR register. The fields in the DBSR are defined as shown in **Table 9-3**.

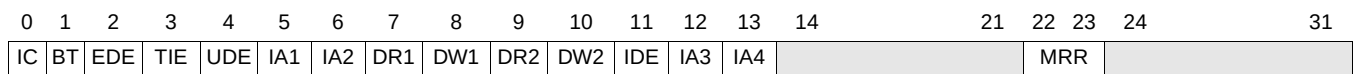


Figure 9-3: Debug-Status Register (DBSR)

Table 9-3: Debug-Status Register (DBSR) Field Definitions

Bit	Name	Function	Description
0	IC	Instruction-Complete Debug Event 0—Did not occur 1—Occurred	Indicates whether an instruction-complete debug event occurred.
1	BT	Branch-Taken Debug Event 0—Did not occur 1—Occurred	Indicates whether a branch-taken debug event occurred.
2	EDE	Exception-Taken Debug Event 0—Did not occur 1—Occurred	Indicates whether an exception-taken debug event occurred.
3	TDE	Trap-Instruction Debug Event 0—Did not occur 1—Occurred	Indicates whether a trap-instruction debug event occurred.
4	UDE	Unconditional Debug Event 0—Did not occur 1—Occurred	Indicates whether an unconditional debug event occurred.
5	IA1	Instruction-Address Compare 1 Debug Event 0—Did not occur 1—Occurred	Indicates whether an IAC1 debug event occurred.

Table 9-3: Debug-Status Register (DBSR) Field Definitions (Continued)

Bit	Name	Function	Description
6	IA2	Instruction-Address Compare 2 Debug Event 0—Did not occur 1—Occurred	Indicates whether an IAC2 debug event occurred.
7	DR1	Data-Address Compare 1 Read Debug Event 0—Did not occur 1—Occurred	Indicates whether a DAC1-read debug event occurred.
8	DW1	Data-Address Compare 1 Write Debug Event 0—Did not occur 1—Occurred	Indicates whether a DAC1-write debug event occurred.
9	DR2	Data-Address Compare 2 Read Debug Event 0—Did not occur 1—Occurred	Indicates whether a DAC2-read debug event occurred.
10	DW2	Data-Address Compare 2 Write Debug Event 0—Did not occur 1—Occurred	Indicates whether a DAC2-write debug event occurred.
11	IDE	Imprecise Debug Event 0—No debug event occurred 1—At least one debug event occurred	Indicates whether a debug event occurred when debug interrupts were disabled (MSR[DE]=0). This bit is not set if MSR[DE]=1.
12	IA3	Instruction-Address Compare 3 Debug Event 0—Did not occur 1—Occurred	Indicates whether an IAC3 debug event occurred.
13	IA4	Instruction-Address Compare 4 Debug Event 0—Did not occur 1—Occurred	Indicates whether an IAC4 debug event occurred.
14:21		Reserved	
22:23	MRR	Most-Recent Reset 00—No reset 01—Processor reset 10—Chip reset 11—System reset	Indicates the type of reset that last occurred.
24:31		Reserved	

The DBSR is a privileged SPR with an address of 1008 (0x3F0). Hardware sets the status bits and software is responsible for reading and clearing the bits. It is read using the **mfsprr** instruction. The register is cleared, but not directly written, using the **mtsprr** instruction. Values in the source register, **rS**, behave as a mask when clearing the DBSR. Here, a value of 0b1 in any bit position of **rS** *clears* the corresponding bit in the DBSR. A value of 0b0 in an **rS** bit position does not alter the corresponding bit in the DBSR.

Instruction Address-Compare Registers

The PPC405 contains four 32-bit instruction address-compare registers: IAC1, IAC2, IAC3, and IAC4. These registers are used by the instruction address-compare debug event. Figure 9-4 shows the format of the IAC_n registers. The instruction effective-addresses loaded in these registers must be word aligned (address bits 30:31 must be 0).

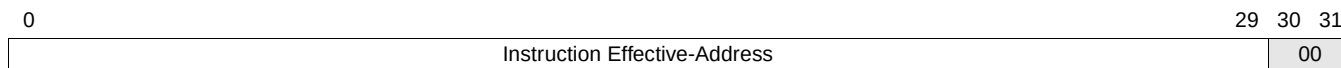


Figure 9-4: Instruction Address-Compare Registers (IAC1–IAC4)

The IAC_n registers are privileged SPRs with the following addresses:

- IAC1—1012 (0x3F4).
- IAC2—1013 (0x3F5).
- IAC3—948 (0x3B4).
- IAC4—949 (0x3B5).

These registers are read and written using the **mf spr** and **mt spr** instructions.

Data Address-Compare Registers

The PPC405 contains two 32-bit data address-compare registers, DAC1 and DAC2. These registers are used by the data address-compare debug event. Figure 9-5 shows the format of the DAC_n registers. Any byte-aligned data effective-address can be loaded in these registers.

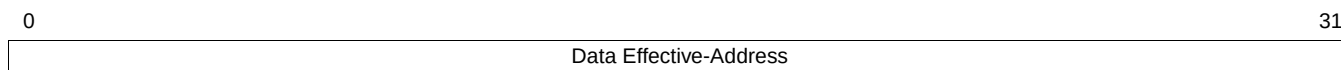


Figure 9-5: Data Address-Compare Registers (DAC1, DAC2)

The DAC_n registers are privileged SPRs with the following addresses:

- DAC1—1014 (0x3F6).
- DAC2—1015 (0x3F7).

These registers are read and written using the **mf spr** and **mt spr** instructions.

Data Value-Compare Registers

The PPC405 contains two 32-bit data value-compare registers, DVC1 and DVC2. These registers are used by the data value-compare debug event. Figure 9-6 shows the format of the DVC_n registers. Any data value can be loaded in these registers.

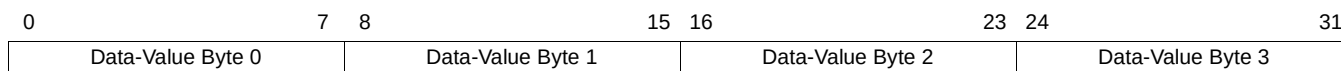


Figure 9-6: Data Value-Compare Registers (DVC1, DVC2)

The DVC_n registers are privileged SPRs with the following addresses:

- DVC1—950 (0x3B6).
- DVC2—951 (0x3B7).

These registers are read and written using the **mf spr** and **mt spr** instructions.

Debug Events

A debug event occurs when a debug condition is detected by the processor. Debug conditions are enabled using the debug-control registers (DBCR0 and DBCR1). Some of the debug events make use of one or more of the compare registers (IAC_n, DAC_n, and DVC_n). Depending on the debug mode, a debug event causes the following to occur:

- In internal-debug mode, a debug event is synonymous with debug exception. A debug event can cause a debug interrupt if debug interrupts are enabled (MSR[DE]=1). If debug interrupts are disabled, a debug event results in a *pending* debug interrupt. A debug interrupt occurs when a debug interrupt is pending and software sets MSR[DE] to 1.
- In external-debug mode, a debug event stops the processor. An external debugger connected to the processor through the JTAG port can restart the processor. A debug event can also cause a debug interrupt if both internal-debug mode and debug exceptions are enabled.
- If debug interrupts are enabled and both internal-debug and external-debug mode are enabled, a debug event stops the processor and the debug interrupt is pending.
- In debug-wait mode, a debug event stops the processor. A critical or noncritical external interrupt can restart the processor to handle the interrupt. The processor stops again when the interrupt handler is exited. An external debugger connected to the processor through the JTAG port can restart the processor.
- In real-time trace mode, a debug event can cause an external trigger event. Trigger events are used by external tools to collect instruction-trace information.

Debug status is recorded in the debug-status register (DBSR). A debug event can set debug-status bits even if all debug modes and debug exceptions are disabled. System software can use this capability to periodically poll the DBSR rather than use debug exceptions. Three events do not operate in this manner:

- Instruction-complete (IC).
- Branch-taken (BT).
- Instruction address-compare (IAC) when toggling is used.

The corresponding sections for these debug events describe the conditions under which debug status is not updated.

When debug interrupts are disabled (MSR[DE]=0), debug events are often recorded imprecisely. The occurrence of a debug event is reported by the debug status register, but the processor continues to operate normally and the debug interrupt is pending. When debug interrupts are later enabled, the pending interrupt causes a debug interrupt to immediately occur. See **Imprecise Debug Event**, [page 592](#) for more information.

Debug events are not caused by speculatively executed instructions. The processor only reports events for resolved instructions that reflect the normal operation of the sequential-execution model.

Table 9-4 summarizes the debug resources used by each debug event.

Table 9-4: Debug Resources Used by Debug Events

Debug Event	DBCR0	DBCR1	DBSR	IAC	DAC	DVC
IC Instruction Complete	IC		IC			
BT Branch Taken	BT		BT			
EDE Exception Taken	EDE		EDE			

Table 9-4: Debug Resources Used by Debug Events (Continued)

Debug Event	DBCRO	DBCRI	DBSR	IAC	DAC	DVC
TDE Trap Instruction	TDE		TDE			
UDE Unconditional			UDE			
IAC Instruction Address-Compare	IA1, IA2, IA3, IA4 IA12, IA12X, IA12T IA34, IA34X, IA34T		IA1, IA2, IA3, IA4	IAC1, IAC2, IAC3, IAC4		
DAC Data Address-Compare		D1R, D2R, D1W, D2W D1S, D2S DA12, DA12X	DR1, DR2 DW1, DW2		DAC1, DAC2	
DVC Data Value-Compare		D1R, D2R, D1W, D2W D1S, D2S DV1M, DV2M DV1BE, DV2BE	DR1, DR2 DW1, DW2		DAC1, DAC2	DVC1, DVC2
IDE Imprecise			IDE			

Instruction-Complete Debug Event

An instruction-complete (IC) debug event occurs immediately *after* completing execution of each instruction. It is enabled by setting DBCRO[IC]=1 and disabled by clearing DBCRO[IC]=0. The processor reports the occurrence of an IC debug event by setting the IC bit in the debug-status register (DBSR[IC]) to 1. After an IC event is recorded by a debugger, the status bit should be cleared to prevent ambiguity when recording future debug events.

The IC debug event *does not* set the DBSR status bit if all of the following are true:

- Internal-debug mode is enabled.
- Debug exceptions are disabled.
- External-debug mode is disabled.

Instruction completion is a common event (it can occur every processor clock) and this condition prevents the DBSR from recording its obvious occurrence when exceptions are disabled.

Many instructions do not complete execution when they cause an exception (other than the debug exception). Instructions that cause an exception do not result in an IC debug event. This **sc** instruction, however, causes a system-call exception *after* it executes. Here, the debug event occurs after the **sc** instruction, but before control is transferred to the system-call interrupt handler.

The IC debug event is useful for single-stepping through a program. Either the debug-interrupt handler (internal-debug mode) or an external debugger attached to the JTAG port (external-debug mode) can read and report the processor state and single-step to the next instruction.

If debug interrupts are enabled, the SRR2 register is loaded with the effective address of the instruction following the one that caused the IC event.

Branch-Taken Debug Event

A branch-taken (BT) debug event occurs immediately *before* executing a resolved (non-speculative) branch instruction. It is enabled by setting DBCR0[BT]=1 and disabled by clearing DBCR0[BT]=0. The processor reports the occurrence of a BT debug event by setting the BT bit in the debug-status register (DBSR[BT]) to 1. After a BT event is recorded by a debugger, the status bit should be cleared to prevent ambiguity when recording future debug events.

The BT debug event *does not* set a DBSR status bit if all of the following are true:

- Internal-debug mode is enabled.
- Debug exceptions are disabled.
- External-debug mode is disabled.

Branches are a common event and this condition prevents the DBSR from recording their obvious occurrence when exceptions are disabled.

This debug event is useful for single-stepping through branches to narrow the search for code sequences of interest. Once identified, debug software can enable IC debug events and single-step the code sequence instruction-by-instruction.

If debug interrupts are enabled, the SRR2 register is loaded with the effective address of the branch instruction that caused the BT event.

Exception-Taken Debug Event

An exception-taken (EDE) debug event occurs immediately *after* an exception occurs, but before the first instruction in the exception handler is executed. It is enabled by setting DBCR0[EDE]=1 and disabled by clearing DBCR0[EDE]=0. The processor reports the occurrence of an EDE debug event by setting the EDE bit in the debug-status register (DBSR[EDE]) to 1. After an EDE event is recorded by a debugger, the status bit should be cleared to prevent ambiguity when recording future debug events.

Noncritical exceptions always cause an EDE event when EDE is enabled. Critical exceptions cause an EDE event only when EDE is enabled *and* external-debug mode is enabled.

This debug event is useful for debugging interrupt handlers. Upon entering an interrupt handler, debug software can enable IC debug events and single-step the handler instruction-by-instruction.

If debug interrupts are enabled, the SRR2 register is loaded with the 32-bit exception-vector physical address. This corresponds to the effective address of the first instruction in the interrupt handler.

Trap-Instruction Debug Event

A trap-instruction (TDE) debug event occurs immediately *before* executing a trap instruction (**tw** or **twi**), if the conditions are such that a program exception would normally occur (invoking the system trap-handler). If the trap conditions are not met, the debug event does not occur and the program executes normally. The event is enabled by setting DBCR0[TDE]=1 and disabled by clearing DBCR0[TDE]=0. The processor reports the occurrence of a TDE debug event by setting the TDE bit in the debug-status register (DBSR[TDE]) to 1. After a TDE event is recorded by a debugger, the status bit should be cleared to prevent ambiguity when recording future debug events.

When TDE events are enabled, execution of a trap instruction *does not* cause a program exception if any of the following conditions are true:

- Internal-debug mode is enabled and debug exceptions are enabled.
- External-debug mode is enabled.
- Debug wait-mode is enabled.

A program exception does occur when TDE events are enabled and internal-debug mode is enabled, but debug interrupts are disabled. In this case, the processor records an imprecise-debug exception by setting DBSR[IDE]=1.

If debug interrupts are enabled, the SRR2 register is loaded with the effective address of the trap instruction that caused the TDE event.

Unconditional Debug Event

An unconditional (UDE) debug event occurs immediately if either of the following two conditions are true:

- An external debugger attached to the JTAG port causes the event.
- The external unconditional-debug-event signal is asserted.

There is no enable bit for this event. The processor reports a UDE event by setting the UDE bit in the debug-status register (DBSR[UDE]) to 1. After a UDE event is recorded by a debugger, the status bit should be cleared to prevent ambiguity when recording future debug events.

If debug interrupts are enabled, the SRR2 register is loaded with the effective address of the instruction that would have executed had the UDE event not occurred.

Instruction Address-Compare Debug Event

An instruction address-compare (IAC) debug event occurs immediately *before* executing an instruction. The effective address of the instruction must match the value contained in one of the four IAC_n registers. The IAC event is controlled by conditions specified in the DBCR0 register. Three IAC conditions can be specified:

- Check for an exact instruction-address match.
- Check for an instruction-address match within a range of addresses.
- Check for an instruction-address match outside a range of addresses.

If debug interrupts are enabled, the SRR2 register is loaded with the effective address of the instruction that caused the IAC event.

IAC Exact-Address Match

An IAC exact-address match causes a debug event when the effective address in the specified IAC_n register exactly matches the effective address of the executing instruction. IAC_n register comparisons are enabled by setting the appropriate IAC_n enable bits in the DBCR0 register to 1. If a match occurs, the corresponding status bit in DBSR is set to 1.

Table 9-6 shows the control bits used to enable the IAC exact-address-match debug events, the IAC_n register used in the comparison, and the debug-status register bit set when the event occurs. Any number of the IAC exact-address-match conditions can be enabled simultaneously. IAC address-range comparisons must be disabled as follows:

- DBCR0[IA12]=0 for IAC1 and IAC2 exact-match comparisons.
- DBCR0[IA34]=0 for IAC3 and IAC4 exact-match comparisons.

Table 9-5: IAC Exact-Address Match Resources

Event Enable Bit (DBCR0)	IAC Range Disable (DBCR0)	IAC Register Used	Event Status Bit (DBSR)
IA1	IA12=0	IAC1	IA1
IA2		IAC2	IA2
IA3	IA34=0	IAC3	IA3
IA4		IAC4	IA4

The processor does not clear the DBSR status bits when IAC events fail to occur. After an IAC event is recorded by a debugger, the corresponding status bits should be cleared to prevent ambiguity when recording future debug events.

IAC Address-Range Match

An IAC address-range match causes a debug event when the effective address of the executing instruction falls within a range of addresses specified an IAC_n register pair, as follows:

- IA12 designates an address range specified by the IAC1 and IAC2 register pair. To enable range comparisons using this register pair, software must:
 - Set DBCR0[IA12]=1.
 - Set either (or both) IA1=1 or IA2=1.
- IA34 designates an address range specified by the IAC3 and IAC4 register pair. To enable range comparisons using this register pair, software must:
 - Set DBCR0[IA34]=1.
 - Set either (or both) IA3=1 or IA4=1.

If IAC address-range comparison is enabled for a register pair, IAC exact-address comparison is disabled for that register pair.

When an address-range match is detected, the IA_n enable bits in DBCR0 determine which DBSR status bits are set to 1. For example, both DBSR[IA1, IA2] are set to 1 if DBCR0[IA1, IA2]=1 when an IA12 address-range match is detected. However, only DBSR[IA1] is set to 1 if DBCR0[IA1]=1 and DBCR0[IA2]=0 when an IA12 address-range match is detected. The processor does not clear the DBSR status bits when IAC events fail to occur. After an IAC event is recorded by a debugger, the corresponding status bits should be cleared to prevent ambiguity when recording future debug events.

Inclusive and Exclusive Ranges

The DBCR0[IA12X, IA34X] bits specify whether the corresponding address ranges are inclusive or exclusive, as follows:

- When clear, the corresponding range is inclusive.

If DBCR0[IA12X]=0, instruction addresses from (IAC1) to (IAC2)-1 fall within the range. Addresses from 0 to (IAC1)-1 and (IAC2) to 0xFFFF_FFFF fall outside the range.

If DBCR0[IA34X]=0, instruction addresses from (IAC3) to (IAC4)-1 fall within the range. Addresses from 0 to (IAC3)-1 and (IAC4) to 0xFFFF_FFFF fall outside the range.
- When set, the corresponding range is exclusive.

If DBCR0[IA12X]=1, instruction addresses from 0 to (IAC1)-1 and (IAC2) to 0xFFFF_FFFF fall within the range. Addresses from (IAC1) to (IAC2)-1 fall outside the range.

If DBCR0[IA34X]=1, instruction addresses from 0 to (IAC3)-1 and (IAC4) to 0xFFFF_FFFF fall within the range. Addresses from (IAC3) to (IAC4)-1 fall outside the range.

Figure 9-7 illustrates how ranges are specified using DBCR0[IA12X]. No shading indicates addresses that are in range and gray-shading indicates addresses that are out of range.

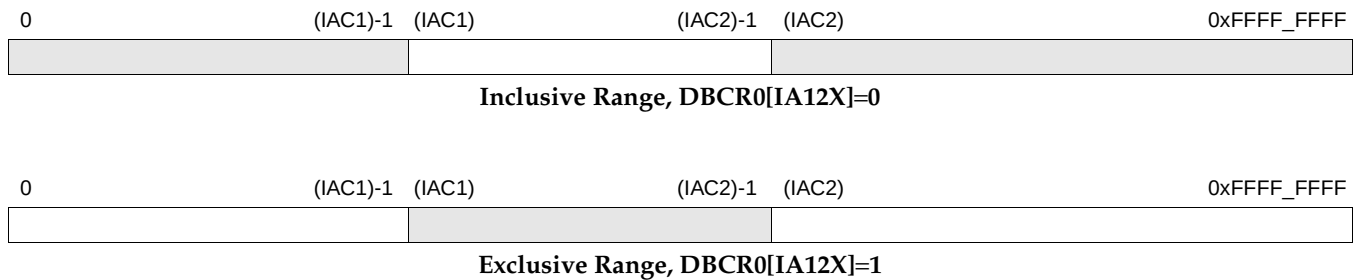


Figure 9-7: IAC Address-Range Specification

Range Toggling

Range comparisons can be set to toggle between inclusive and exclusive each time a debug event occurs on the specified range. DBCR0[IA12T]=1 enables toggling of the DBCR0[IA12X] bit and DBCR0[IA34T]=1 enables toggling of the DBCR0[IA34X] bit. Clearing a toggle bit disables toggling of the corresponding range bit.

As an example, assume IA12 exclusive-range toggling is enabled (IA12T=1 and IA12X=1):

- The first IAC event occurs when an instruction address is in the exclusive IA12 range. The processor clears IA12X to 0.
- The second IAC event occurs when an instruction address is in the inclusive IA12 range. The processor sets IA12X to 1.
- The third IAC event occurs when an instruction address is in the exclusive IA12 range. The processor clears IA12X to 0.
- And so on.

The IAC debug event does not set a DBSR status bit when toggling is used if all of the following are true:

- Internal-debug mode is enabled.
- Debug exceptions are disabled.
- External-debug mode is disabled.

When toggling is enabled IAC events occur frequently. This condition prevents the DBSR from recording their obvious occurrence when exceptions are disabled.

Data Address-Compare Debug Event

A data address-compare (DAC) debug event occurs *before* executing a data-access instruction. The effective address of the operand must match the value contained in one of the two DACn registers. Aligned memory accesses generate a single effective address that is used in checking for a DAC event. Unaligned memory accesses, load/store multiple instructions, and load/store

string instructions can generate multiple effective addresses, all of which are used to check for a DAC event. The DAC event is controlled by conditions specified in the DBCR1 register.

A variety of DAC conditions can be specified:

- Check for an exact data-address match.
- Check for a data-address match using halfword, word, or cacheline granularity.
- Check for a data-address match within a range of addresses.
- Check for a data-address match outside a range of addresses.

Each of the above DAC conditions can be further controlled to cause a debug event only if the matching data access is a read or a write.

If debug interrupts are enabled, the SRR2 register is loaded with the effective address of the instruction that caused the DAC event.

DAC Exact-Address Match

A DAC exact-address match causes a debug event when the effective address contained in the specified DAC n register matches the effective address of the operand. Read and write accesses can be checked independently. If a match occurs, the corresponding status bit in DBSR is set to 1.

Table 9-6 shows the control bits used to enable the DAC exact-address-match debug events, the type of access that is checked by each event, the DAC n register used in the comparison, and the debug-status register bit set when the event occurs. Any number of DAC exact-address-match conditions can be enabled simultaneously. DAC address-range comparison must be disabled (DBCR1[DA12]=0).

Table 9-6: DAC Exact-Address Match Resources

Event Enable Bit (DBCR1)	Type of Access Checked	DAC Register Used	Event Status Bit (DBSR)
D1R	Load (Read)	DAC1	DR1
D1W	Store (Write)		DW1
D2R	Load (Read)	DAC2	DR2
D2W	Store (Write)		DW2

The processor does not clear the DBSR status bits when DAC events fail to occur. After a DAC event is recorded by a debugger, the corresponding status bits should be cleared to prevent ambiguity when recording future debug events.

Specifying Exact-Match Granularity

Software can specify an operand-size granularity for use when performing the address comparison with each DAC register. Normally, the comparison checks for an exact address match or a byte-granular match. The comparison can be modified to check for halfword, word, and cache-line granular matches. This is useful when a debugger wants to cause a DAC event to occur when *any* byte in a word is accessed.

Granularity is specified using the DBCR1[D1S] size field for comparisons against the DAC1 register and the DBCR1[D2S] size field for comparisons against the DAC2 register. This field specifies which low-order address bits are *ignored* during the comparison. Because low-order address bits are ignored, the comparison is aligned on an address boundary equivalent to the

granularity. The following table shows the possible size-field values, the address bits that are ignored during the comparison, and the resulting granularity used in the comparison.

Table 9-7: Effect of D1S/D2S Size-Field Encoding

Size-Field Encoding	Address Bits Used	Address Bits Ignored	Granularity
00	0:31	—	Byte
01	0:30	31	Halfword
10	0:29	30:31	Word
11	0:26	27:31	Cacheline

Table 9-8 shows an example of using the D1S size field. The table shows how comparisons against the DAC address are modified using the size field. The first four entries apply byte-granular comparisons and only one of the four accesses produces a match. The second set of four entries apply a word-granular comparison. Here, all four of the accesses produce a match.

Table 9-8: Examples of Using the D1S Size Field

DAC Address	D1S Value (Granularity)	Operand Address	Access Size	DAC Match
0x0002	00 (Byte)	0x0000	Byte	No
		0x0000	Word	No
		0x0002	Word	Yes
		0x0003	Byte	No
0x0002	10 (Word)	0x0000	Byte	Yes
		0x0000	Word	Yes
		0x0002	Word	Yes
		0x0003	Byte	Yes

The load-string and store-string instructions move bytes of data between memory and registers. However, when these instructions are used to access data PPC405 moves four bytes at a time by using word-aligned effective addresses and an access size of one word. Bytes not required by the instructions are discarded. Thus, it is not possible to produce a byte-granular DAC match on every byte address referenced by a string instruction. In some cases, software must use a word-size granularity to produce a DAC match on a specific byte address.

DAC Address-Range Match

A DAC address-range match causes a debug event when the effective address of the operand falls within a range specified by the DA12 register pair. DAC1 and DAC2 form the DA12 pair. DA12 range comparison is enabled by setting DBCR1[DA12]=1. When DAC address-range comparison is enabled, DAC exact-address comparison is disabled. The DBCR1[D1S, D2S] size bits are not used by DAC address-range comparisons.

Read and write accesses can be checked independently. To check read accesses, software sets the D1R and/or D2R bits in the DBCR1 register. Only one of the two bits must be set to enable read checking for the entire range. If

a read-access match is detected, the corresponding status bits in the DBSR are set (DR1 and/or DR2). Likewise, write-access for the entire range is checked by setting the D1W and/or D2W bits in the DBCR1 register. If a write-access match is detected, the corresponding status bits in the DBSR are set (DW1 and/or DW2).

Inclusive and Exclusive Ranges

The DBCR1[DA12X] bit determines whether the address range specified by the DAC_n registers is inclusive or exclusive:

- When DBCR1[DA12X]=0, the range is *inclusive*. Addresses from (DAC1) to (DAC2)-1 fall within the range. Addresses from 0 to (DAC1)-1 and (DAC2) to 0xFFFF_FFFF fall outside the range.
- When DBCR1[DA12X]=1, the range is *exclusive*. Addresses from 0 to (DAC1)-1 and (DAC2) to 0xFFFF_FFFF fall within the range. Addresses from (DAC1) to (DAC2)-1 fall outside the range.

Figure 9-8 shows the range specification based on the value of DBCR1[DA12X]. No shading indicates addresses that are in range and gray-shading indicates addresses that are out of range.

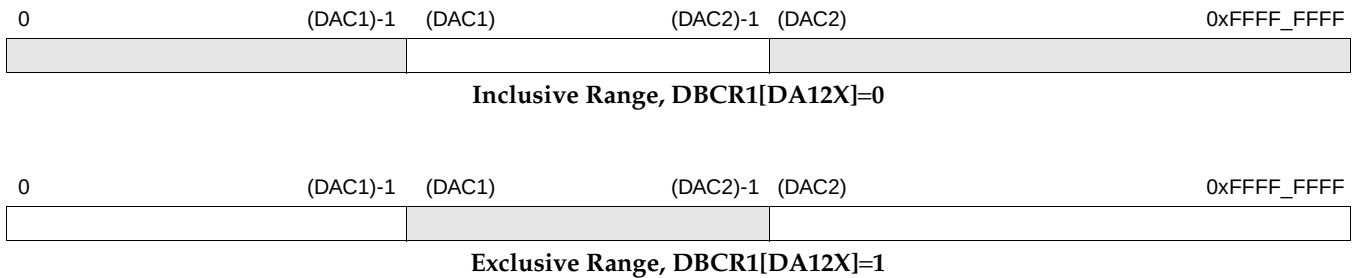


Figure 9-8: DAC Address-Range Specification

Table 9-9 summarizes the DBCR1 bits used to control DAC address-range comparisons and the DBSR bits used to report their status.

Table 9-9: DAC Address-Range Match Resources

Event Enable Bit (DBCR1)	DBCR1 [DA12X]	Type of Access Checked	Event Status Bit (DBSR)
D1R and/or D2R	0	Load (read) inclusive (DAC1) and (DAC2)-1	DR1 and/or DR2
D1W and/or D2W		Store (write) inclusive (DAC1) and (DAC2)-1	DW1 and/or DW2
D1R and/or D2R	1	Load (read) exclusive (DAC1) and (DAC2)-1	DR1 and/or DR2
D1W and/or D2W		Store (write) exclusive (DAC1) and (DAC2)-1	DW1 and/or DW2

The processor does not clear the DBSR status bits when DAC events fail to occur. After a DAC event is recorded by a debugger, the corresponding status bits should be cleared to prevent ambiguity when recording future debug events.

DAC Events Caused by Cache Instructions

DAC events can be caused by the execution of cache-control instructions. The following summarizes the type of DAC events that can occur when a cache-control instruction is executed:

- Cache-control instructions that can modify data are treated as stores

(writes) by the debug mechanism. Instructions that can cause loss of data through invalidation are also treated as stores. Both types of instructions can cause DAC-write events. Instructions in this category are **dcbi** and **dcbz**.

- Cache-control instructions that invalidate unmodified are treated as loads. These instructions can cause DAC-read events but not DAC-write events. The **icbi** instruction falls in this category.
- Cache-control instructions that are not address specific do not cause DAC events. Instructions in this category are **dccci**, **iccci**, **dcread**, and **icread**.
- Cache-control instructions that update system memory with data already present in the cache are treated as loads (reads) by the access-protection mechanism. However, the debug mechanism can be used to cause a DAC-write event when these instructions are executed. Instructions in this category are **dcbf** and **dcbst**.
- Cache-control instructions that are speculative are treated as loads by the debug mechanism. These instructions can cause DAC-read events. Instructions in this category are **dcbt**, **dcbtst**, and **icbt**.
- Cache-control instructions that allocate cachelines are treated as stores. These instructions can cause DAC-write events. The **dcba** instruction falls in this category.

Table 9-10 summarizes the type of DAC event that can occur for each cache-control instruction.

Table 9-10: DAC Events Caused by Cache-Control Instructions

Instruction	DAC Read	DAC Write
dcba	No	Yes
dcbf	No	Yes
dcbi	No	Yes
dcbst	No	Yes
dcbt	Yes	No
dcbtst	Yes	No
dcbz	No	Yes
dccci	Does not cause DAC events.	
dcread	Does not cause DAC events.	
icbi	Yes	No
icbt	Yes	No
iccci	Does not cause DAC events.	
icread	Does not cause DAC events.	

Data Value-Compare Debug Event

A data value-compare (DVC) debug event occurs when:

1. A DAC match occurs. The operand effective-address of the data-access instruction must match the value contained in one of the DAC_n registers, using the conditions specified by the DBCR1 register.

2. If the preceding DAC comparison detects a matching address, the data-value accessed at that address must match the value contained in one of the DVC_n registers, using the conditions specified by the DBCR1 register.

The DAC comparison performed in the first step is set up to perform exact-address or address-range comparisons as described in the previous section (**Data Address-Compare Debug Event**). However, the DAC comparison does *not* cause a DAC debug event. Because DVC and DAC events share the same DAC registers, control bits, and status bits, a DAC event is disabled when the corresponding DVC event is enabled, as follows:

- If DVC1 events are enabled, DAC1 events are disabled.
- If DVC2 events are enabled, DAC2 events are disabled.
- If DVC1 and DVC2 events are enabled (as in range comparisons), DAC1 and DAC2 events are disabled.

Unlike DAC events, the DVC event occurs *after* the data-access instruction executes. If debug interrupts are enabled, the SRR2 register is loaded with the effective address of the instruction following the one that caused the DVC event.

DVC events are enabled by loading a non-zero value ($\neq 0b0000$) into the byte-enable controls of the corresponding DVC_n register. A non-zero value loaded into DBCR1[DV1BE] enables DVC1 events and a non-zero value loaded into DBCR1[DV2BE] enables DVC2 events. Referring to **Figure 9-6, page 579**, the byte-enables specify which DVC_n register bytes participate in the DVC comparison:

- DV_nBE₀ controls participation of DVC_n data-value byte 0.
- DV_nBE₁ controls participation of DVC_n data-value byte 1.
- DV_nBE₂ controls participation of DVC_n data-value byte 2.
- DV_nBE₃ controls participation of DVC_n data-value byte 3.

When a DV_nBE bit is set to 1, the specified byte in DVC_n is compared against the corresponding operand byte. If the bit is cleared to 0, the specified byte is not compared. If DV_nBE=0b0000, no bytes participate in the comparison and the DVC_n event is disabled.

The data-value compare-mode bits in DBCR1 control how the enabled DVC_n bytes are compared against the operand value. The DV1M bits control the DVC1 comparison and the DV2M bits control the DVC2 comparison. The modes defined by these two-bit fields are:

- 00—The effect of this mode is undefined and should not be used.
- 01—AND mode. All DVC_n bytes selected by DV_nBE must match the corresponding operand bytes.
- 10—OR mode. At least one of the DVC_n bytes selected by DV_nBE must match the corresponding operand byte.
- 11—AND-OR mode. This mode uses the following algorithm to determine whether a DVC event occurs:

```
( DVnBE0 ∧ (DVn[byte_0] = data_value[byte_0]) ) ∧
  DVnBE1 ∧ (DVn[byte_1] = data_value[byte_1]) ) ∨
( DVnBE2 ∧ (DVn[byte_2] = data_value[byte_2]) ) ∧
  DVnBE3 ∧ (DVn[byte_3] = data_value[byte_3]) )
```

This comparison mode is useful when the byte enables are set to 0b1111. Here, a DVC event occurs if either the upper halfword or lower halfword of the DVC_n register matches the corresponding operand halfword.

Table 9-11 shows example settings of DV1BE and DV1M and how they affect detection of a DVC1 match.

Table 9-11: Examples of Using DVC1 Controls

Data Value	DVC1 Value	DV1BE	DV1M	DVC1 Match
0xABCD_FFFF	0xABCD_0123	0b0111	01 (AND)	No
			10 (OR)	Yes
			11 (AND-OR)	No
		0b1000	01 (AND)	Yes
			10 (OR)	Yes
			11 (AND-OR)	No
		0b1100	01 (AND)	Yes
			10 (OR)	Yes
			11 (AND-OR)	Yes
		0b1111	01 (AND)	No
			10 (OR)	Yes
			11 (AND-OR)	Yes

Occasionally, it is desirable to cause a DVC event during an access to unaligned data. Software can use both DVC1 and DVC2 to (and the corresponding DAC_n registers) to detect accesses to either portion of the misaligned data. However, misaligned accesses can result in the generation of two effective addresses that are accessed separately by the processor. If the first address causes a DVC event, that event is recorded before completing access to the second address. If an interrupt occurs as a result of the DVC event, the second access is lost. This can result in a corrupted register and/or memory value.

DVC read and write events are enabled by initializing the DAC comparison and the D_nR and D_nW control bits in DBCR1. When a DVC event occurs, DBSR status bits are set to reflect the event. Read and write DVC events are recorded independently using the DR_n and DW_n status bits. [Table 9-12](#) summarizes how the status bits are used by DVC events.

Table 9-12: DVC Event Status

DAC Enable Bit (DBCR1)	Type of Access Checked	Registers Used		DVC Status Bit (DBSR)
D1R	Load (Read)	DAC1	DVC1	DR1
D1W	Store (Write)			DW1
D2R	Load (Read)	DAC2	DVC2	DR2
D2W	Store (Write)			DW2

Status bits can be set by either DAC events or DVC events. However, a DAC event can occur only when DVC events are disabled. DAC matches do not set the status bits if DVC events are enabled but fail to occur. After a DAC or DVC event is recorded by a debugger, the corresponding status bits should be cleared to prevent ambiguity when recording future debug events.

Imprecise Debug Event

Imprecise (IDE) debug events are the result of any debug event occurring when debug interrupts are disabled ($MSR[DE]=0$). Internal-debug mode can be enabled or disabled. When this happens, the imprecise-debug-exception bit in the debug-status register ($DBSR[IDE]$) is set to 1. This bit is set in *addition to* all other debug-status bits associated with the actual event.

If $DBSR[IDE]=1$ and debug interrupts are enabled, a debug interrupt immediately occurs. The $SRR2$ register is loaded with the effective address of the instruction following the one that enabled debug interrupts. For example, assume internal-debug mode and debug interrupts are both disabled. If $MSR[DE]$ is enabled first, followed by an enable of $DBCR0[IDM]$, $SRR2$ is loaded with the instruction address following the one that enabled $DBCR0[IDM]$.

To prevent repeated interrupts from occurring, the interrupt handler must clear $DBSR[IDE]$ before returning. After the event is recorded by a debugger, debug-status bits should be cleared to prevent ambiguity when recording future debug events.

The following debug events can result in an imprecise debug event when $MSR[DE]=0$:

- Instruction complete (IC), if $DBCR0[IDM]=0$. If internal-debug mode is enabled, IC events cannot cause imprecise debug events when $MSR[DE]=0$.
- Branch taken (BT), if $DBCR0[IDM]=0$. If internal-debug mode is enabled, BT events cannot cause imprecise debug events when $MSR[DE]=0$.
- Exception taken (EDE).
- Trap instruction (TDE).
- Unconditional (UDE).
- Instruction address-compare (IAC). However, if IAC range toggling is enabled and internal-debug mode is enabled, IAC events cannot cause imprecise debug events when $MSR[DE]=0$.
- Data address-compare (DAC).
- Data value-compare (DVC).

This feature is useful for indicating that one or more debug events occurred during execution of a critical-interrupt handler (debug interrupts are disabled by critical interrupts). Upon returning from the interrupt handler, debug interrupts are re-enabled and the processor immediately transfers control to the debug-interrupt handler.

Freezing the Timers

The PPC405 timers can be frozen (stopped) when a debug event occurs. This is done by setting the freeze timers bit (FT) in $DBCR0$ to 1. If $DBCR0[FT]=1$ when any debug event occurs, the time base stops incrementing and the programmable-interval timer stops decrementing. Freezing the timers also prevents the occurrence of the PIT, FIT, and WDT timer events. The timers are not frozen when a debug event occurs and $DBCR0[FT]=0$.

After the timers are frozen, they are not unfrozen until the record of all debug events is cleared from the debug-status register. All bits in the $DBSR$ *except for* the most-recent reset (MRR) must be cleared to 0 to restart the timers. The timers are unfrozen when the processor recognizes the cleared state of the $DBSR$.

Debug Interface

The PPC405 provides a JTAG interface and trace interface to support testing and debugging of both hardware and software. Typically, the JTAG interface is exposed at the board level as a JTAG debug port, where an external debugger can connect to it using a JTAG connector. The trace interface is also exposed at the board level using a separate interface.

JTAG Debug Port

The PPC405 JTAG (Joint Test Action Group) debug port complies with IEEE standard 1149.1–1990, *IEEE Standard Test Access Port and Boundary Scan Architecture*. This standard describes a method for accessing internal chip resources using a four-signal or five-signal interface. The PPC405 JTAG debug port supports scan-based board testing and is further enhanced to support the attachment of debug tools. These enhancements comply with the IEEE 1149.1 specifications for vendor-specific extensions and are compatible with standard JTAG hardware for boundary-scan system testing.

The PPC405 JTAG debug port supports the following;

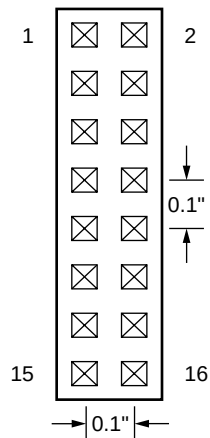
- *JTAG Signals*—The JTAG debug port implements the four required JTAG signals: TCK, TMS, TDI, and TDO. It also implements the optional TRST signal.
- *JTAG Clock*—The frequency of the JTAG clock signal (TCK) can range from 0 MHz (DC) to one-half of the processor clock frequency.
- *JTAG Reset*—The JTAG-debug port logic is reset at the same time the system is reset, using the JTAG reset signal (TRST). When TRST is asserted, the JTAG TAP controller returns to the test-logic reset state.

The JTAG debug port supports the required *extest*, *idcode*, *sample/preload*, and *bypass* instructions. The optional *highz* and *clamp* instructions are also supported. Invalid instructions behave as the *bypass* instruction.

Refer to the **PPC405 Processor Block Manual** for more information on the JTAG debug-port signals. Information on JTAG is found in the IEEE standard 1149.1–1990.

JTAG Connector

A male, 16-pin 2x8-header connector is suggested for use as the JTAG debug port connector. This connector supports direct attachment to the IBM RISCWatch debugger. The layout of the connector is shown in **Figure 9-9** and the signals are described in **Table 9-13**. At the board level, the connector should be placed as close as possible to the processor chip to ensure signal integrity. Position 14 is used as a connection key and does not contain a pin.



UG011_49_033101

Figure 9-9: JTAG-Connector Physical Layout

Table 9-13: JTAG Connector Signals

Pin	I/O	Signal Name	Description
1	O	TDO	JTAG test-data out.
2	NC	Reserved (no connection)	
3	I	TDI ¹	JTAG test-data in.
4	I	TRST	
5	NC	Reserved (no connection)	
6	I	+Power ²	Processor power OK
7	I	TCK ³	JTAG test clock.
8	NC	Reserved (no connection)	
9	I	TMS	JTAG test-mode select.
10	NC	Reserved (no connection)	
11	I	HALT	Processor halt.
12	NC	Reserved (no connection)	
13	NC	Reserved (no connection)	
14	KEY	No pin should be placed at this position.	
15	NC	Reserved (no connection)	
16		GND	Ground

Notes:

1. A 10K Ω pull-up resistor should be connected to this signal to reduce chip-power consumption. The pull-up resistor is not required.
2. The +POWER signal, is provided by the board, and indicates whether the processor is operating. This signal does not supply *power* to the debug tools or to the processor. A series resistor (1K Ω or less) should be used to provide short-circuit current-limiting protection.
3. A 10K Ω pull-up resistor must be connected to these signals to ensure proper chip operation when these inputs are not used.

BSDL

The *boundary-scan description language* (BSDL) provides a description of component testability features. It is used by automated test-pattern generation tools for package-interconnect tests and by electronic design-automation (EDA) tools for verification and for synthesizing test logic. BSDL supports extensions that can be used for internal-test generation and to write software for hardware debugging and diagnostics.

The primary components of BSDL include:

- The *logical-port description*, which assigns symbolic names to each pin at the chip level. Pins are also assigned a logical-type description of *in*, *out*, *inout*, *buffer*, or *linkage*. This description defines the direction of information flow through the pin.
- The *physical-pin map*, which provides correlation between the chip-level logical ports and the physical pin locations on a specific package. A BSDL description can contain several physical pin maps that describe different packages. Every pin map within the BSDL description is given a unique name.
- The *instruction statements*, which describe bit patterns that must be shifted into the instruction register to place the chip into the various test modes defined by the BSDL standard. Instruction-statements also support instruction descriptions unique to the chip.
- The *boundary-register description*, which lists each shift cell (also known as a shift stage) in the boundary register. Each cell is numbered. Cell 0 is defined as the cell closest to the test-data out (TDO) pin. The cell with the highest number is defined as the cell closest to the test-data in (TDI) pin. Cells contain additional information, including the cell type, the logical port associated with the cell, the logical function of the cell, the "safe" value for the cell, the "disable" value for the cell, the reset value for the cell, and a control number.

For more information, refer to IEEE standard 1149.1b-1994, which defines BSDL. This standard is a supplement to IEEE standards 1149.1-1990 (standard test-access port) and 1149.1a-1993 (boundary-scan architecture). BSDL is a subset of the *VHSIC hardware description language* (VHDL), a standard defined by IEEE 1076-1993.

Reset and Initialization

This chapter describes the reset operations recognized by the PPC405, the initial state of the PPC405 after a reset, and an example of the initialization code required to configure the processor. Initialization of external devices (on-chip or off-chip) is outside the scope of this document.

Reset

A *reset* causes the processor to perform a hardware initialization. It always occurs when the processor is powered-up and can occur at any time during normal operation. If it occurs during normal operation, instruction execution is immediately halted and all processor state is lost.

The PPC405 recognizes three types of reset:

- A *processor reset* affects the processor only, including the execution units and cache units. External devices (on-chip and off-chip) are not affected. This type of reset is sometimes referred to as a core reset.
- A *chip reset* affects the processor and all other devices or peripherals located on the same chip as the processor.
- A *system reset* affects the processor chip and all other devices or peripherals external to the processor chip that are connected to the same system-reset network. The scope of a system reset depends on the system implementation.

The type of reset is recorded in the most-recent reset field of the debug-status register (DBSR[MRR]). System software can examine this field if it needs to determine the cause of a reset. The effect of a reset on the processor is always the same regardless of the type.

Reset is caused by any of the following conditions:

- The processor is powered-up. Normally, the system performs a power-up sequence that includes asserting the external reset signals during a system reset.
- During normal operation, a system reset can be asserted using external reset signals. The processor logs this as a system reset, never as a processor reset or a chip reset.
- The second time-out of the watchdog timer can be programmed to cause a reset.
- Software can cause a reset by writing a non-zero value into the reset field of debug-control register 0 (DBCR0[RST]).
- An external debug tool can force a reset through the JTAG debug port.

Throughout this document, the term “reset” is applied collectively to all forms of reset. A type of reset is specified explicitly only when it is germane to the discussion.

Processor State After Reset

System software is responsible for fully initializing and configuring most processor resources. After a reset, the contents of most PPC405 registers are undefined and software

should not rely on any initial values contained in those registers. The machine-state register and several special-purpose registers have defined contents following a reset. This enables the processor to quickly initialize the minimum number of registers for proper instruction fetching and execution.

At the chip level, device control registers can be initialized to defined values following a reset. However, the registers and their initial contents are implementation-dependent.

Machine-State Register

Following a reset, the machine-state register (MSR) is cleared to 0x0000_0000. **Table 10-1** lists the implication of reset on the processor state as controlled by the MSR.

Table 10-1: MSR State Following Reset

MSR Bit	Value	Implication
AP	0	Auxiliary-processor unit unavailable.
APE	0	Auxiliary-processor unit exceptions disabled.
WE	0	Wait state disabled.
CE	0	Critical interrupts (external) disabled.
EE	0	Noncritical interrupts (external) disabled.
PR	0	Processor is in privileged mode.
FP	0	Floating-point unit unavailable.
ME	0	Machine-check exceptions disabled.
FE0	0	Floating-point exceptions disabled.
DWE	0	Debug-wait mode disabled.
DE	0	Debug exceptions disabled.
FE1	0	Floating-point exceptions disabled.
IR	0	Processor is in real mode (instruction translation is disabled).
DR	0	Processor is in real mode (data translation is disabled).

Special-Purpose Registers

Table 10-2 shows the contents of the special-purpose registers (SPRs) that have defined values following a reset. The contents of all other SPRs are undefined after a reset.

Table 10-2: SPR Contents Following Reset

Register	Value	Comment
DBCR0	0x0000_0000	Debug modes, events, and instruction comparisons are disabled.
DBCR1	0x0000_0000	Data comparisons are disabled.
DBSR	Undefined ¹	Most-recent reset (MRR) is set as specified in the note.
DCCR	0x0000_0000	Data-cache is disabled.
ESR	0x0000_0000	No exception syndromes are recorded.
ICCR	0x0000_0000	Instruction-cache is disabled.
PVR	0x2001_0820	Identifies the processor.
SGR	0xFFFF_FFFF	All memory is guarded.
SLER	0x0000_0000	All memory is big endian.
SU0R	0x0000_0000	All user-defined memory attributes are disabled.
TCR	Undefined ²	Watchdog-reset control (WRC) is cleared.
TSR	Undefined ¹	Most-recent watchdog reset (WRS) is set as specified in the note.

Notes:

- The most-recent reset bits are set as follows:
 - 00—No reset occurred. This is the value of WRS if the watchdog timer *did not* cause the reset.
 - 01—A processor-only reset occurred.
 - 10—A chip reset occurred.
 - 11—A system reset occurred.
- WRC is cleared, disabling watchdog time-out resets.

First Instruction

After the processor completes the hardware-initialization sequence caused by a reset, it performs an instruction fetch from the address 0xFFFF_FFFC. This first instruction is typically an unconditional branch to the initialization code. If the instruction at this address is not a branch, instruction fetching wraps to address 0x0000_0000. The system must be designed to provide non-volatile memory that contains the first instruction and the initialization code.

Because the processor is initially in big endian mode, initialization code must be in big endian format. It must remain in big endian format until memory and the processor are configured for little-endian operation.

Initialization

During reset, the minimum number of resources required for software execution are initialized by the processor. Initialization software is generally required to fully configure both the processor and system for normal operation. The following provides a checklist of tasks the initialization code should follow when performing this configuration.

- Configure the real-mode memory system by updating the storage-attribute control registers.
 - After reset, all memory is marked as guarded storage, preventing speculative instruction fetches. To improve fetch performance, the

- SGR register should be updated to mark memory as guarded only where necessary. All remaining memory should not be guarded.
 - Initially, memory is big endian. If little-endian memory is accessed, the SLER register must be updated appropriately.
 - User-defined storage attributes are disabled. If used by system software, they must be enabled in the SU0R register.
2. Configure the CCR0 register to specify how data and instructions are loaded from system memory. Because this register is uninitialized, it is important for software to update this register to maximize performance. If possible:
 - Loads, stores, and instruction fetches should allocate cachelines on a miss.
 - Prefetching should be enabled from cacheable and non-cacheable memory.
 - The request sizes for non-cacheable instruction fetches and data accesses should be set to the cache-line size (8 words).
 3. Configure the instruction cache to further improve instruction-fetch performance.
 - The instruction cache must first be invalidated. The contents of the cache are undefined following a reset and it is possible that some cachelines are improperly marked valid. Cache invalidation guarantees that false hits do not occur.
 - After reset, all memory is initialized as non-cacheable (the ICCR register is cleared). Software should update this register as appropriate to enable instruction caching.
 4. Configure the data cache to improve data-access performance.
 - Like the instruction cache, the data cache must first be invalidated. The contents of the cache are undefined following a reset and it is possible that some cachelines are improperly marked valid. Cache invalidation guarantees that false hits do not occur.
 - The DCWR register must be initialized to specify which memory locations use a write-back caching policy and which locations use a write-through policy. This specification is required only for those locations marked cacheable in the next step.
 - After reset, all memory is initialized as non-cacheable (the DCCR register is cleared). Software should update this register as appropriate to enable data caching.
 5. Configure the interrupt-handling mechanism. Internal exceptions are always enabled. Up to this point it is important that initialization code not cause an exception.
 - Interrupt handlers must be loaded into the appropriate system memory locations.
 - The interrupt-handler table must be loaded with the "glue code" that properly transfers control to the interrupt handlers following an exception.
 - The EVPR register must be loaded with the base address of the interrupt-handler table.
 - The timer resources must be initialized. If timers are not used, the TCR register must be initialized to prevent the occurrence of timer exceptions. Timer exceptions are enabled when critical and noncritical external exceptions are enabled.

- Enable critical and noncritical external exceptions by setting their enable bits in the MSR register.
- 6. If necessary, additional processor features can be initialized, including the memory-management resources.
- 7. System-level initialization is typically required. This often involves configuration of external devices and the loading of device drivers into system memory.

Following the initialization sequence outlined above, the operating system and application software can be loaded and executed.

Sample Initialization Code

Following is sample initialization code that illustrates the steps outlined above. The sample code is presented as pseudocode. Where appropriate, function calls are given names similar to the PowerPC instruction mnemonics. Specific chip-level implementations containing the PPC405 might require a different initialization sequence to ensure the processor is properly configured.

```

/* ----- */
/* PPC405 INITIALIZATION PSEUDOCODE */
/* ----- */

@0xFFFFF0C:      /* Initial instruction fetch from
0xFFFF_F0C. */
ba(init_code);    /* Branch to initialization code. */

@init_code:

/* ----- */
/* Configure guarded attribute for performance. */
/* ----- */
mtspr(SGR, guarded_attribute);

/* ----- */
/* Configure endian and user-defined attributes. */
/* ----- */
mtspr(SLER, endian);
mtspr(SUOR, user_defined);

/* ----- */
/* Configure CCR0. */
/* ----- */
mtspr(CCR0, prefetch_enables);
mtspr(CCR0, allocate_on_fetch_miss);
mtspr(CCR0, allocate_on_load_miss);
mtspr(CCR0, allocate_on_store_miss);
mtspr(CCR0, non_cachable_line_fill);

/* ----- */
/*
/* Invalidate the instruction cache and enable cachability.
/*
/* ----- */
/*
iccci;                /* Flash invalidate the
cache. */
mtspr(ICCR, i_cache_cachability); /* Enable the instruction
cache */

```

```

isync;                                     /* Synchronize the context.
*/

/* ----- */
/* Invalidate the data cache and enable cachability. */
/* ----- */
address = 0; /* Start with the first congruence class. */

/* Iterate through the data-cache congruence classes. */
for (line = 0; line <256; line++)
{
    dccci(address); /* Invalidate the congruence class. */
    address += 32; /* Point to the next congruence class. */
}

mtspr(DCWR, write-back, write-through); /* Set the caching
policy. */
mtspr(DCCR, d_cache_cachability);      /* Enable the data
cache. */
isync;                                     /* Synchronize the
context. */

/* ----- */
/* Prepare the system for interrupts. */
/* ----- */

/* Load interrupt handlers. */
/* Initialize interrupt-vector table. */

/* Initialize exception-vector prefix */
mtspr(EVPR, prefix_addr);

/* ----- */
/* Prepare system for asynchronous interrupts. */
/* ----- */

/* Initialize and configure timer resources. */
mtspr(PIT, 0); /* Disable PIT. */
mtspr(TSR, 0xFFFFFFFF); /* Clear TSR */
mtspr(TCR, timer_enable); /* Enable desired timers */
mtspr(TBL, 0); /* First clear TBL to avoid
rollover. */
mtspr(TBU, time_base_u); /* Set TBU to desired value. */
mtspr(TBL, time_base_l); /* Set TBL to desired value. */
mtspr(PIT, pit_count); /* Initialize PIT. */

/* Enable exceptions immediately to avoid missing timer
events. */
mtmsr(enable_exceptions);

/* ----- */
/* The MSR also controls: */
/* 1. Privileged and user mode */
/* 2. Address translation */
/* These can be initialized by the operating system. */
/* ----- */

/* If enabling translation, the TLB must be initialized. */

/* Set the machine state as desired. */
mtmsr(machine_state);

```

```
/* ----- */
/* Initialize other processor resources. */
/* ----- */

/* ----- */
/* Initialize non-processor resources. */
/* ----- */

/* ----- */
/* Branch to operating system or application code. */
/* ----- */
```


Instruction Set

This chapter lists the PPC405 instructions in alphabetical order by mnemonic. **Figure 11-1** shows an example format for an instruction description.

Mnemonic	→	add														
Name	→	Add														
Syntax	→	add rD, rA, rB (OE=0, Rc=0) add. rD, rA, rB (OE=0, Rc=1) addo rD, rA, rB (OE=1, Rc=0) addo. rD, rA, rB (OE=1, Rc=1)														
Form	→	XO Instruction Form														
Encoding	→	<table><tr><td>31</td><td>rD</td><td>rA</td><td>rB</td><td>OE</td><td>266</td><td>Rc</td></tr><tr><td>0</td><td>6</td><td>11</td><td>16</td><td>21 22</td><td></td><td>31</td></tr></table>	31	rD	rA	rB	OE	266	Rc	0	6	11	16	21 22		31
31	rD	rA	rB	OE	266	Rc										
0	6	11	16	21 22		31										
Description	→	Description The sum of the contents of register rA and register rB is loaded into register rD.														
Pseudocode	→	Pseudocode $(rD) \leftarrow (rA) + (rB)$														
Registers Altered	→	Registers Altered - rD. - CR[CR0]_{LT,GT,EQ,SO} if Rc=1. - XER[SO,OV] if OE=1. If an overflow occurs, it is possible the contents of CR0 do not reflect the infinitely-precise result.														
Exceptions	→	Exceptions None														
Compatibility	→	Compatibility This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.														

UG011_50_033101

Figure 11-1: Instruction Description Format

Each instruction description contains the following information shown in **Figure 11-1**:

Mnemonic—A short, single-word name for the base instruction. Throughout this document, instruction mnemonics are shown in lowercase bold (e.g. **add**).

Name—The descriptive name for the instruction. For example, the descriptive name for the **srawi** instruction is *Shift Right Algebraic Word Immediate*.

Syntax—The assembler syntax used for the instruction. Some instructions have up to four possible syntax variations. These variations depend on whether the instruction form contains an overflow-enable bit (OE) and/or a record bit (Rc). For these instructions, the use of the OE and Rc bits is reflected in the instruction mnemonic.

Form—The format used to encode the instruction. All PowerPC instructions are encoded using one of the following forms: A, B, D, I, M, SC, X, XL, XO, XFX, or XFL. See **Instructions Grouped by Form**, [page 828](#) for a description of each form and a list of instructions sorted by form.

Encoding—The specific encoding used to specify the instruction and its operands. See **Instruction Encoding**, below for more information.

Description—A description of how each instruction operates on the specified operands. The effect of the instruction on the CR and XER registers is also described. For some instructions, additional information is provided as to the purpose and use of the instruction. Many descriptions have cross-references to more detail in other sections of the manual. If simplified mnemonics are defined for an instruction, a cross-reference into [Appendix C, Simplified Mnemonics](#) is provided.

Pseudocode—A description of the instruction operation using a semi-formal language. The pseudocode conventions are used throughout this document and are described in the Preface in **Pseudocode Conventions**, [page 351](#). The precedence of pseudocode operations is further described in the Preface in **Operator Precedence**, [page 353](#).

Registers Altered—A summary of the PowerPC registers that are modified by executing the instruction.

Exceptions—A list of the exceptions that can occur as a result of executing the instruction. Asynchronous exceptions and exceptions associated with instruction fetching are not listed because those exceptions can occur with any instruction. This section also describes the effect of invalid instruction forms on instruction execution.

Compatibility—A brief description of instruction portability to other PowerPC implementations.

Instruction Encoding

All instructions are four bytes long and are word aligned. Bits 0:5 always contain the primary opcode, which is used to determine the instruction form. The instruction form defines fields within the encoding for identifying the operands. Some instruction forms define an extended opcode field for specifying additional instructions.

All instruction fields belong to one of the following categories:

- Defined

These instructions contain values, such as opcodes, that cannot be altered. The instruction encoding diagrams specify the values of defined fields. If any bit in a defined field does not contain the expected value, the instruction is illegal and an illegal-instruction exception occurs.

- Variable

These fields contain operands, such as general-purpose register identifiers or displacement values, that can vary from instruction to instruction. The instruction encoding diagrams specify the operands in variable fields.

- Reserved

Bits in a reserved field should be cleared to 0. In the instruction encoding diagrams, reserved fields are shaded and contain a value of 0. If any bit in a reserved field does not contain 0, the instruction form is invalid and its result is undefined. Unless otherwise noted, invalid instruction forms execute without causing an illegal-instruction exception.

Split-Field Notation

Some instructions contain a field with an encoding that is a permutation of the corresponding assembler operand. Such fields are called *split fields*. Split fields are used by instructions that move data between the general-purpose registers and the special-purpose registers, device-control registers, and the time-base registers. For these instructions, assembler operands and split fields are indicated as follows:

- In the **mf spr** and **mt spr** instructions, SPRN is the assembler operand and SPRF is the split field. SPRF corresponds to SPRN as follows:
 - $\text{SPRF}_{0:4}$ is equivalent to $\text{SPRN}_{5:9}$.
 - $\text{SPRF}_{5:9}$ is equivalent to $\text{SPRN}_{0:4}$.
- In the **mf dcr** and **mt dcr** instructions, DCRN is the assembler operand and DCRF is the split field. DCRF corresponds to DCRN as follows:
 - $\text{DCRF}_{0:4}$ is equivalent to $\text{DCRN}_{5:9}$.
 - $\text{DCRF}_{5:9}$ is equivalent to $\text{DCRN}_{0:4}$.
- In the **mf tb** instruction, TBRN is the assembler operand and TBRF is the split field. TBRF corresponds to TBRN as follows:
 - $\text{TBRF}_{0:4}$ is equivalent to $\text{TBRN}_{5:9}$.
 - $\text{TBRF}_{5:9}$ is equivalent to $\text{TBRN}_{0:4}$.

Throughout this document, references to SPRs, DCRs, and time-base registers use the respective SPRN, DCRN, and TBRN values. The assembler handles the conversion to the split-field format when encoding the instruction.

Alphabetical Instruction Listing

The following pages list the instructions supported by the PPC405 in alphabetical order.

add

Add

add	rD, rA, rB	(OE=0, Rc=0)
add.	rD, rA, rB	(OE=0, Rc=1)
addo	rD, rA, rB	(OE=1, Rc=0)
addo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	266	Rc
0	6	1	1	2	2	3
		1	6	1	2	1

Description

The sum of the contents of register **rA** and register **rB** is loaded into register **rD**.

Pseudocode

$$(rD) \leftarrow (rA) + (rB)$$

Registers Altered

- **rD**.
- **CR[CR0]_{LT, GT, EQ, SO}** if **Rc=1**.
- **XER[SO, OV]** if **OE=1**.

If an overflow occurs, it is possible that the contents of **CR0** do not reflect the infinitely precise result.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

addc

Add Carrying

addc	rD, rA, rB	(OE=0, Rc=0)
addc.	rD, rA, rB	(OE=0, Rc=1)
addco	rD, rA, rB	(OE=1, Rc=0)
addco.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	10	Rc
0	6	1	1	2	2	3
		1	6	1	2	1

Description

The sum of the contents of register **rA** and register **rB** is loaded into register **rD**. XER[CA] is updated to reflect the unsigned magnitude of the resulting sum.

Pseudocode

```

(rD) ← (rA) + (rB)
if (rD) > 232 - 1
  then XER[CA] ← 1
  else XER[CA] ← 0

```

Registers Altered

- **rD**.
- XER[CA].
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

adde

Add Extended

adde	rD, rA, rB	(OE=0, Rc=0)
adde.	rD, rA, rB	(OE=0, Rc=1)
addeo	rD, rA, rB	(OE=1, Rc=0)
addeo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	138	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The sum of the contents of register **rA**, register **rB**, and **XER[CA]** is loaded into register **rD**. **XER[CA]** is updated to reflect the unsigned magnitude of the resulting sum.

The add-extended instructions can be used to perform addition on integers larger than 32 bits, as described on [page 426](#).

Pseudocode

```

(rD) ← (rA) + (rB) + XER[CA]
if (rD) > 232 - 1
  then XER[CA] ← 1
  else XER[CA] ← 0

```

Registers Altered

- **rD**.
- **XER[CA]**.
- **CR[CR0]_{LT, GT, EQ, SO}** if **Rc=1**.
- **XER[SO, OV]** if **OE=1**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

addi

Add Immediate

addi rD, rA, SIMM

D Instruction Form

14	rD	rA	SIMM
0	6	1	1
		1	6
			3
			1

Description

If the rA field is 0, the SIMM field is sign-extended to 32 bits and loaded into register rD. If the rA field is nonzero, the SIMM field is sign-extended to 32 bits and added to the contents of register rA. The resulting sum is loaded into register rD.

Simplified mnemonics defined for this instruction are described in the following sections:

- **Load Address**, [page 870](#).
- **Load Immediate**, [page 870](#).
- **Subtract Instructions**, [page 867](#).

Pseudocode

$$(rD) \leftarrow (rA|0) + \text{EXTS}(\text{SIMM})$$

Registers Altered

- rD.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

addic

Add Immediate Carrying

addic rD, rA, SIMM

D Instruction Form

12	rD	rA	SIMM
0	6	1	1
		1	6
			3
			1

Description

The SIMM field is sign-extended to 32 bits and added to the contents of register rA. The resulting sum is loaded into register rD. XER[CA] is updated to reflect the unsigned magnitude of the resulting sum.

Simplified mnemonics defined for this instruction are described in **Subtract Instructions**, [page 867](#).

Pseudocode

```

(rD) ← (rA) + EXTS(SIMM)
if (rD) > 232 - 1
    then XER[CA] ← 1
    else XER[CA] ← 0

```

Registers Altered

- rD.
- XER[CA].

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

addic.

Add Immediate Carrying and Record

addic. rD, rA, SIMM

D Instruction Form

13	rD	rA	SIMM
0	6	1	1
		1	6
			3
			1

Description

The SIMM field is sign-extended to 32 bits and added to the contents of register rA. The resulting sum is loaded into register rD. XER[CA] is updated to reflect the unsigned magnitude of the resulting sum.

addic. is one of three instructions that implicitly update CR[CR0] without having an RC field. The other instructions are **andi.** and **andis..**

Simplified mnemonics defined for this instruction are described in **Subtract Instructions**, [page 867](#).

Pseudocode

```

(rD) ← (rA) + EXTS(SIMM)
if (rD) > 232 - 1
  then XER[CA] ← 1
  else XER[CA] ← 0

```

Registers Altered

- rD.
- XER[CA].
- CR[CR0]_{LT, GT, EQ, SO}.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

addis

Add Immediate Shifted

addis rD, rA, SIMM

D Instruction Form

15	rD	rA	SIMM
0	6	1	1
		1	6
			3
			1

Description

If the **rA** field is 0, the **SIMM** field is concatenated on the right with sixteen 0-bits and the result is loaded into register **rD**. If the **rA** field is nonzero, the **SIMM** field is concatenated on the right with sixteen 0-bits and the result is added to the contents of register **rA**. The resulting sum is loaded into register **rD**.

Simplified mnemonics defined for this instruction are described in the following sections:

- **Load Immediate**, page 870.
- **Subtract Instructions**, page 867.

An **addis** instruction followed by an **ori** instruction can be used to load an arbitrary 32-bit value in a GPR, as shown in the following example:

```
addis    rD, 0, high 16 bits of value
ori      rD, rD, low 16 bits of value
```

Pseudocode

$$(rD) \leftarrow (rA[0]) + (SIMM \parallel 160)$$

Registers Altered

- **rD**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

addme

Add to Minus One Extended

addme	rD, rA	(OE=0, Rc=0)
addme.	rD, rA	(OE=0, Rc=1)
addmeo	rD, rA	(OE=1, Rc=0)
addmeo.	rD, rA	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	0	0	0	0	0	OE	234	Rc
0	6	1	1					2 2		3
		1	6					1 2		1

Description

The sum of the contents of register **rA**, the XER[CA] bit, and the value -1 is loaded into register **rD**. XER[CA] is updated to reflect the unsigned magnitude of the resulting sum.

The add-extended instructions can be used to perform addition on integers larger than 32 bits, as described on [page 426](#).

Pseudocode

```

(rD) ← (rA) + XER[CA] + (−1)
if (rD)  $\geq 2^{32} - 1$ 
  then XER[CA] ← 1
  else XER[CA] ← 0

```

Registers Altered

- **rD**.
- XER[CA].
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

addze

Add to Zero Extended

addze	rD, rA	(OE=0, Rc=0)
addze.	rD, rA	(OE=0, Rc=1)
addzeo	rD, rA	(OE=1, Rc=0)
addzeo.	rD, rA	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	0 0 0 0 0	OE	202	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The sum of the contents of register rA and XER[CA] is loaded into register rD. XER[CA] is updated to reflect the unsigned magnitude of the resulting sum.

The add-extended instructions can be used to perform addition on integers larger than 32 bits, as described on [page 426](#).

Pseudocode

```

(rD) ← (rA) + XER[CA]
if (rD) > 232 - 1
  then XER[CA] ← 1
  else XER[CA] ← 0

```

Registers Altered

- rD.
- XER[CA].
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

and

AND

and rA, rS, rB (Rc=0)
and. rA, rS, rB (Rc=1)

X Instruction Form

31	rS	rA	rB	28	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The contents of register **rS** are ANDed with the contents of register **rB** and the result is loaded into register **rA**.

Pseudocode

$$(rA) \leftarrow (rS) \wedge (rB)$$

Registers Altered

- **rA**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

andc

AND with Complement

andc rA, rS, rB (Rc=0)

andc. rA, rS, rB (Rc=1)

X Instruction Form

31	rS	rA	rB	60	Rc
0	6	1	1	2 2	3
		1	6	1	1

Description

The contents of register **rS** are ANDed with the one's complement of the contents of register **rB** and the result is loaded into register **rA**.

Pseudocode

$$(rA) \leftarrow (rS) \wedge \neg(rB)$$

Registers Altered

- **rA**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

andi.

AND Immediate

andi. rA, rS, UIMM

D Instruction Form

28	rS	rA	UIMM
0	6	1	1
		1	6
			3
			1

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the left. The contents of register rS are ANDed with the extended UIMM field and the result is loaded into register rA.

andi. is one of three instructions that implicitly update CR[CR0] without having an Rc field. The other instructions are **addic.** and **andis..**

The **andi.** instruction can be used to test whether any of the 16 least-significant bits in a GPR are 1-bits.

Pseudocode

$$(rA) \leftarrow (rS) \wedge (^{16}0 \parallel UIMM)$$

Registers Altered

- rA.
- CR[CR0]_{LT, GT, EQ, SO}.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

andis.

AND Immediate Shifted

andis. rA, rS, UIMM

D Instruction Form

29	rS	rA	UIMM
0	6	1	3
		1	1

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the right. The contents of register **rS** are ANDed with the extended UIMM field and the result is loaded into register **rA**.

andis. is one of three instructions that implicitly update CR[CR0] without having an Rc field. The other instructions are **addic.** and **andi..**

The **andis.** instruction can be used to test whether any of the 16 most-significant bits in a GPR are 1-bits.

Pseudocode

$$(rA) \leftarrow (rS) \wedge (UIMM \parallel 160)$$

Registers Altered

- **rA.**
- CR[CR0]_{LT, GT, EQ, SO}.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

b

Branch

b	target	(AA=0, LK=0)
ba	target	(AA=1, LK=0)
bl	target	(AA=0, LK=1)
bla	target	(AA=1, LK=1)

I Instruction Form

18		LI	AA	LK
0	6		3	3
			0	1

Description

target is a 32-bit operand that specifies a displacement to the branch-target address. The assembler sets the instruction-opcode LI field to the value of *target*_{6:29}.

The next instruction address (NIA) is the effective address of the branch target. The NIA is calculated by adding the displacement to a base address, which are formed as follows:

- The displacement is obtained by concatenating two 0-bits to the right of the BD field and sign-extending the result to 32 bits.
- If the AA field contains 0 (relative addressing), the branch-instruction address is used as the base address. The branch-instruction address is the current instruction address (CIA).
- If the AA field contains 1 (absolute addressing), the base address is 0.

Program flow is transferred to the NIA. If the LK field contains 1, then the address of the instruction following the branch instruction (CIA + 4) is loaded into the LR.

Pseudocode

```

If AA = 1
  then NIA ← EXTS(LI || 0b00)
  else NIA ← CIA + EXTS(LI || 0b00)
if LK = 1 then
  (LR) ← CIA + 4

```

Registers Altered

- LR if LK=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

bc**Branch Conditional**

bc	BO, BI, target	(AA=0, LK=0)
bca	BO, BI, target	(AA=1, LK=0)
bcl	BO, BI, target	(AA=0, LK=1)
bcla	BO, BI, target	(AA=1, LK=1)

B Instruction Form

16	BO	BI	BD	AA	LK
0	6	11	16	30	31

Description

target is a 32-bit operand that specifies a displacement to the branch-target address. The assembler sets the instruction-opcode BD field to the value of *target*_{16:29}.

The next instruction address (NIA) is the effective address of the branch target. The NIA is calculated by adding the displacement to a base address, which are formed as follows:

- The displacement is obtained by concatenating two 0-bits to the right of the BD field and sign-extending the result to 32 bits.
- If the AA field contains 0 (relative addressing), the branch-instruction address is used as the base address. The branch-instruction address is the current instruction address (CIA).
- If the AA field contains 1 (absolute addressing), the base address is 0.

Program flow is transferred to the NIA. If the LK field contains 1, then the address of the instruction following the branch instruction (CIA + 4) is loaded into the LR.

The BO field specifies whether the branch is conditional on the contents of the CTR and/or the CR registers and how those conditions are tested. The BO field also specifies whether the CTR is decremented. The encoding of the BO field is described in **Conditional Branch Control**, page 403. The BI field specifies which CR bit is tested if the branch is conditional on the CR register.

Simplified mnemonics defined for this instruction are described in the following sections:

- ?<Fill in list after appendix is built>

Pseudocode

```

if BO2 = 0 then
    CTR ← CTR - 1
    CTR_cond_met ← BO2 ∨ ((CTR ≠ 0) ⊕ BO3)
    CR_cond_met ← BO0 ∨ (CRBI = BO1)
    if CTR_cond_met ∧ CR_cond_met
    then if AA = 1
        then NIA ← EXTS(BD || 0b00)
        else NIA ← CIA + EXTS(BD || 0b00)
    else NIA ← CIA + 4
    if LK = 1 then
        (LR) ← CIA + 4

```

Registers Altered

- CTR if BO₂=0.

- LR if LK=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

bcctr

Branch Conditional to Count Register

bcctr	BO, BI	(LK=0)
bcctrl	BO, BI	(LK=1)

XL Instruction Form

19	BO	BI	0 0 0 0 0	528	LK
0	6	1	1	2	3
		1	6	1	1

Description

The next instruction address (NIA) is the effective address of the branch target. The NIA is formed by concatenating the 30 most-significant bits of the CTR with two 0-bits on the right. Program flow is transferred to the NIA. If the LK field contains 1, then the address of the instruction following the branch instruction (CIA + 4) is loaded into the LR.

The BO field specifies whether the branch is conditional on the contents of the CTR and/or the CR registers and how those conditions are tested. The BO field also specifies whether the CTR is decremented. The encoding of the BO field is described in **Conditional Branch Control**, page 403. The BI field specifies which CR bit is tested if the branch is conditional on the CR register.

Simplified mnemonics defined for this instruction are described in the following sections:

- ?<Fill in list after appendix is built>

Pseudocode

```

if BO2 = 0 then
    CTR ← CTR - 1
    CTR_cond_met ← BO2 ∨ ((CTR ≠ 0) ⊕ BO3)
    CR_cond_met ← BO0 ∨ (CRBI = BO1)
    if CTR_cond_met ∧ CR_cond_met
    then NIA ← CTR0:29 || 0b00
    else NIA ← CIA + 4
    if LK = 1 then
        (LR) ← CIA + 4

```

Registers Altered

- CTR if BO₂=0.
- LR if LK=1.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- BO₂=0. In this case the branch is taken if the branch condition is true. The contents of

the decremented CTR are used as the NIA.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

bclr

Branch Conditional to Link Register

bclr	BO, BI	(LK=0)
bclrl	BO, BI	(LK=1)

XL Instruction Form

19	BO	BI	0 0 0 0 0	16	LK
0	6	1	1	2	3
		1	6	1	1

Description

The next instruction address (NIA) is the effective address of the branch target. The NIA is formed by concatenating the 30 most-significant bits of the LR with two 0-bits on the right. Program flow is transferred to the NIA. If the LK field contains 1, then the address of the instruction following the branch instruction (CIA + 4) is loaded into the LR.

The BO field specifies whether the branch is conditional on the contents of the CTR and/or the CR registers and how those conditions are tested. The BO field also specifies whether the CTR is decremented. The encoding of the BO field is described in **Conditional Branch Control**, page 403. The BI field specifies which CR bit is tested if the branch is conditional on the CR register.

Simplified mnemonics defined for this instruction are described in the following sections:

- ?<Fill in list after appendix is built>

Pseudocode

```

if BO2 = 0 then
    CTR ← CTR - 1
    CTR_cond_met ← BO2 ∨ ((CTR ≠ 0) ⊕ BO3)
    CR_cond_met ← BO0 ∨ (CRBI = BO1)
    if CTR_cond_met ∧ CR_cond_met
    then NIA ← LR0:29 || 0b00
    else NIA ← CIA + 4
    if LK = 1 then
        (LR) ← CIA + 4

```

Registers Altered

- CTR if BO₂=0.
- LR if LK=1.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

cmp

Compare

cmp crfD, 0, rA, rB

X Instruction Form

31	crfD	0 0	rA	rB	0	0
0	6	9	1	1	2	3
			1	6	1	1

Description

A 32-bit signed comparison is performed between the contents of register **rA** and register **rB**. **crfD** which CR field is updated to reflect the comparison results. The value of XER[SO] is loaded into the same CR field.

Simplified mnemonics defined for this instruction are described in **Compare Instructions**, [page 864](#).

Pseudocode

```

c0:3 ← 0b0000
if (rA) < (rB) then c0 ← 1
if (rA) > (rB) then c1 ← 1
if (rA) = (rB) then c2 ← 1
c3 ← XER[SO]
n ← crfD
CR[CRn] ← c0:3

```

Registers Altered

- CR[CRn] as specified by the **crfD** field.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

cmpi

Compare Immediate

cmpi crfD, 0, rA, SIMM

D Instruction Form

11	crfD	0 0	rA	SIMM
0	6	9	11	16
				31

Description

The SIMM field is sign-extended to 32 bits. A 32-bit signed comparison is performed between the contents of register rA and the sign-extended SIMM field. crfD specifies which CR field is updated to reflect the comparison results. The value of XER[SO] is loaded into the same CR field.

Simplified mnemonics defined for this instruction are described in [Compare Instructions, page 864](#).

Pseudocode

```

c0:3 ← 0b0000
if (rA) < EXTS(SIMM) then c0 ← 1
if (rA) > EXTS(SIMM) then c1 ← 1
if (rA) = EXTS(SIMM) then c2 ← 1
c3 ← XER[SO]
n ← crfD
CR[CRn] ← c0:3

```

Registers Altered

- CR[CR_n] as specified by the crfD field.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

cml

Compare Logical

cml **crfD**, 0, **rA**, **rB**

X Instruction Form

31	crfD	0 0	rA	rB	32	0
0	6	9	1	1	2	3
			1	6	1	1

Description

A 32-bit unsigned comparison is performed between the contents of register **rA** and register **rB**. **crfD** specifies which CR field is updated to reflect the comparison results. The value of XER[SO] is loaded into the same CR field.

Simplified mnemonics defined for this instruction are described in **Compare Instructions**, [page 864](#).

Pseudocode

```

c0:3 ← 0b0000
if (rA) < (rB) then c0 ← 1
if (rA) > (rB) then c1 ← 1
if (rA) = (rB) then c2 ← 1
c3 ← XER[SO]
n ← crfD
CR[CRn] ← c0:3

```

Registers Altered

- CR[CRn] as specified by the **crfD** field.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

cmpli

Compare Logical Immediate

cmpli **crfD**, 0, **rA**, **UIMM**

D Instruction Form

10	crfD	0 0	rA	UIMM
0	6	9	1 1	1 6 3 1

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the left. A 32-bit unsigned comparison is performed between the contents of register **rA** and the zero-extended UIMM field. **crfD** specifies which CR field is updated to reflect the comparison results. The value of XER[SO] is loaded into the same CR field.

Simplified mnemonics defined for this instruction are described in **Compare Instructions**, [page 864](#).

Pseudocode

```

c0:3 ← 0b0000
if (rA) < (160 || UIMM)then  c0 ← 1
if (rA) > (160 || UIMM)then  c1 ← 1
if (rA) = (160 || UIMM)then  c2 ← 1
c3 ← XER[SO]
n ← crfD
CR[CRn] ← c0:3

```

Registers Altered

- CR[CRn] as specified by the **crfD** field.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

cntlzw

Count Leading Zeros Word

cntlzw rA, rS (Rc=0)

cntlzw. rA, rS (Rc=1)

X Instruction Form

31	rS	rA	0 0 0 0 0	26	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The consecutive leading 0 bits in register rS are counted and the count is loaded into register rA. This count ranges from 0 through 32, inclusive.

Pseudocode

```

n ← 0
do while n < 32
  if (rS)n = 1 then leave
  n ← n + 1
(rA) ← n

```

Registers Altered

- rA.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

crand

Condition Register AND

crand **crbD, crbA, crbB**

XL Instruction Form

19	crbD	crbA	crbB	257	0
0	6	1	1	2	3
		1	6	1	1

Description

The CR bit specified by **crbA** is ANDed with the CR bit specified by **crbB** and the result is loaded into the CR bit specified by **crbD**.

Pseudocode

$$CR[crbD] \leftarrow CR[crbA] \wedge CR[crbB]$$

Registers Altered

- CR.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

crandc

Condition Register AND with Complement

crandc **crbD, crbA, crbB**

XL Instruction Form

19	crbD	crbA	crbB	129	0
0	6	1	1	2	3
		1	6	1	1

Description

The CR bit specified by **crbA** is ANDed with the one's complement of the CR bit specified by **crbB** and the result is loaded into the CR bit specified by **crbD**.

Pseudocode

$$\text{CR}[\text{crbD}] \leftarrow \text{CR}[\text{crbA}] \wedge \neg \text{CR}[\text{crbB}]$$

Registers Altered

- CR.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

creqv

Condition Register Equivalent

creqv **crbD**, **crbA**, **crbB**

XL Instruction Form

19	crbD	crbA	crbB	289	0
0	6	1	1	2	3
		1	6	1	1

Description

The CR bit specified by **crbA** is XORed with the CR bit specified by **crbB** and the one's complement of the result is loaded into the CR bit specified by **crbD**.

Simplified mnemonics defined for this instruction are described in **CR-Logical Instructions**, [page 864](#).

Pseudocode

$$CR[crbD] \leftarrow \neg(CR[crbA] \oplus CR[crbB])$$

Registers Altered

- CR.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

crnand

Condition Register NAND

crnand crbD, crbA, crbB

XL Instruction Form

19	crbD	crbA	crbB	225	0
0	6	1	1	2	3
		1	6	1	1

Description

The CR bit specified by **crbA** is ANDed with the CR bit specified by **crbB** and the one's complement of the result is loaded into the CR bit specified by **crbD**.

Pseudocode

$$\text{CR}[\text{crbD}] \leftarrow \neg(\text{CR}[\text{crbA}] \wedge \text{CR}[\text{crbB}])$$

Registers Altered

- CR.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

crnor

Condition Register NOR

crnor **crbD**, **crbA**, **crbB**

XL Instruction Form

19	crbD	crbA	crbB	33	0
0	6	1	1	2	3
		1	6	1	1

Description

The CR bit specified by **crbA** is ORed with the CR bit specified by **crbB** and the one's complement of the result is loaded into the CR bit specified by **crbD**.

Simplified mnemonics defined for this instruction are described in **CR-Logical Instructions**, [page 864](#).

Pseudocode

$$CR[crbD] \leftarrow \neg(CR[crbA] \vee CR[crbB])$$

Registers Altered

- CR.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

cror

Condition Register OR

cror **crbD, crbA, crbB**

XL Instruction Form

19	crbD	crbA	crbB	449	0
0	6	1	1	2	3
		1	6	1	1

Description

The CR bit specified by **crbA** is ORed with the CR bit specified by **crbB** and the result is loaded into the CR bit specified by **crbD**.

Simplified mnemonics defined for this instruction are described in **CR-Logical Instructions**, [page 864](#).

Pseudocode

$$CR[crbD] \leftarrow CR[crbA] \vee CR[crbB]$$

Registers Altered

- CR.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

crorc

Condition Register OR with Complement

crorc **crbD**, **crbA**, **crbB**

XL Instruction Form

19	crbD	crbA	crbB	417	0
0	6	1	1	2	3
		1	6	1	1

Description

The CR bit specified by **crbA** is ORed with the one's complement of the CR bit specified by **crbB** and the result is loaded into the CR bit specified by **crbD**.

Pseudocode

$$CR[crbD] \leftarrow CR[crbA] \vee \neg CR[crbB]$$

Registers Altered

- CR.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

crxor

Condition Register XOR

crxor crbD, crbA, crbB

XL Instruction Form

19	crbD	crbA	crbB	193	0
0	6	1	1	2	3
		1	6	1	1

Description

The CR bit specified by **crbA** is XORed with the CR bit specified by **crbB** and the result is loaded into the CR bit specified by **crbD**.

Simplified mnemonics defined for this instruction are described in **CR-Logical Instructions**, [page 864](#).

Pseudocode

$$\text{CR}[\text{crbD}] \leftarrow \text{CR}[\text{crbA}] \oplus \text{CR}[\text{crbB}]$$

Registers Altered

- CR.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

dcba

Data Cache Block Allocate

dcba rA, rB

X Instruction Form

31	0 0 0 0 0	rA	rB	758	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The operation of this instruction depends on the cachability and caching policy of EA as follows:

- If EA is cached by the data cache and has a write-back caching policy, the value of all bytes in the data cacheline referenced by EA become undefined. The data cacheline remains valid.
- If EA is not cached but is cachable with a write-back caching policy, a corresponding data cacheline is allocated and the value of the bytes in that line are undefined.
- If EA is cachable and has a write-through caching policy, a no-operation occurs. This is true whether or not EA is cached by the data cache.
- If EA is not cachable, a no-operation occurs.

dcba provides a hint that a block of memory is either no longer needed, or will soon be written. There is no need to retain the data in the memory block. Establishing a data cacheline without reading from main memory can improve performance.

dcba establishes an address in the data cache without copying data from main memory. Software must ensure that the established address does not represent an invalid main-memory address. A subsequent operation could cause the processor to attempt a write of the cacheline to the invalid main-memory address, possibly causing a machine-check exception to occur.

Pseudocode

$EA \leftarrow (rA|0) + (rB)$

Allocate data cacheline corresponding to EA

Registers Altered

- None.

Exceptions

This instruction is considered a “store” with respect to data-access exceptions. However, this instruction does not cause data storage exceptions or data TLB-miss exceptions. If conditions occur that would otherwise cause these exceptions, **dcba** is treated as a no-operation. This instruction is also considered a “store” with respect to data address-compare (DAC) debug exceptions. Debug exceptions can occur as a result of executing this instruction.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture. Implementation of this instruction is optional, and it is not guaranteed to be implemented on all PowerPC processors.

dcbf

Data Cache Block Flush

dcbf rA, rB

X Instruction Form

31	0 0 0 0 0	rA	rB	86	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

If EA is cached by the data cache, the corresponding data cacheline is invalidated. If the data cacheline is marked as modified, the contents of the cacheline are written (flushed) to main memory prior to the invalidation. The flush operation is performed whether or not the corresponding storage attribute indicates EA is cachable. If EA is not cached, no operation is performed.

Pseudocode

```
EA ← (rA[0] + (rB)
Flush data cacheline corresponding to EA
```

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

This instruction is considered a “load” with respect to the above data-access exceptions. It is considered a “store” with respect to data address-compare (DAC) debug exceptions. Debug exceptions can occur as a result of executing this instruction.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture.

dcbi

Data Cache Block Invalidate

dcbi rA, rB

X Instruction Form

31	0 0 0 0 0	rA	rB	470	0
0	6	1	1	2	3
		1	6	1	1

Description

This is a privileged instruction.

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

If EA is cached by the data cache, the corresponding data cacheline is invalidated. The invalidation is performed whether or not the corresponding storage attribute indicates EA is cachable. If modified data exists in the cacheline, it is lost. If EA is not cached, no operation is performed.

Pseudocode

$EA \leftarrow (rA|0) + (rB)$

Invalidate data cacheline corresponding to EA

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.

A data-storage exception occurs if the U0 storage attribute associated with the EA is set to 1 and U0 exceptions are enabled (CCR0[U0XE]=1).

- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.
- Program—Attempted execution of this instruction from user mode.

This instruction is considered a “store” with respect to the above data-access exceptions. It is also considered a “store” with respect to data address-compare (DAC) debug exceptions. Debug exceptions can occur as a result of executing this instruction.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the operating-environment architecture level (OEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture.

dcbst

Data Cache Block Store

dcbst rA, rB

X Instruction Form

31	0 0 0 0 0	rA	rB	54	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

If EA is cached by the data cache, the corresponding data cacheline is checked to see if it is marked modified. If it is modified, it is stored to main memory and marked as unmodified. The store operation is performed whether or not the corresponding storage attribute indicates EA is cachable. No operation occurs if the data cacheline is unmodified, or if EA is not cached.

Pseudocode

```
EA ← (rA|0) + (rB)
Store modified data cacheline corresponding to EA
```

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

This instruction is considered a “load” with respect to the above data-access exceptions. It is considered a “store” with respect to data address-compare (DAC) debug exceptions. Debug exceptions can occur as a result of executing this instruction.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture.

dcbt

Data Cache Block Touch

dcbt rA, rB

X Instruction Form

31	0 0 0 0 0	rA	rB	278	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

If EA is cachable but not in the data cache, the corresponding cacheline is loaded into the data cache from main memory. If EA is already cached, or if the storage attributes indicate EA is not cachable, no operation is performed.

This instruction is a hint to the processor that the program will likely load data from EA in the near future. The processor can potentially improve performance by loading the cacheline into the data cache.

Pseudocode

$EA \leftarrow (rA|0) + (rB)$
 Prefetch data cacheline corresponding to EA

Registers Altered

- None.

Exceptions

This instruction is considered a “load” with respect to data-storage exceptions. However, this instruction does not cause data storage exceptions or data TLB-miss exceptions. If conditions occur that would otherwise cause these exceptions, **dcbt** is treated as a no-operation. This instruction is also considered a “load” with respect to data address-compare (DAC) debug exceptions. Debug exceptions can occur as a result of executing this instruction.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture.

dcbtst

Data Cache Block Touch for Store

dcbtst rA, rB

X Instruction Form

31	0 0 0 0 0	rA	rB	246	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

If EA is cachable but not in the data cache, the corresponding cacheline is loaded into the data cache from main memory. If EA is already cached, or if the storage attributes indicate EA is not cachable, no operation is performed.

This instruction is a hint to the processor that the program will likely store data to the EA in the near future. The processor can potentially improve performance by loading the cacheline into the data cache. In the PPC405, this instruction operates identically to **dcbt**. In other PowerPC implementations, this instruction can cause unique bus cycles to occur and additional cache-coherency state can be associated with the cacheline.

Pseudocode

$$EA \leftarrow (rA|0) + (rB)$$

Prefetch data cacheline corresponding to EA

Registers Altered

- None.

Exceptions

This instruction is considered a “load” with respect to data-storage exceptions. However, this instruction does not cause data storage exceptions or data TLB-miss exceptions. If conditions occur that would otherwise cause these exceptions, **dcbtst** is treated as a no-operation. This instruction is also considered a “load” with respect to data address-compare (DAC) debug exceptions. Debug exceptions can occur as a result of executing this instruction.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture.

dcbz

Data Cache Block Set to Zero

dcbz rA, rB

X Instruction Form

31	0 0 0 0 0	rA	rB	1014	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The operation of this instruction depends on the cachability and caching policy of EA as follows:

- If EA is cached by the data cache and has a write-back caching policy, the value of all bytes in the data cacheline referenced by EA are cleared to 0. The data cacheline is marked modified.
- If EA is not cached but is cachable with a write-back caching policy, a corresponding data cacheline is allocated and the value of the bytes in that line are cleared to 0. The data cacheline is marked modified.
- If EA is cachable and has a write-through caching policy, an alignment exception occurs. This is true whether or not EA is cached.
- If EA is not cachable, an alignment exception occurs.

dcbz establishes an address in the data cache without copying data from main memory. Software must ensure that the established address does not represent an invalid main-memory address. A subsequent operation could cause the processor to attempt a write of the cacheline to the invalid main-memory address, possibly causing a machine-check exception to occur.

Pseudocode

```
EA ← (rA[0] + (rB))
Clear contents of data cacheline corresponding to EA
```

Registers Altered

- None.

Exceptions

- Alignment—if the EA is marked as non-cachable or write-through. The alignment exception handler can emulate the effect of this instruction by storing zeros to the corresponding block of main memory.

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.

A data-storage exception occurs if the U0 storage attribute associated with the EA is set to 1 and U0 exceptions are enabled (CCR0[U0XE]=1).

- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

This instruction is considered a “store” with respect to the above data-access exceptions. It is also considered a “store” with respect to data address-compare (DAC) debug exceptions. Debug exceptions can occur as a result of executing this instruction.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture.

dccc*i*

Data Cache Congruence Class Invalidate

dccc*i* rA, rB

X Instruction Form

31	0 0 0 0 0	rA	rB	454	0
0	6	1	1	2	3
		1	6	1	1

Description

This is a privileged instruction.

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

Both data cachelines in the congruence class specified by EA_{19:26} are invalidated. The invalidation is performed whether or not the corresponding storage attribute indicates EA is cachable. The invalidation is also performed whether or not EA is cached in either line. If modified data exists in the cachelines, it is lost.

This instruction is intended for use during initialization to invalidate the entire data cache before is enabled. A sequence of **dccc*i*** instructions should be executed, one for each congruence class. Afterwards, cachability can be enabled.

Pseudocode

$$EA \leftarrow (rA|0) + (rB)$$

Invalidate the data-cache congruence class specified by EA_{19:26}

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.

A data-storage exception occurs if the U0 storage attribute associated with the EA is set to 1 and U0 exceptions are enabled (CCR0[U0XE]=1).

- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.
- Program—Attempted execution of this instruction from user mode.

This instruction is considered a “store” with respect to the above data-access exceptions. It can cause data-access exceptions related to the EA even though the instruction is not address specific (multiple addresses are selected by a single EA). This instruction does not cause data address-compare (DAC) debug exceptions.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

dcread

Data Cache Read

dcread rD, rA, rB

X Instruction Form

31	rD	rA	rB	486	0
0	6	1	1	2	3
		1	6	1	1

Description

This is a privileged instruction.

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

This instruction can be used as a data-cache debugging aid. It is used to read information for a specific data cacheline. The cache information is loaded in register **rD**.

EA_{19:26} is used to specify a congruence class within the data cache. CCR0[CWS] is used to select one of the two cachelines within the congruence class. If CCR0[CWS]=0, the line in way A is selected. If CCR0[CWS]=1, the line in way B is selected.

If CCR0[CIS]=0, the information read is a word of data from the selected cacheline. EA_{27:29} is used as an index to select the word from the 32-byte line. If CCR0[CIS]=1, the information read is the tag associated with the selected cacheline.

Following execution of this instruction, **rD** contains the following:

Bit	Name	Function	Description
0:18	INFO	Data-Cache Information CCR0[CIS]=0—Data word. CCR0[CIS]=1—Data tag.	Contains either the cache-line tag or a single data word from the cacheline. If a data word is loaded it is specified using effective-address bits EA _{27:29} . CCR0[CIS] controls the type of information loaded into this field.
19:25		Reserved	
26	D	Dirty 0—Cacheline is not dirty. 1—Cacheline is dirty.	Contains a copy of the cache-line dirty bit indicating whether or not the line contains modified data.

Bit	Name	Function	Description
27	V	Valid 0—Cacheline is not valid. 1—Cacheline is valid.	Contains a copy of the cache-line valid bit.
28:30		Reserved	
31	LRU	Least-Recently Used 0—Way A is least-recently used. 1—Way B is least-recently used.	Contains the LRU bit for the congruence class associated with the cacheline.

Pseudocode

```

EA ← (rA|0) + (rB)
if ((CCR0[CIS] = 0) ∧ (CCR0[CWS] = 0)) then (rD) ← (data-cache data, way A)
if ((CCR0[CIS] = 0) ∧ (CCR0[CWS] = 1)) then (rD) ← (data-cache data, way B)
if ((CCR0[CIS] = 1) ∧ (CCR0[CWS] = 0)) then (rD) ← (data-cache tag, way A)
if ((CCR0[CIS] = 1) ∧ (CCR0[CWS] = 1)) then (rD) ← (data-cache tag, way B)

```

Registers Altered

- rD.

Exceptions

- Alignment—if the EA is not aligned on a word boundary ($EA_{30:31} \neq 00$).
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.
- Program—Attempted execution of this instruction from user mode.

This instruction is considered a “load” with respect to the above data-access exceptions. It can cause data TLB-miss exceptions related to EA even though the instruction is not address specific (multiple addresses are selected by a single EA). This instruction cannot cause data-storage exceptions. This instruction does not cause data address-compare (DAC) debug exceptions.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

divw

Divide Word

divw	rD, rA, rB	(OE=0, Rc=0)
divw.	rD, rA, rB	(OE=0, Rc=1)
divwo	rD, rA, rB	(OE=1, Rc=0)
divwo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	491	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The contents of register **rA** (dividend) are divided by the contents of register **rB** (divisor). The quotient is loaded into register **rD**. Both the dividend and the divisor are interpreted as signed integers. The quotient is the unique signed integer that satisfies the equation:

$$\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}$$

where the remainder has the same sign as the dividend, and:

- $0 \leq \text{remainder} < |\text{divisor}|$, if the dividend is positive.
- $-|\text{divisor}| < \text{remainder} \leq 0$, if the dividend is negative.

The 32-bit remainder can be calculated using the following sequence of instructions:

divw	rD, rA, rB	# rD = quotient
mullw	rD, rD, rB	# rD = quotient $\times$ divisor
subf	rD, rD, rA	# rD = remainder

The contents of register **rD** are undefined if an attempt is made to perform either of the following invalid divisions:

- $0x8000\ 0000 \div -1$.
- $n \div 0$, where n is any number.

The contents of **CR[CR0]_{LT, GT, EQ}** are undefined if the **Rc** field is set to 1 and an invalid division is performed. Both invalid divisions set **XER[OV, SO]** to 1 if the **OE** field contains 1.

Pseudocode

$$(\text{rD}) \leftarrow (\text{rA}) \div (\text{rB})$$

Registers Altered

- **rD**.
- **CR[CR0]_{LT, GT, EQ, SO}** if **Rc**=1.
- **XER[OV, SO]** if **OE**=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

divwu

Divide Word Unsigned

divwu	rD, rA, rB	(OE=0, Rc=0)
divwu.	rD, rA, rB	(OE=0, Rc=1)
divwuo	rD, rA, rB	(OE=1, Rc=0)
divwuo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	459	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The contents of register **rA** (dividend) are divided by the contents of register **rB** (divisor). The quotient is loaded into register **rD**. Both the dividend and the divisor are interpreted as unsigned integers. The quotient is the unique unsigned integer that satisfies the equation:

$$\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}$$

where $0 \leq \text{remainder} < \text{divisor}$.

The 32-bit unsigned remainder can be calculated using the following sequence of instructions:

divwu	rD, rA, rB	# rD = quotient
mullw	rD, rD, rB	# rD = quotient $\times$ divisor
subf	rD, rD, rA	# rD = remainder

If **Rc=1**, **CR[CR0]_{LT, GT, EQ}** are set using a signed comparison of the result to 0 even though the instruction produces an unsigned integer as a quotient.

The contents of register **rD** are undefined if an attempt is made to perform the invalid division $n \div 0$ (where n is any number). The contents of **CR[CR0]_{LT, GT, EQ}** are undefined if the **Rc** field is set to 1 and an invalid division is performed. An invalid division sets **XER[OV, SO]** to 1 if the **OE** field contains 1.

Pseudocode

$$(\mathbf{rD}) \leftarrow (\mathbf{rA}) \div (\mathbf{rB})$$

Registers Altered

- **rD**.
- **CR[CR0]_{LT, GT, EQ, SO}** if **Rc=1**.
- **XER[OV, SO]** if **OE=1**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

eieio

Enforce In Order Execution of I/O

eieio

X Instruction Form

31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	854	0
0	6			2	3
				1	1

Description

The **eieio** instruction enforces ordering of load and store operations. It ensures that all loads and stores preceding **eieio** in program order complete with respect to main memory before loads and stores following **eieio** access main memory. It is intended for use in managing shared data structures, in accessing memory-mapped I/O, and in preventing load/store combining operations in main memory.

With the exception of the **dcba** and **dcbz** instructions, **eieio** does not affect the order of cache operations. This is true whether the cache operation is initiated explicitly by the execution of a cache-control instruction, or implicitly during the normal operation of the cache controller.

eieio orders memory access, not instruction completion. Non-memory instructions following **eieio** can complete before the memory operations ordered by **eieio**. The **sync** instruction is used to guarantee ordering of both instruction completion and storage access. The PPC405 implements **eieio** and **sync** identically (this is permitted by the PowerPC architecture). Programmers should use the appropriate ordering instruction to maximize the performance of software that is portable between various PowerPC implementations.

Pseudocode

Force prior memory accesses to complete before starting subsequent accesses

Registers Altered

- None.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture and the PowerPC embedded-environment architecture. The instruction is not part of the PowerPC Book-E architecture.

eqv

Equivalent

eqv	rA, rS, rB	(Rc=0)
eqv.	rA, rS, rB	(Rc=1)

X Instruction Form

31	rS	rA	rB	284	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The contents of register **rS** are XORed with the contents of register **rB**. A one's complement of the result is calculated and loaded in register **rA**.

Pseudocode

$$(rA) \leftarrow \neg((rS) \oplus (rB))$$

Registers Altered

- **rA**.
- **CR[CR0]_{LT, GT, EQ, SO}** if **Rc=1**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

extsb

Extend Sign Byte

extsb rA, rS (Rc=0)
extsb. rA, rS (Rc=1)

X Instruction Form

31	rS	rA	0 0 0 0 0	954	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The least-significant byte of register rS is sign-extended to 32 bits by replicating bit rS₂₄ into bits 0 through 23 of the result. The result is loaded into register rA.

Pseudocode

(rA) ← EXTS(rS_{24:31})

Registers Altered

- rA.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

extsh

Extend Sign Halfword

extsh **rA, rS** (**Rc=0**)
extsh. **rA, rS** (**Rc=1**)

X Instruction Form

31	rS	rA	0 0 0 0 0	922	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The least-significant halfword of register **rS** is sign-extended to 32 bits by replicating bit **rS₁₆** into bits 0 through 15 of the result. The result is loaded into register **rA**.

Pseudocode

$(rA) \leftarrow \text{EXTS}(rS_{16:31})$

Registers Altered

- **rA**.
- **CR[CR0]_{LT, GT, EQ, SO}** if **Rc=1**.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

icbi

Instruction Cache Block Invalidate

icbi **rA, rB****X Instruction Form**

31	0 0 0 0 0	rA	rB	982	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

If EA is cached by the instruction cache, the corresponding instruction cacheline is invalidated. The invalidation is performed whether or not the corresponding storage attribute indicates EA is cachable. If EA is not cached, no operation is performed.

Pseudocode

$$EA \leftarrow (rA[0] + (rB))$$

Invalidate instruction cacheline corresponding to EA

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

This instruction is considered a “load” with respect to the above data-access exceptions. It is also considered a “load” with respect to data address-compare (DAC) debug exceptions. Debug exceptions can occur as a result of executing this instruction.

Instruction-storage exceptions and instruction TLB-miss exceptions are associated with instruction *fetching*, not with instruction *execution*. Exceptions that occur during the execution of instruction-cache operations cause data-storage exceptions and data TLB-miss exceptions.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture.

icbt

Instruction Cache Block Touch

icbt rA, rB

X Instruction Form

31	0 0 0 0 0	rA	rB	262	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

If EA is cachable but not in the instruction cache, the corresponding cacheline is loaded into the instruction cache from main memory. If EA is already cached, or if the storage attributes indicate the EA is not cachable, no operation is performed.

This instruction is a hint to the processor that the program will likely execute the instruction referenced by the EA in the near future. The processor can potentially improve performance by loading the cacheline into the instruction cache.

Pseudocode

$$EA \leftarrow (rA[0] + (rB))$$

Prefetch instruction-cacheline corresponding to EA

Registers Altered

- None.

Exceptions

This instruction is considered a “load” with respect to data-storage exceptions. However, this instruction does not cause data storage exceptions or data TLB-miss exceptions. If conditions occur that would cause these exceptions, **icbt** is treated as a no-op. This instruction is also considered a “load” with respect to data address-compare (DAC) debug exceptions. Debug exceptions can occur as a result of executing this instruction.

Instruction-storage exceptions and instruction TLB-miss exceptions are associated with instruction *fetching*, not with instruction *execution*. Exceptions that occur during the execution of instruction-cache operations cause data-storage exceptions and data TLB-miss exceptions.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC embedded-environment architecture and the PowerPC Book-E architecture. It is not defined by the PowerPC architecture, and is therefore not implemented by all PowerPC processors.

iccci

Instruction Cache Congruence Class Invalidate

iccci rA, rB

X Instruction Form

31	0 0 0 0 0	rA	rB	966	0
0	6	1	1	2	3
		1	6	1	1

Description**This is a privileged instruction.**

This instruction invalidates all lines in the instruction cache. The operands are not used. In previous implementations, the operands were used to calculate an effective address (EA) for use in protection checks. The instruction form is retained for software and tool compatibility.

This instruction is intended for use during initialization to invalidate the entire instruction cache before is enabled.

Pseudocode

Invalidate the instruction-cache

Registers Altered

- None.

Exceptions

- Program—Attempted execution of this instruction from user mode.

This instruction does not cause data-storage exceptions, data TLB-miss exceptions, or data address-compare (DAC) debug exceptions.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

icread

Instruction Cache Read

icread rA, rB

X Instruction Form

31	0 0 0 0 0	rA	rB	998	0
0	6	1	1	2	3
		1	6	1	1

Description

This is a privileged instruction.

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register rB are used as the index.
- If the rA field is 0, the base address is 0.
- If the rA field is not 0, the contents of register rA are used as the base address.

This instruction can be used as an instruction-cache debugging aid. It is used to read information for a specific instruction cacheline. The cache information is loaded into the ICDBDR.

EA_{19:26} is used to specify a congruence class within the instruction cache. CCR0[CWS] is used to select one of the two cachelines within the congruence class. If CCR0[CWS]=0 the line in way A is selected. If CCR0[CWS]=1 the line in way B is selected.

If CCR0[CIS]=0 the information read is the referenced instruction in the selected cacheline. EA_{27:29} is used as an index to select the instruction from the 32-byte line. If CCR0[CIS]=1 the information read is the tag associated with the selected cacheline.

Following execution of this instruction, ICDBDR contains the following:

Bit	Name	Function	Description
0:21	INFO	Instruction-Cache Information CCR0[CIS]=0—Instruction word. CCR0[CIS]=1—Instruction tag.	Contains either the cache-line tag or a single instruction word from the cacheline. If an instruction word is loaded it is specified using effective-address bits EA _{27:29} . CCR0[CIS] controls the type of information loaded into this field.
22:26		Reserved	

Bit	Name	Function	Description
27	V	Valid 0—Cacheline is not valid. 1—Cacheline is valid.	Contains a copy of the cache-line valid bit.
28:30		Reserved	
31	LRU	Least-Recently Used 0—Way A is least-recently used. 1—Way B is least-recently used.	Contains the LRU bit for the congruence class associated with the cacheline.

The processor does not automatically wait for the ICDBDR to be updated by an **icread** before executing a **mfsprr** that reads the ICDBDR. An **isync** instruction should be inserted between the **icread** and the **mfsprr** used to access the ICDBDR.

Pseudocode

```

EA ← (rA|0) + (rB)
if ((CCR0[CIS] = 0) ∧ (CCR0[CWS] = 0))
    then (ICDBDR) ← (instruction-cache word, way A)
if ((CCR0[CIS] = 0) ∧ (CCR0[CWS] = 1))
    then (ICDBDR) ← (instruction-cache word, way B)
if ((CCR0[CIS] = 1) ∧ (CCR0[CWS] = 0))
    then (ICDBDR) ← (instruction-cache tag, way A)
if ((CCR0[CIS] = 1) ∧ (CCR0[CWS] = 1))
    then (ICDBDR) ← (instruction-cache tag, way B)

```

Registers Altered

- ICDBDR.

Exceptions

- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.
- Program—Attempted execution of this instruction from user mode.

This instruction is considered a “load” with respect to the above data-access exceptions. It can cause data TLB-miss exceptions related to the EA even though the instruction is not address specific (multiple addresses are selected by a single EA). This instruction cannot cause data-storage exceptions. This instruction does not cause data address-compare (DAC) debug exceptions.

Instruction-storage exceptions and instruction TLB-miss exceptions are associated with instruction *fetching*, not with instruction *execution*. Exceptions that occur during the execution of instruction-cache operations cause data-storage exceptions and data TLB-miss exceptions.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

isync

Instruction Synchronize

isync

XL Instruction Form

19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150	0
0	6		2		3
			1		1

Description

The **isync** instruction is context synchronizing. It enforces ordering of all instructions executed by the processor. It ensures that all instructions preceding **isync** in program order complete before **isync** completes. Accesses to main memory caused by instructions preceding the **isync** are not guaranteed to have completed.

Instructions following the **isync** are not started until the **isync** completes execution. Prefetched instructions are discarded by the execution of **isync**. All instructions following **isync** are executed in the context established by the instructions preceding the **isync**.

isync does not affect the processor caches.

Pseudocode

Synchronize context

Registers Altered

- None.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture.

lbz

Load Byte and Zero

lbz rD, d(rA)

D Instruction Form

34	rD	rA	d
0	6	1	1
		1	6
			3
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- If the rA field is 0, the base address is 0.
- If the rA field is not 0, the contents of register rA are used as the base address.

The byte referenced by EA is extended to 32 bits by concatenating 24 0-bits on the left. The result is loaded into register rD.

Pseudocode

$$\begin{aligned} \text{EA} &\leftarrow (\text{rA}[0] + \text{EXTS}(\text{d})) \\ (\text{rD}) &\leftarrow {}^{24}0 \parallel \text{MS}(\text{EA}, 1) \end{aligned}$$

Registers Altered

- rD.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lbzu

Load Byte and Zero with Update

lbzu rD, d(rA)

D Instruction Form

35	rD	rA	d
0	6	1	1
		1	6
			3
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- The contents of register rA are used as the base address.

The byte referenced by EA is extended to 32 bits by concatenating 24 0-bits on the left. The result is loaded into register rD. The EA is loaded into rA.

Pseudocode

```
EA ← (rA) + EXTS(d)
(rD) ← 240 || MS(EA,1)
(rA) ← EA
```

Registers Altered

- rA.
- rD.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- rA=rD.
- rA=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

lbzux

Load Byte and Zero with Update Indexed

lbzux rD, rA, rB

X Instruction Form

31	rD	rA	rB	119	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- The contents of register **rA** are used as the base address.

The byte referenced by EA is extended to 32 bits by concatenating 24 0-bits on the left. The result is loaded into register **rD**. The EA is loaded into **rA**.

Pseudocode

```
EA ← (rA) + (rB)
(rD) ← 240 || MS(EA,1)
(rA) ← EA
```

Registers Altered

- **rA**.
- **rD**.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- **rA=rD**.
- **rA=0**.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lbzx

Load Byte and Zero Indexed

lbzx rD, rA, rB

X Instruction Form

31	rD	rA	rB	87	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The byte referenced by EA is extended to 32 bits by concatenating 24 0-bits on the left. The result is loaded into register **rD**.

Pseudocode

$$\begin{aligned} \text{EA} &\leftarrow (\text{rA}[0] + (\text{rB})) \\ (\text{rD}) &\leftarrow {}^{24}0 \parallel \text{MS}(\text{EA}, 1) \end{aligned}$$

Registers Altered

- **rD**.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lha

Load Halfword Algebraic

lha rD, d(rA)

D Instruction Form

42	rD	rA	d
0	6	1	3
		1	1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- If the rA field is 0, the base address is 0.
- If the rA field is not 0, the contents of register rA are used as the base address.

The halfword referenced by EA is sign-extended to 32 bits and loaded into register rD.

Pseudocode

```
EA ← (rA|0) + EXTS(d)
(rD) ← EXTS(MS(EA,2))
```

Registers Altered

- rD.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lhau

Load Halfword Algebraic with Update

lhau rD, d(rA)

D Instruction Form

43	rD	rA	d	
0	6	1	1	3
		1	6	1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- The contents of register rA are used as the base address.

The halfword referenced by EA is sign-extended to 32 bits and loaded into register rD. The EA is loaded into rA.

Pseudocode

```
EA ← (rA) + EXTS(d)
(rD) ← EXTS(MS(EA,2))
(rA) ← EA
```

Registers Altered

- rA.
- rD.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- rA=rD.
- rA=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

lhaux

Load Halfword Algebraic with Update Indexed

lhaux rD, rA, rB

X Instruction Form

31	rD	rA	rB	375	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- The contents of register **rA** are used as the base address.

The halfword referenced by EA is sign-extended to 32 bits and loaded into register **rD**. The EA is loaded into **rA**.

Pseudocode

```
EA ← (rA) + (rB)
(rD) ← EXTS(MS(EA,2))
(rA) ← EA
```

Registers Altered

- **rA**.
- **rD**.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- **rA=rD**.
- **rA=0**.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lhax

Load Halfword Algebraic Indexed

lhax rD, rA, rB

X Instruction Form

31	rD	rA	rB	343	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The halfword referenced by EA is sign-extended to 32 bits and loaded into register **rD**.

Pseudocode

```
EA ← (rA|0) + (rB)
(rD) ← EXTS(MS(EA,2))
```

Registers Altered

- **rD**.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lhbrx

Load Halfword Byte-Reverse Indexed

lhbrx rD, rA, rB

X Instruction Form

31	rD	rA	rB	790	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The memory halfword referenced by EA is byte-reversed and extended to 32 bits by concatenating 16 0-bits to its left. The result is loaded into register **rD**. The byte-reversal operation consists of:

- Bits 0:7 of the memory word are loaded into **rD**[24:31].
- Bits 8:15 of the memory word are loaded into **rD**[16:23].
- 16 0-bits are loaded into **rD**[0:15].

Pseudocode

$$\begin{aligned} \text{EA} &\leftarrow (\text{rA} \mid 0) + (\text{rB}) \\ (\text{rD}) &\leftarrow {}^{16}0 \parallel \text{MS}(\text{EA} + 1, 1) \parallel \text{MS}(\text{EA}, 1) \end{aligned}$$

Registers Altered

- **rD**.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lhz

Load Halfword and Zero

lhz rD, d(rA)

D Instruction Form

40	rD	rA	d
0	6	1	1
		1	6
			3
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- If the rA field is 0, the base address is 0.
- If the rA field is not 0, the contents of register rA are used as the base address.

The halfword referenced by EA is extended to 32 bits by concatenating 16 0-bits on the left. The result is loaded into register rD.

Pseudocode

$$\begin{aligned} \text{EA} &\leftarrow (\text{rA}[0] + \text{EXTS}(\text{d})) \\ (\text{rD}) &\leftarrow {}^{16}0 \parallel \text{MS}(\text{EA}, 2) \end{aligned}$$

Registers Altered

- rD.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lhzu

Load Halfword and Zero with Update

lhzu rD, d(rA)

D Instruction Form

41	rD	rA	d
0	6	1	3
		1	1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- The contents of register rA are used as the base address.

The halfword referenced by EA is extended to 32 bits by concatenating 16 0-bits on the left. The result is loaded into register rD. The EA is loaded into rA.

Pseudocode

```
EA ← (rA) + EXTS(d)
(rD) ← 160 || MS(EA,2)
(rA) ← EA
```

Registers Altered

- rA.
- rD.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- rA=rD.
- rA=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

lhzux

Load Halfword and Zero with Update Indexed

lhzux rD, rA, rB

X Instruction Form

31	rD	rA	rB	311	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- The contents of register **rA** are used as the base address.

The halfword referenced by EA is extended to 32 bits by concatenating 16 0-bits on the left. The result is loaded into register **rD**. The EA is loaded into **rA**.

Pseudocode

```
EA ← (rA) + (rB)
(rD) ← 160 || MS(EA,2)
(rA) ← EA
```

Registers Altered

- **rA**.
- **rD**.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- **rA=rD**.
- **rA=0**.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lhzx

Load Halfword and Zero Indexed

lhzx rD, rA, rB

X Instruction Form

31	rD	rA	rB	279	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The halfword referenced by EA is extended to 32 bits by concatenating 16 0-bits on the left. The result is loaded into register **rD**.

Pseudocode

$$\begin{aligned} \text{EA} &\leftarrow (\text{rA}[0] + (\text{rB})) \\ (\text{rD}) &\leftarrow {}^{16}0 \parallel \text{MS}(\text{EA}, 2) \end{aligned}$$

Registers Altered

- **rD**.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

Imw

Load Multiple Word

Imw rD, d(rA)

D Instruction Form

46	rD	rA	d
0	6	1	1
		1	6
			3
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- If the rA field is 0, the base address is 0.
- If the rA field is not 0, the contents of register rA are used as the base address.

Let $n = 32 - rD$.

n consecutive words starting at the memory address referenced by EA are loaded into GPRs rD through r31.

Pseudocode

```

EA ← (rA|0) + EXTS(d)
n ← rD
do while n ≤ 31
  if ((n ≠ rA) ∨ (n = 31))
    then (GPR(n)) ← MS(EA,4)
  n ← n + 1
  EA ← EA + 4

```

Registers Altered

- rD through r31.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- rA is in the range of registers to be loaded, including the case rA=rD=0. The word that would have been loaded into rA is discarded.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

lswi

Load String Word Immediate

lswi rD, rA, NB

X Instruction Form

31	rD	rA	NB	597	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is determined by the rA field as follows:

- If the rA field is 0, the EA is 0.
- If the rA field is not 0, the contents of register rA are used as the EA.

Let n specify the byte count. If the NB field is 0, n is 32. Otherwise, n is equal to NB.

Let nr specify the number of registers to load with data. $nr = \text{CEIL}(n/4)$.

Let R_{FINAL} specify the last register to be loaded with data. n consecutive bytes starting at the memory address referenced by EA are loaded into GPRs rD through R_{FINAL} . The sequence of registers wraps around to r0 if necessary. $R_{\text{FINAL}} = rD + nr - 1$ (modulo 32).

Bytes are loaded in each register starting with the most-significant register byte and ending with the least-significant register byte. If the byte count is exhausted before R_{FINAL} is filled, the remaining bytes in R_{FINAL} are loaded with 0.

Pseudocode

```

EA ← (rA|0)
if NB = 0
  then n ← 32
  else n ← NB
R_FINAL ← ((rD + CEIL(n/4) - 1) % 32)
reg ← rD - 1
bit ← 0
do while n > 0
  if bit = 0
    then
      reg ← reg + 1
      if reg = 32
        then reg ← 0
      if ((reg ≠ rA) ∨ (reg = R_FINAL))
        then (GPR(reg)) ← 0
      if ((reg ≠ rA) ∨ (reg = R_FINAL))
        then (GPR(reg)bit:bit+7) ← MS(EA,1)
      bit ← bit + 8
      if bit = 32
        then bit ← 0
      EA ← EA + 1
      n ← n - 1

```

Registers Altered

- rD and subsequent GPRs as described above.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- rA is in the range of registers to be loaded, including the case rA=rD=0. Bytes that would have been loaded into rA are discarded.
- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lswx

Load String Word Indexed

lswx rD, rA, rB

X Instruction Form

31	rD	rA	rB	533	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

Let n specify the byte count contained in **XER[TBC]**.

Let nr specify the number of registers to load with data. $nr = \text{CEIL}(n/4)$.

Let R_{FINAL} specify the last register to be loaded with data. n consecutive bytes starting at the memory address referenced by EA are loaded into GPRs **rD** through R_{FINAL} . The sequence of registers wraps around to **r0** if necessary. $R_{\text{FINAL}} = rD + nr - 1$ (modulo 32).

Bytes are loaded in each register starting with the most-significant register byte and ending with the least-significant register byte. If the byte count is exhausted before R_{FINAL} is filled, the remaining bytes in R_{FINAL} are loaded with 0.

If **XER[TBC]** = 0, the contents of register **rD** are unchanged and **lswx** is treated as a no-operation.

Pseudocode

```

EA    ← (rA|0) + (rB)
n     ← XER[TBC]
R_FINAL ← ((rD + CEIL(n/4) - 1) % 32)
reg    ← rD - 1
bit    ← 0
do while n > 0
  if bit = 0
    then
      reg ← reg + 1
      if reg = 32
        then reg ← 0
      if ((reg ≠ rA) ∨ (reg = R_FINAL))
        then (GPR(reg)) ← 0
      if ((reg ≠ rA) ∨ (reg = R_FINAL))
        then (GPR(reg)bit:bit+7) ← MS(EA,1)
      bit ← bit + 8

```

```
if bit = 32
  then bit ← 0
EA ← EA + 1
n ← n - 1
```

Registers Altered

- rD and subsequent GPRs as described above.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

If XER[TBC]=0, data-storage and data TLB-miss exceptions do not occur. However, a data machine-check exception can occur when XER[TBC]=0 if the following conditions are true:

- The instruction access passes all protection checks.
- The data address is cachable.
- Access of the data address causes a data-cacheline fill request due to a miss.
- The data-cacheline fill request encounters some form of bus error.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- rA is in the range of registers to be loaded, including the case rA=rD=0. Bytes that would have been loaded into rA are discarded.
- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lwarx

Load Word and Reserve Indexed

lwarx rD, rA, rB

X Instruction Form

31	rD	rA	rB	20	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The word referenced by EA is loaded into register **rD**. A reservation bit internal to the processor is set.

The **lwarx** and the **stwcx**. instructions should be paired in a loop to create the effect of an atomic memory operation for accessing a semaphore. See **Semaphore Synchronization**, [page 462](#) for more information.

Pseudocode

```
EA      ← (rA|0) + (rB)
(rD)    ← MS(EA,4)
RESERVE← 1
```

Registers Altered

- **rD**.

Exceptions

- Alignment—if the EA is not aligned on a word boundary.
- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lwbrx

Load Word Byte-Reverse Indexed

lwbrx rD, rA, rB

X Instruction Form

31	rD	rA	rB	534	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The memory word referenced by EA is byte-reversed and the result is loaded into register **rD**. The byte-reversal operation consists of:

- Bits 0:7 of the memory word are loaded into **rD**[24:31].
- Bits 8:15 of the memory word are loaded into **rD**[16:23].
- Bits 16:23 of the memory word are loaded into **rD**[8:15].
- Bits 23:31 of the memory word are loaded into **rD**[0:7].

Pseudocode

$$\begin{aligned} \text{EA} &\leftarrow (\text{rA}[0] + (\text{rB})) \\ (\text{rD}) &\leftarrow \text{MS}(\text{EA}+3,1) \parallel \text{MS}(\text{EA}+2,1) \parallel \text{MS}(\text{EA}+1,1) \parallel \text{MS}(\text{EA},1) \end{aligned}$$

Registers Altered

- **rD**.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lwz

Load Word and Zero

lwz rD, d(rA)

D Instruction Form

32	rD	rA	d
0	6	1	1
		1	6
			3
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- If the rA field is 0, the base address is 0.
- If the rA field is not 0, the contents of register rA are used as the base address.

The word referenced by EA is loaded into register rD.

Pseudocode

```
EA ← (rA|0) + EXTS(d)
(rD) ← MS(EA,4)
```

Registers Altered

- rD.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

lwzu

Load Word and Zero with Update

lwzu rD, d(rA)

D Instruction Form

33	rD	rA	d
0	6	1	1
		1	6
			3
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- The contents of register rA are used as the base address.

The word referenced by EA is loaded into register rD. The EA is loaded into rA.

Pseudocode

```

EA ← (rA) + EXTS(d)
(rD) ← MS(EA,4)
(rA) ← EA

```

Registers Altered

- rA.
- rD.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- rA=rD.
- rA=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

lwzux

Load Word and Zero with Update Indexed

lwzux rD, rA, rB

X Instruction Form

31	rD	rA	rB	55	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- The contents of register **rA** are used as the base address.

The word referenced by EA is loaded into register **rD**. The EA is loaded into **rA**.

Pseudocode

```
EA ← (rA) + (rB)
(rD) ← MS(EA,4)
(rA) ← EA
```

Registers Altered

- **rA**.
- **rD**.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- **rA=rD**.
- **rA=0**.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

lwzx

Load Word and Zero Indexed

lwzx rD, rA, rB

X Instruction Form

31	rD	rA	rB	23	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The word referenced by EA is loaded into register **rD**.

Pseudocode

```
EA ← (rA|0) + (rB)
(rD) ← MS(EA,4)
```

Registers Altered

- **rD**.

Exceptions

- Data storage—if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

macchw

Multiply Accumulate Cross Halfword to Word Modulo Signed

macchw	rD, rA, rB	(OE=0, Rc=0)
macchw.	rD, rA, rB	(OE=0, Rc=1)
macchw0	rD, rA, rB	(OE=1, Rc=0)
macchw0.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	172	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of **rA** is multiplied by the high-order halfword of **rB**. The signed product is added to the contents of **rD** and the sum is stored as a 33-bit temporary result. The contents of **rD** are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in [Figure 3-28, page 444](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)0:15 signed
temp0:32 ← prod0:31 + (rD)
(rD)      ← temp1:32

```

Registers Altered

- **rD**.
- **CR[CR0]_{LT, GT, EQ, SO}** if **Rc=1**.
- **XER[SO, OV]** if **OE=1**.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

macchws

Multiply Accumulate Cross Halfword to Word Saturate Signed

macchws	rD, rA, rB	(OE=0, Rc=0)
macchws.	rD, rA, rB	(OE=0, Rc=1)
macchwso	rD, rA, rB	(OE=1, Rc=0)
macchwso.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	236	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of **rA** is multiplied by the high-order halfword of **rB**. The signed product is added to the contents of **rD** and the sum is stored as a 33-bit temporary result.

If the result does not overflow, the low-order 32 bits of the temporary result are stored in **rD**. If the result overflows, **rD** is loaded with the nearest representable value. If the result is less than -2^{31} , the value stored in **rD** is -2^{31} . If the result is greater than $2^{31} - 1$, the value stored in **rD** is $2^{31} - 1$. An example of this operation is shown in [Figure 3-28, page 444](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)0:15 signed
temp0:32 ← prod0:31 + (rD)
if ((prod0 = rD0) ∧ (rD0 ≠ temp1))
then (rD) ← (rD0 || 31(¬rD0))
else (rD) ← temp1:32

```

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

macchwsu

Multiply Accumulate Cross Halfword to Word Saturate Unsigned

macchwsu	rD, rA, rB	(OE=0, Rc=0)
macchwsu.	rD, rA, rB	(OE=0, Rc=1)
macchwsuo	rD, rA, rB	(OE=1, Rc=0)
macchwsuo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	204	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of rA is multiplied by the high-order halfword of rB. The unsigned product is added to the contents of rD and the sum is stored as a 33-bit temporary result.

If the result does not overflow, the low-order 32 bits of the temporary result are stored in rD. If the result overflows, rD is loaded with the nearest representable value. If the result is greater than $2^{32} - 1$, the value stored in rD is $2^{32} - 1$. An example of this operation is shown in [Figure 3-28, page 444](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)0:15 unsigned
temp0:32 ← prod0:31 + (rD)
(rD)      ← (temp1:32 ∨ 32temp0)

```

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

macchwu

Multiply Accumulate Cross Halfword to Word Modulo Unsigned

macchwu	rD, rA, rB	(OE=0, Rc=0)
macchwu.	rD, rA, rB	(OE=0, Rc=1)
macchwuo	rD, rA, rB	(OE=1, Rc=0)
macchwuo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	140	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of rA is multiplied by the high-order halfword of rB. The unsigned product is added to the contents of rD and the sum is stored as a 33-bit temporary result. The contents of rD are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in [Figure 3-28, page 444](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)0:15 unsigned
temp0:32 ← prod0:31 + (rD)
(rD)      ← temp1:32

```

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

machhw

Multiply Accumulate High Halfword to Word Modulo Signed

machhw	rD, rA, rB	(OE=0, Rc=0)
machhw.	rD, rA, rB	(OE=0, Rc=1)
machhwo	rD, rA, rB	(OE=1, Rc=0)
machhwo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	44	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The high-order halfword of rA is multiplied by the high-order halfword of rB. The signed product is added to the contents of rD and the sum is stored as a 33-bit temporary result. The contents of rD are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in [Figure 3-29, page 446](#).

Pseudocode

```

prod0:31 ← (rA)0:15 × (rB)0:15 signed
temp0:32 ← prod0:31 + (rD)
(rD)      ← temp1:32

```

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

machhws

Multiply Accumulate High Halfword to Word Saturate Signed

machhws	rD, rA, rB	(OE=0, Rc=0)
machhws.	rD, rA, rB	(OE=0, Rc=1)
machhwso	rD, rA, rB	(OE=1, Rc=0)
machhwso.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	108	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The high-order halfword of rA is multiplied by the high-order halfword of rB. The signed product is added to the contents of rD and the sum is stored as a 33-bit temporary result.

If the result does not overflow, the low-order 32 bits of the temporary result are stored in rD. If the result overflows, rD is loaded with the nearest representable value. If the result is less than -2^{31} , the value stored in rD is -2^{31} . If the result is greater than $2^{31} - 1$, the value stored in rD is $2^{31} - 1$. An example of this operation is shown in [Figure 3-29, page 446](#).

Pseudocode

```

prod0:31 ← (rA)0:15 × (rB)0:15 signed
temp0:32 ← prod0:31 + (rD)
if ((prod0 = rD0) ∧ (rD0 ≠ temp1))
  then (rD) ← (rD0 || 31(¬rD0))
  else (rD) ← temp1:32

```

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

machwsu

Multiply Accumulate High Halfword to Word Saturate Unsigned

machwsu	rD, rA, rB	(OE=0, Rc=0)
machwsu.	rD, rA, rB	(OE=0, Rc=1)
machwsuo	rD, rA, rB	(OE=1, Rc=0)
machwsuo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	76	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The high-order halfword of rA is multiplied by the high-order halfword of rB. The unsigned product is added to the contents of rD and the sum is stored as a 33-bit temporary result.

If the result does not overflow, the low-order 32 bits of the temporary result are stored in rD. If the result overflows, rD is loaded with the nearest representable value. If the result is greater than $2^{32} - 1$, the value stored in rD is $2^{32} - 1$. An example of this operation is shown in [Figure 3-29, page 446](#).

Pseudocode

```

prod0:31 ← (rA)0:15 × (rB)0:15 unsigned
temp0:32 ← prod0:31 + (rD)
(rD)      ← (temp1:32 ∨ 32temp0)

```

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

machhwu

Multiply Accumulate High Halfword to Word Modulo Unsigned

machhwu	rD, rA, rB	(OE=0, Rc=0)
machhwu.	rD, rA, rB	(OE=0, Rc=1)
machhwuo	rD, rA, rB	(OE=1, Rc=0)
machhwuo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	12	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The high-order halfword of **rA** is multiplied by the high-order halfword of **rB**. The unsigned product is added to the contents of **rD** and the sum is stored as a 33-bit temporary result. The contents of **rD** are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in [Figure 3-29, page 446](#).

Pseudocode

```

prod0:31 ← (rA)0:15 × (rB)0:15 unsigned
temp0:32 ← prod0:31 + (rD)
(rD)      ← temp1:32

```

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

macldhw

Multiply Accumulate Low Halfword to Word Modulo Signed

macldhw	rD, rA, rB	(OE=0, Rc=0)
macldhw.	rD, rA, rB	(OE=0, Rc=1)
macldhwo	rD, rA, rB	(OE=1, Rc=0)
macldhwo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	428	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of **rA** is multiplied by the low-order halfword of **rB**. The signed product is added to the contents of **rD** and the sum is stored as a 33-bit temporary result. The contents of **rD** are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in [Figure 3-30, page 449](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)16:31 signed
temp0:32 ← prod0:31 + (rD)
(rD)      ← temp1:32

```

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

maclhws

Multiply Accumulate Low Halfword to Word Saturate Signed

maclhws	rD, rA, rB	(OE=0, Rc=0)
maclhws.	rD, rA, rB	(OE=0, Rc=1)
maclhwsO	rD, rA, rB	(OE=1, Rc=0)
maclhwsO.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	492	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of **rA** is multiplied by the low-order halfword of **rB**. The signed product is added to the contents of **rD** and the sum is stored as a 33-bit temporary result.

If the result does not overflow, the low-order 32 bits of the temporary result are stored in **rD**. If the result overflows, **rD** is loaded with the nearest representable value. If the result is less than -2^{31} , the value stored in **rD** is -2^{31} . If the result is greater than $2^{31} - 1$, the value stored in **rD** is $2^{31} - 1$. An example of this operation is shown in [Figure 3-30, page 449](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)16:31 signed
temp0:32 ← prod0:31 + (rD)
if ((prod0 = rD0) ∧ (rD0 ≠ temp1))
  then (rD) ← (rD0 || 31(¬rD0))
  else (rD) ← temp1:32

```

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

maclhwsu

Multiply Accumulate Low Halfword to Word Saturate Unsigned

maclhwsu	rD, rA, rB	(OE=0, Rc=0)
maclhwsu.	rD, rA, rB	(OE=0, Rc=1)
maclhwsuo	rD, rA, rB	(OE=1, Rc=0)
maclhwsuo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	460	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of rA is multiplied by the low-order halfword of rB. The unsigned product is added to the contents of rD and the sum is stored as a 33-bit temporary result.

If the result does not overflow, the low-order 32 bits of the temporary result are stored in rD. If the result overflows, rD is loaded with the nearest representable value. If the result is greater than $2^{32} - 1$, the value stored in rD is $2^{32} - 1$. An example of this operation is shown in [Figure 3-30, page 449](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)16:31 unsigned
temp0:32 ← prod0:31 + (rD)
(rD) ← (temp1:32 ∨ 32temp0)

```

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

macldwu

Multiply Accumulate Low Halfword to Word Modulo Unsigned

macldwu	rD, rA, rB	(OE=0, Rc=0)
macldwu.	rD, rA, rB	(OE=0, Rc=1)
macldwuo	rD, rA, rB	(OE=1, Rc=0)
macldwuo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	396	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of rA is multiplied by the low-order halfword of rB. The unsigned product is added to the contents of rD and the sum is stored as a 33-bit temporary result. The contents of rD are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in [Figure 3-30, page 449](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)16:31 unsigned
temp0:32 ← prod0:31 + (rD)
(rD)      ← temp1:32

```

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

mcrf

Move Condition Register Field

mcrf crfD, crfS

XL Instruction Form

19	crfD	0 0	crfS	0 0 0 0 0 0 0	0	0
0	6	9	1	1	2	3
			1	4	1	1

Description

The contents of the CR field specified by **crfS** are loaded into the CR field specified by **crfD**.

Pseudocode

```

m ← crfS
n ← crfD
(CR[CRn]) ← (CR[CRm])

```

Registers Altered

- CR[CRn] where *n* is specified by **crfD**.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

mcrxr

Move to Condition Register from XER

mcrxr crfD

X Instruction Form

31	crfD	0 0 0 0 0 0 0 0 0 0 0 0 0 0	512	0
0	6	9	2	3
			1	1

Description

The contents of $XER_{0:3}$ are loaded into the CR field specified by **crfD**. The contents of $XER_{0:3}$ are then cleared to 0.

Pseudocode

```

n      ← crfD
(CR[CRn]) ← XER0:3
XER0:3  ← 0b0000

```

Registers Altered

- CR[CRn] where *n* is specified by the **crfD** field.
- $XER_{0:3}$.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

mfcrr

Move from Condition Register

mfcrr rD

X Instruction Form

31	rD	0 0 0 0 0 0 0 0 0 0 0 0	19	0
0	6	1	2	3
		1	1	1

Description

The contents of the CR are loaded into register rD.

Pseudocode

(rD) ← (CR)

Registers Altered

- rD.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

mfdcr

Move from Device Control Register

mfdcr rD, DCRN

XFX Instruction Form

31	rD	DCRF	323	0
0	6	1	2	3
		1	1	1

Description

This is a privileged instruction.

The contents of the DCR specified by the DCR number (DCRN) are loaded into register rD. The DCRF opcode field is a split field representing DCRN. See **Split-Field Notation**, [page 607](#) for more information.

Pseudocode

$$\begin{aligned} \text{DCRN} &\leftarrow \text{DCRF}_{5:9} \parallel \text{DCRF}_{0:4} \\ (\text{rD}) &\leftarrow (\text{DCR}(\text{DCRN})) \end{aligned}$$

Registers Altered

- rD.

Exceptions

- Program—Attempted execution of this instruction from user mode.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- Use of an unsupported DCRF value.

Compatibility

This instruction is defined by the PowerPC embedded-environment architecture and the PowerPC Book-E architecture. It is not defined by the PowerPC architecture, and is therefore not implemented by all PowerPC processors. The specific registers accessed by this instruction are implementation dependent.

mfmsr

Move from Machine State Register

mfmsr rD

X Instruction Form

31	rD	0 0 0 0 0 0 0 0 0 0 0 0	83	0
0	6	1	2	3
		1	1	1

Description

This is a privileged instruction.
 The contents of the MSR are loaded into register rD.

Pseudocode

(rD) ← (MSR)

Registers Altered

- rD.

Exceptions

- Program—Attempted execution of this instruction from user mode.
 Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:
- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the operating-environment architecture level (OEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture. It is implemented by all PowerPC processors.

mfspir

Move from Special Purpose Register

mfspir rD, SPRN

XFX Instruction Form

31	rD	SPRF	339	0
0	6	1	2	3
		1	1	1

Description

The contents of the SPR specified by the SPR number (SPRN) are loaded into register rD. The SPRF opcode field is a split field representing SPRN. See **Split-Field Notation**, [page 607](#) for more information. See **Appendix A, Register Summary** for a listing of the SPRs supported by the PPC405 and their corresponding SPRN and SPRF values.

Simplified mnemonics defined for this instruction are described in **Special-Purpose Registers**, [page 866](#).

Pseudocode

$$\begin{aligned} \text{SPRN} &\leftarrow \text{SPRF}_{5:9} \parallel \text{SPRF}_{0:4} \\ (\text{rD}) &\leftarrow (\text{SPR}(\text{SPRN})) \end{aligned}$$

Registers Altered

- rD.

Exceptions

- Program—Attempted execution of this instruction from user mode if SPRF[0] (bit 11 of the instruction opcode) is 1.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- Use of an unsupported SPRF value.

Compatibility

This instruction is defined by the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture. It is part of the user instruction-set architecture (UISA) and the operating-environment architecture (OEA). It is implemented by all PowerPC processors. However, not all SPRs supported by the PPC405 are supported by other PowerPC processors.

mftb

Move from Time Base

mftb rD, TBRN

XFX Instruction Form

31	rD	TBRF	371	0
0	6	1	2	3
		1	1	1

Description

The contents of the TBR specified by the TBR number (TBRN) are loaded into register rD. The TBRF opcode field is a split field representing TBRN. See **Split-Field Notation**, [page 607](#) for more information. The following TBRN values are recognized:

- Time-base lower register (TBL)—268 (0x10C).
- Time-base upper register (TBU)—269 (0x10D).

Simplified mnemonics defined for this instruction are described in **Special-Purpose Registers**, [page 866](#).

Pseudocode

```
TBRN ← TBRF5:9 || TBRF0:4
(rD) ← (TBR(TBRN))
```

Registers Altered

- rD.

Exceptions

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- Use of an unsupported TBRF value.

Compatibility

This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture and the PowerPC embedded-environment architecture. The PowerPC Book-E architecture does not support this instruction, but does support the time-base registers. Software running on PowerPC Book-E processors must use the **mfspr** instruction to access the time-base registers.

mtcrf

Move to Condition Register Fields

mtcrf CRM, rS

XFX Instruction Form

31	rS	0	CRM	0	144	0
0	6	1 1		2 2		3
		1 2		0 1		1

Description

Some or all of the contents of register **rS** are loaded into the CR under the control of the CRM field.

Each bit in the CRM field specifies a set of 4 bits in both the **rS** and CR registers. If a CRM bit is set to 1, the specified set of bits in **rS** are copied into the corresponding CR bits. If a CRM bit is cleared to 0, the specified set of bits in **rS** are not copied and the corresponding CR bits are unchanged. The following table shows the relationship between the CRM field and the **rS** and CR registers. The **CR_n** field is shown for completeness.

CRM Bit Number	rS Bits	CR Bits	CR _n Field
0	0:3	0:3	CR0
1	4:7	4:7	CR1
2	8:11	8:11	CR2
3	12:15	12:15	CR3
4	16:19	16:19	CR4
5	20:23	20:23	CR5
6	24:27	24:27	CR6
7	28:31	28:31	CR7

See **mtcrf Field Mask (CRM)**, [page 459](#), for more information on the CRM field and an example of its use.

Simplified mnemonics defined for this instruction are described in **Other Simplified Mnemonics**, [page 870](#).

Pseudocode

$$\text{mask} \leftarrow {}^4(\text{CRM}_0) \parallel {}^4(\text{CRM}_1) \parallel \dots \parallel {}^4(\text{CRM}_6) \parallel {}^4(\text{CRM}_7)$$

$$(\text{CR}) \leftarrow ((\text{rS}) \wedge \text{mask}) \vee ((\text{CR}) \wedge \neg \text{mask})$$

Registers Altered

- CR.

Exceptions

- None.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

mtdcr

Move to Device Control Register

mtdcr DCRN, rS

XFX Instruction Form

31	rS	DCRF	451	0
0	6	1	2	3
		1	1	1

Description

This is a privileged instruction.

The contents of register rS are loaded into the DCR specified by the DCR number (DCRN). The DCRF opcode field is a split field representing DCRN. See **Split-Field Notation**, [page 607](#) for more information.

Pseudocode

$$\begin{aligned} \text{DCRN} &\leftarrow \text{DCRF}_{5:9} \parallel \text{DCRF}_{0:4} \\ (\text{DCR}(\text{DCRN})) &\leftarrow (\text{rS}) \end{aligned}$$

Registers Altered

- DCR(DCRN).

Exceptions

- Program—Attempted execution of this instruction from user mode.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- Use of an unsupported DCRF value.

Compatibility

This instruction is defined by the PowerPC embedded-environment architecture and the PowerPC Book-E architecture. It is not defined by the PowerPC architecture, and is therefore not implemented by all PowerPC processors. The specific registers accessed by this instruction are implementation dependent.

mtmsr

Move to Machine State Register

mtmsr rS

X Instruction Form

31	rS	0 0 0 0 0 0 0 0 0 0 0	146	0
0	6	1	2	3
		1	1	1

Description

This is a privileged instruction.
 The contents of register rS are loaded into the MSR.

Pseudocode

(MSR) ← (rS)

Registers Altered

- MSR.

Exceptions

- Program—Attempted execution of this instruction from user mode.
 Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:
- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the operating-environment architecture level (OEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture. It is implemented by all PowerPC processors.

mtspr

Move to Special Purpose Register

mtspr SPRN, rS

XFX Instruction Form

31	rS	SPRF	467	0
0	6	1	2	3
		1	1	1

Description

The contents of register **rS** are loaded into the SPR specified by the SPR number (SPRN). The SPRF opcode field is a split field representing SPRN. See **Split-Field Notation**, [page 607](#) for more information. See **Appendix A, Register Summary** for a listing of the SPRs supported by the PPC405 and their corresponding SPRN and SPRF values.

Simplified mnemonics defined for this instruction are described in **Special-Purpose Registers**, [page 866](#).

Pseudocode

$$\begin{aligned} \text{SPRN} &\leftarrow \text{SPRF}_{5:9} \parallel \text{SPRF}_{0:4} \\ (\text{SPR}(\text{SPRN})) &\leftarrow (\text{rS}) \end{aligned}$$

Registers Altered

- SPR(SPRN).

Exceptions

- Program—Attempted execution of this instruction from user mode if SPRF[0] (bit 11 of the instruction) is 1.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- Use of an unsupported SPRF value.

Compatibility

This instruction is defined by the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture. It is part of the user instruction-set architecture (UISA) and the operating-environment architecture (OEA). It is implemented by all PowerPC processors. However, not all SPRs supported by the PPC405 are supported by other PowerPC processors.

mulchw

Multiply Cross Halfword to Word Signed

mulchw rD, rA, rB (Rc=0)

mulchw. rD, rA, rB (Rc=1)

X Instruction Form

4	rD	rA	rB	168	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The low-order halfword of rA is multiplied by the high-order halfword of rB. The resulting signed 32-bit product is loaded into register rD. An example of this operation is shown in [Figure 3-34, page 456](#).

Pseudocode

$$(rD)_{0:31} \leftarrow (rA)_{16:31} \times (rB)_{0:15} \text{ signed}$$

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

mulchwu

Multiply Cross Halfword to Word Unsigned

mulchwu rD, rA, rB (Rc=0)

mulchwu. rD, rA, rB (Rc=1)

X Instruction Form

4	rD	rA	rB	136	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The low-order halfword of rA is multiplied by the high-order halfword of rB. The resulting unsigned 32-bit product is loaded into register rD. An example of this operation is shown in [Figure 3-34, page 456](#).

Pseudocode

$$(rD)_{0:31} \leftarrow (rA)_{16:31} \times (rB)_{0:15} \text{ unsigned}$$

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

mulhww

Multiply High Halfword to Word Signed

mulhww	rD, rA, rB	(Rc=0)
mulhww.	rD, rA, rB	(Rc=1)

X Instruction Form

4	rD	rA	rB	40	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The high-order halfword of rA is multiplied by the high-order halfword of rB. The resulting signed 32-bit product is loaded into register rD. An example of this operation is shown in [Figure 3-35, page 457](#).

Pseudocode

$$(rD)_{0:31} \leftarrow (rA)_{0:15} \times (rB)_{0:15} \text{ signed}$$

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

mulhhwu

Multiply High Halfword to Word Unsigned

mulhhwu rD, rA, rB (Rc=0)
mulhhwu. rD, rA, rB (Rc=1)

X Instruction Form

4	rD	rA	rB	8	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The high-order halfword of rA is multiplied by the high-order halfword of rB. The resulting unsigned 32-bit product is loaded into register rD. An example of this operation is shown in [Figure 3-35, page 457](#).

Pseudocode

$$(rD)_{0:31} \leftarrow (rA)_{0:15} \times (rB)_{0:15} \text{ unsigned}$$

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

mulhw

Multiply High Word

mulhw rD, rA, rB (Rc=0)
mulhw. rD, rA, rB (Rc=1)

XO Instruction Form

31	rD	rA	rB	0	75	Rc
0	6	1	1	2	2	3
		1	6	1	2	1

Description

The contents of register **rA** are multiplied with the contents of register **rB**, forming a 64-bit signed product. The most-significant 32 bits of the result are loaded into register **rD**.

mulhwu should be used if the operands are to be interpreted as unsigned quantities.

This instruction can be used with **mullw** or **mulli** to calculate a full 64-bit product.

Pseudocode

$$\begin{aligned} \text{prod}_{0:63} &\leftarrow (\text{rA}) \times (\text{rB}) \text{ signed} \\ (\text{rD}) &\leftarrow \text{prod}_{0:31} \end{aligned}$$

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

mulhwu

Multiply High Word Unsigned

mulhwu rD, rA, rB (Rc=0)
mulhwu. rD, rA, rB (Rc=1)

XO Instruction Form

31	rD	rA	rB	0	11	Rc
0	6	1	1	2		3
		1	6	1		1

Description

The contents of register **rA** are multiplied with the contents of register **rB**, forming a 64-bit unsigned product. The most-significant 32 bits of the result are loaded into register **rD**.

mulhw should be used if the operands are to be interpreted as signed quantities.

Pseudocode

```
prod0:63 ← (rA) × (rB) unsigned
(rD)      ← prod0:31
```

Registers Altered

- **rD**.
- CR[CR0]_{LT,GT,EQ,SO} if Rc=1.

Exceptions

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

mullhw

Multiply Low Halfword to Word Signed

mullhw rD, rA, rB (Rc=0)
mullhw. rD, rA, rB (Rc=1)

X Instruction Form

4	rD	rA	rB	424	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The low-order halfword of rA is multiplied by the low-order halfword of rB. The resulting signed 32-bit product is loaded into register rD. An example of this operation is shown in [Figure 3-36, page 458](#).

Pseudocode

$$(rD)_{0:31} \leftarrow (rA)_{16:31} \times (rB)_{16:31} \text{ signed}$$

Registers Altered

- rD.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

mullhww

Multiply Low Halfword to Word Unsigned

mullhww rD, rA, rB (OE=0, Rc=0)

mullhww. rD, rA, rB (OE=0, Rc=1)

X Instruction Form

4	rD	rA	rB	392	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The low-order halfword of **rA** is multiplied by the low-order halfword of **rB**. The resulting unsigned 32-bit product is loaded into register **rD**. An example of this operation is shown in [Figure 3-36, page 458](#).

Pseudocode

$$(rD)_{0:31} \leftarrow (rA)_{16:31} \times (rB)_{16:31} \text{ unsigned}$$

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

mulli

Multiply Low Immediate

mulli rD, rA, SIMM

D Instruction Form

7	rD	rA	SIMM
0	6	1	1
		1	6
			3
			1

Description

The contents of register **rA** are multiplied with the sign-extended SIMM field, forming a 48-bit signed product. The least-significant 32 bits of the product are loaded into register **rD**.

The result loaded into register **rD** is always correct, regardless of whether the operands are interpreted as signed or unsigned integers.

This instruction can be used with **mulhw** to calculate a full 64-bit product.

Pseudocode

$$\begin{aligned} \text{prod}_{0:47} &\leftarrow (\text{rA}) \times \text{EXTS}(\text{SIMM}) \text{ signed} \\ (\text{rD}) &\leftarrow \text{prod}_{16:47} \end{aligned}$$

Registers Altered

- **rD**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

mullw

Multiply Low Word

mullw	rD, rA, rB	(OE=0, Rc=0)
mullw.	rD, rA, rB	(OE=0, Rc=1)
mullwo	rD, rA, rB	(OE=1, Rc=0)
mullwo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	235	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The contents of register **rA** are multiplied with the contents of register **rB**, forming a 64-bit signed product. The least-significant 32 bits of the result are loaded into register **rD**.

If the signed product cannot be represented in 32 bits and OE=1, XER[SO, OV] are set to 1. This overflow indication is correct only if the operands are interpreted as signed integers. The result loaded into register **rD** is always correct, regardless of whether the operands are interpreted as signed or unsigned integers.

This instruction can be used with **mulhw** to calculate a full 64-bit product.

Pseudocode

```

prod0:63 ← (rA) × (rB) signed
(rD)      ← prod32:63

```

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

nand

NAND

nand	rA, rS, rB	(Rc=0)
nand.	rA, rS, rB	(Rc=1)

X Instruction Form

31	rS	rA	rB	476	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The contents of register **rS** are ANDed with the contents of register **rB** and the one's complement of the result is loaded into register **rA**.

The one's complement of a number can be obtained using **nand** with **rS = rB**.

Pseudocode

$$(rA) \leftarrow \neg((rS) \wedge (rB))$$

Registers Altered

- **rA**.
- **CR[CR0]_{LT,GT,EQ,SO}** if **Rc=1**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

neg

Negate

neg	rD, rA	(OE=0, Rc=0)
neg.	rD, rA	(OE=0, Rc=1)
nego	rD, rA	(OE=1, Rc=0)
nego.	rD, rA	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	0	0	0	0	0	OE	104	Rc
0	6	1	1					2 2		3
		1	6					1 2		1

Description

The two's complement of the contents of register **rA** are loaded into register **rD**.

If **rA** contains the most-negative number (0x8000_0000), the result is the most-negative number and XER[OV] is set to 1 if OE=1.

Pseudocode

$$(rD) \leftarrow \neg(rA) + 1$$

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

nmacchw

Negative Multiply Accumulate Cross Halfword to Word Modulo Signed

nmacchw	rD, rA, rB	(OE=0, Rc=0)
nmacchw.	rD, rA, rB	(OE=0, Rc=1)
nmacchw0	rD, rA, rB	(OE=1, Rc=0)
nmacchw0.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	174	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of **rA** is multiplied by the high-order halfword of **rB**. The negated signed product is added to the contents of **rD** and the sum is stored as a 33-bit temporary result. The contents of **rD** are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in [Figure 3-31, page 451](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)0:15 signed
nprod0:31 ← -1 × prod0:31 signed
temp0:32 ← nprod0:31 + (rD)
(rD) ← temp1:32

```

Registers Altered

- **rD**.
- **CR[CR0]_{LT,GT,EQ,SO}** if **Rc=1**.
- **XER[SO,OV]** if **OE=1**

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

nmacchws

Negative Multiply Accumulate Cross Halfword to Word Saturate Signed

nmacchws	rD, rA, rB	(OE=0, Rc=0)
nmacchws.	rD, rA, rB	(OE=0, Rc=1)
nmacchwso	rD, rA, rB	(OE=1, Rc=0)
nmacchwso.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	238	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of **rA** is multiplied by the high-order halfword of **rB**. The negated signed product is added to the contents of **rD** and the sum is stored as a 33-bit temporary result.

If the result does not overflow, the low-order 32 bits of the temporary result are stored in **rD**. If the result overflows, **rD** is loaded with the nearest representable value. If the result is less than -2^{31} , the value stored in **rD** is -2^{31} . If the result is greater than $2^{31} - 1$, the value stored in **rD** is $2^{31} - 1$. An example of this operation is shown in [Figure 3-31, page 451](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)0:15 signed
nprod0:31 ← -1 × prod0:31 signed
temp0:32 ← nprod0:31 + (rD)
if ((nprod0 = rD0) ∧ (rD0 ≠ temp1))
then (rD) ← (rD0 || 31(-rD0))
else (rD) ← temp1:32

```

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

nmachhw

Negative Multiply Accumulate High Halfword to Word Modulo Signed

nmachhw	rD, rA, rB	(OE=0, Rc=0)
nmachhw.	rD, rA, rB	(OE=0, Rc=1)
nmachhwo	rD, rA, rB	(OE=1, Rc=0)
nmachhwo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	46	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The high-order halfword of **rA** is multiplied by the high-order halfword of **rB**. The negated signed product is added to the contents of **rD** and the sum is stored as a 33-bit temporary result. The contents of **rD** are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in [Figure 3-32, page 453](#).

Pseudocode

```

prod0:31 ← (rA)0:15 × (rB)0:15 signed
nprod0:31 ← -1 × prod0:31 signed
temp0:32 ← nprod0:31 + (rD)
(rD) ← temp1:32

```

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

nmachhws

Negative Multiply Accumulate High Halfword to Word Saturate Signed

nmachhws	rD, rA, rB	(OE=0, Rc=0)
nmachhws.	rD, rA, rB	(OE=0, Rc=1)
nmachhwsO	rD, rA, rB	(OE=1, Rc=0)
nmachhwsO.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	110	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The high-order halfword of **rA** is multiplied by the high-order halfword of **rB**. The negated signed product is added to the contents of **rD** and the sum is stored as a 33-bit temporary result.

If the result does not overflow, the low-order 32 bits of the temporary result are stored in **rD**. If the result overflows, **rD** is loaded with the nearest representable value. If the result is less than -2^{31} , the value stored in **rD** is -2^{31} . If the result is greater than $2^{31} - 1$, the value stored in **rD** is $2^{31} - 1$. An example of this operation is shown in [Figure 3-32, page 453](#).

Pseudocode

```

prod0:31 ← (rA)0:15 × (rB)0:15 signed
nprod0:31 ← -1 × prod0:31 signed
temp0:32 ← nprod0:31 + (rD)
if ((nprod0 = rD0) ∧ (rD0 ≠ temp1))
  then (rD) ← (rD0 || 31(-rD0))
  else (rD) ← temp1:32

```

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

nmacldw

Negative Multiply Accumulate Low Halfword to Word Modulo Signed

nmacldw	rD, rA, rB	(OE=0, Rc=0)
nmacldw.	rD, rA, rB	(OE=0, Rc=1)
nmacldwo	rD, rA, rB	(OE=1, Rc=0)
nmacldwo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	430	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of **rA** is multiplied by the low-order halfword of **rB**. The negated signed product is added to the contents of **rD** and the sum is stored as a 33-bit temporary result. The contents of **rD** are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in [Figure 3-33, page 455](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)16:31 signed
nprod0:31 ← -1 × prod0:31 signed
temp0:32 ← nprod0:31 + (rD)
(rD) ← temp1:32

```

Registers Altered

- **rD**.
- **CR[CR0]_{LT,GT,EQ,SO}** if **Rc=1**.
- **XER[SO,OV]** if **OE=1**

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

nmaclhws

Negative Multiply Accumulate Low Halfword to Word Saturate Signed

nmaclhws	rD, rA, rB	(OE=0, Rc=0)
nmaclhws.	rD, rA, rB	(OE=0, Rc=1)
nmaclhwsO	rD, rA, rB	(OE=1, Rc=0)
nmaclhwsO.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	494	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The low-order halfword of **rA** is multiplied by the low-order halfword of **rB**. The negated signed product is added to the contents of **rD** and the sum is stored as a 33-bit temporary result.

If the result does not overflow, the low-order 32 bits of the temporary result are stored in **rD**. If the result overflows, **rD** is loaded with the nearest representable value. If the result is less than -2^{31} , the value stored in **rD** is -2^{31} . If the result is greater than $2^{31} - 1$, the value stored in **rD** is $2^{31} - 1$. An example of this operation is shown in [Figure 3-33, page 455](#).

Pseudocode

```

prod0:31 ← (rA)16:31 × (rB)16:31 signed
nprod0:31 ← -1 × prod0:31 signed
temp0:32 ← nprod0:31 + (rD)
if ((nprod0 = rD0) ∧ (rD0 ≠ temp1))
  then (rD) ← (rD0 || 31(-rD0))
  else (rD) ← temp1:32

```

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

nor

NOR

nor rA, rS, rB (Rc=0)
nor. rA, rS, rB (Rc=1)

X Instruction Form

31	rS	rA	rB	124	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The contents of register **rS** are ORed with the contents of register **rB** and the one's complement of the result is loaded into register **rA**.

The one's complement of a number can be obtained using **nor** with **rS = rB**.

Simplified mnemonics defined for this instruction are described in **Other Simplified Mnemonics**, [page 870](#).

Pseudocode

$$(rA) \leftarrow \neg((rS) \vee (rB))$$

Registers Altered

- **rA**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

or

OR

or rA, rS, rB (Rc=0)
or. rA, rS, rB (Rc=1)

X Instruction Form

31	rS	rA	rB	444	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The contents of register **rS** are ORed with the contents of register **rB** and the result is loaded into register **rA**.

The contents of one register can be copied into another register using **or** with **rS = rB**.

Simplified mnemonics defined for this instruction are described in **Other Simplified Mnemonics**, [page 870](#).

Pseudocode

$$(rA) \leftarrow (rS) \vee (rB)$$

Registers Altered

- **rA**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

orc

OR with Complement

orc	rA, rS, rB	(Rc=0)
orc.	rA, rS, rB	(Rc=1)

X Instruction Form

31	rS	rA	rB	412	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The contents of register **rS** are ORed with the one's complement of the contents of register **rB** and the result is loaded into register **rA**.

Pseudocode

$$(rA) \leftarrow (rS) \vee \neg(rB)$$

Registers Altered

- **rA**.
- **CR[CR0]_{LT, GT, EQ, SO}** if **Rc=1**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

ori

OR Immediate

ori rA, rS, UIMM

D Instruction Form

24	rS	rA	UIMM
0	6	1	1
		1	6
			3
			1

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the left. The contents of the register rS are ORed with the extended UIMM field and the result is loaded into register rA.

Simplified mnemonics defined for this instruction are described in **Other Simplified Mnemonics**, [page 870](#). The preferred no-operation (an instruction that does nothing) is:

ori 0,0,0

Pseudocode

$(rA) \leftarrow (rS) \vee ({}^{16}0 \parallel UIMM)$

Registers Altered

- rA.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

oris

OR Immediate Shifted

oris rA, rS, UIMM

D Instruction Form

25	rS	rA	UIMM
0	6	1	1
		1	6
			3
			1

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the right. The contents of the register rS are ORed with the extended UIMM field and the result is loaded into register rA.

Pseudocode

$$(rA) \leftarrow (rS) \vee (UIMM \parallel ^{16}0)$$

Registers Altered

- rA.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

rfci

Return from Critical Interrupt

rfci

XL Instruction Form

19	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	51	0
0	6	2	3
		1	1

Description

This is a privileged instruction.

The MSR is loaded with the contents of SRR3. The contents of SRR2 are used as the next-instruction address (NIA). Program control is transferred to the NIA. This instruction is context synchronizing. Instructions fetched from the NIA use the new context loaded into the MSR.

Pseudocode

```

(MSR) ← (SRR3)
Synchronize context
NIA ← (SRR2)

```

Registers Altered

- MSR.

Exceptions

- Program—Attempted execution of this instruction from user mode.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the operating-environment architecture level (OEA) of the the PowerPC embedded-environment architecture and the PowerPC Book-E architecture. It is not defined by the PowerPC architecture, and is therefore not implemented by all PowerPC processors.

rfi

Return from Interrupt

rfi

XL Instruction Form

19	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	50	0
0	6	2	3
		1	1

Description

This is a privileged instruction.

The MSR is loaded with the contents of SRR1. The contents of SRR0 are used as the next-instruction address (NIA). Program control is transferred to the NIA. This instruction is context synchronizing. Instructions fetched from the NIA use the new context loaded into the MSR.

Pseudocode

```
(MSR) ← (SRR1)
Synchronize context
NIA ← (SRR0)
```

Registers Altered

- MSR.

Exceptions

- Program—Attempted execution of this instruction from user mode.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the operating-environment architecture level (OEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture. It is implemented by all PowerPC processors.

rlwimi

Rotate Left Word Immediate then Mask Insert

rlwimi rA, rS, SH, MB, ME (Rc=0)

rlwimi. rA, rS, SH, MB, ME (Rc=1)

M Instruction Form

20	rS	rA	SH	MB	ME	Rc
0	6	1	1	2	2	3
		1	6	1	6	1

Description

The MB field and ME field specify bit positions in a 32-bit mask, m . m is generated with 1-bits starting at MB and ending at ME, with 0-bits elsewhere. If MB is at a higher bit position than ME, the 1-bits in the mask wrap from the highest bit position to the lowest. Rotate-instruction masks are further described in **Mask Generation**, [page 436](#).

The contents of register rS are rotated left by the number of bit positions specified by the SH field. The rotated data is inserted into register rA under control of the mask. If a mask bit contains a 1, the corresponding bit in the rotated data is inserted into the corresponding bit of register rA. If a mask bit contains a 0, the corresponding bit in rA is not changed.

This instruction can be used to extract a field from one register and insert it into another register.

Simplified mnemonics defined for this instruction are described in **Rotate and Shift Instructions**, [page 865](#).

Pseudocode

$$\begin{aligned} m &\leftarrow \text{MASK}(\text{MB}, \text{ME}) \\ r &\leftarrow \text{ROTL}((rS), \text{SH}) \\ (rA) &\leftarrow (r \wedge m) \vee ((rA) \wedge \neg m) \end{aligned}$$

Registers Altered

- rA.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

rlwinm

Rotate Left Word Immediate then AND with Mask

rlwinm rA, rS, SH, MB, ME (Rc=0)

rlwinm. rA, rS, SH, MB, ME (Rc=1)

M Instruction Form

21	rS	rA	SH	MB	ME	Rc
0	6	1	1	2	2	3
		1	6	1	6	1

Description

The MB field and ME field specify bit positions in a 32-bit mask, m . m is generated with 1-bits starting at MB and ending at ME, with 0-bits elsewhere. If MB is at a higher bit position than ME, the 1-bits in the mask wrap from the highest bit position to the lowest. Rotate-instruction masks are further described in **Mask Generation**, [page 436](#).

The contents of register rS are rotated left by the number of bit positions specified by the SH field. The rotated data is ANDed with the mask and the result is loaded into register rA.

This instruction can be used to extract, rotate, shift, and clear bit fields.

Simplified mnemonics defined for this instruction are described in **Rotate and Shift Instructions**, [page 865](#).

Pseudocode

```

m      ← MASK(MB, ME)
r      ← ROTL((rS), SH)
(rA)   ← r ∧ m

```

Registers Altered

- rA.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

rlwnm

Rotate Left Word then AND with Mask

rlwnm rA, rS, rB, MB, ME (Rc=0)

rlwnm. rA, rS, rB, MB, ME (Rc=1)

M Instruction Form

23	rS	rA	rB	MB	ME	Rc
0	6	1	1	2	2	3
		1	6	1	6	1

Description

The MB field and ME field specify bit positions in a 32-bit mask, m . m is generated with 1-bits starting at MB and ending at ME, with 0-bits elsewhere. If MB is at a higher bit position than ME, the 1-bits in the mask wrap from the highest bit position to the lowest. Rotate-instruction masks are further described in **Mask Generation**, [page 436](#).

The contents of register rS are rotated left by the number of bit positions specified by the contents of register rB_{27:31}. The rotated data is ANDed with the mask and the result is loaded into register rA.

This instruction can be used to extract and rotate bit fields.

Simplified mnemonics defined for this instruction are described in **Rotate and Shift Instructions**, [page 865](#).

Pseudocode

```

m      ← MASK(MB, ME)
r      ← ROTL((rS), (rB)27:31)
(rA)   ← r ∧ m

```

Registers Altered

- rA.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

SC

System Call

SC

SC Instruction Form

17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	6																								3	3
																									0	1

Description

This instruction causes a system-call exception to occur. The contents of the MSR are loaded into SRR1. The address of the instruction immediately following the **sc** instruction is loaded into SRR0.

The MSR[WE, EE, PR, DR, IR] bits are cleared to 0.

The exception-vector address is used as the next-instruction address (NIA) and program control is transferred to the NIA. The exception vector address is formed by concatenating the high halfword of the exception-vector-prefix register (EVPR) to the left of 0x0C00. This instruction is context synchronizing. Instructions fetched from the NIA use the new context loaded into the MSR.

Pseudocode

```

(SRR1)  ← (MSR)
(MSR[WE, EE, PR, DR, IR]) ← 0
(SRR0)  ← CIA + 4
Synchronize context
NIA     ← EVPR0:15 || 0x0C00

```

Registers Altered

- SRR0.
- SRR1.
- MSR[WE, EE, PR, DR, IR].

Exceptions

- System call—execution of this instruction.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture. It is part of the user instruction-set architecture (UISA) and the operating-environment architecture (OEA). It is implemented by all PowerPC processors.

slw

Shift Left Word

slw rA, rS, rB (Rc=0)
 slw. rA, rS, rB (Rc=1)

X Instruction Form

31	rS	rA	rB	24	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The contents of register rS are shifted left by the number of bits specified by the contents of register rB_{27:31}. Bits shifted left out of the most-significant bit are lost and 0-bits fill vacated bit positions on the right. The result is loaded into register rA.

If rB₂₆ = 1, register rA is cleared to zero.

Pseudocode

```

n ← (rB)27:31
r ← ROTL((rS), n)
if (rB)26 = 0
  then m ← MASK(0, 31 – n)
  else m ← 320
(rA) ← r ∧ m
  
```

Registers Altered

- rA.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

sraw

Shift Right Algebraic Word

sraw rA, rS, rB (Rc=0)

sraw. rA, rS, rB (Rc=1)

X Instruction Form

31	rS	rA	rB	792	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The contents of register **rS** are shifted right by the number of bits specified by the contents of register **rB**_{27:31}. Bits shifted right out of the least-significant bit are lost. The most-significant bit of register **rS** (**rS**₀) is replicated to fill vacated bit positions on the left. The result is loaded into register **rA**.

If **rS** contains a negative number and any 1-bits are shifted out of the least-significant bit position, **XER[CA]** is set to 1. Otherwise **XER[CA]** is cleared to 0.

If **rB**₂₆ = 1, **XER[CA]** and all bits in register **rA** are set to the value of **rS**₀.

Pseudocode

```

n    ← (rB)27:31
r    ← ROTL((rS), 32 – n)
if (rB)26 = 0
    then m ← MASK(n, 31)
    else m ← 320
s    ← (rS)0
(rA) ← (r ∧ m) ∨ (32s ∧ ¬m)
XER[CA] ← s ∧ ((r ∧ ¬m) ≠ 0)

```

Registers Altered

- **rA**.
- **XER[CA]**.
- **CR[CR0]_{LT, GT, EQ, SO}** if Rc=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

srawi

Shift Right Algebraic Word Immediate

srawi rA, rS, SH (Rc=0)
srawi. rA, rS, SH (Rc=1)

X Instruction Form

31	rS	rA	SH	824	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The contents of register rS are shifted right by the number of bits specified by the SH field. Bits shifted right out of the least-significant bit are lost. The most-significant bit of register rS (rS₀) is replicated to fill vacated bit positions on the left. The result is loaded into register rA.

If rS contains a negative number and any 1-bits are shifted out of the least-significant bit position, XER[CA] is set to 1. Otherwise XER[CA] is cleared to 0.

Pseudocode

```

n    ← SH
r    ← ROTL((rS), 32 - n)
m    ← MASK(n, 31)
s    ← (rS)0
(rA) ← (r ∧ m) ∨ (32s ∧ ¬m)
XER[CA] ← s ∧ ((r ∧ ¬m) ≠ 0)

```

Registers Altered

- rA.
- XER[CA].
- CR[CR0]_{LT,GT,EQ,SO} if Rc=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

srw

Shift Right Word

srw **rA, rS, rB** (**Rc=0**)

srw. **rA, rS, rB** (**Rc=1**)

X Instruction Form

31	rS	rA	rB	536	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The contents of register **rS** are shifted right by the number of bits specified by the contents of register **rB**_{27:31}. Bits shifted right out of the least-significant bit are lost and 0-bits fill the vacated bit positions on the left. The result is loaded into register **rA**.

If **rB**₂₆ = 1, register **rA** is cleared to 0.

Pseudocode

```

n    ← (rB)27:31
r    ← ROTL((rS), 32 – n)
if (rB)26 = 0
    then m ← MASK(n, 31)
    else m ← 320
(rA) ← r ∧ m

```

Registers Altered

- **rA**.
- **CR[CR0]_{LT, GT, EQ, SO}** if **Rc=1**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

stb

Store Byte

stb **rS, d(rA)****D Instruction Form**

38	rS	rA	d
0	6	1	1
		1	6
			3
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- If the rA field is 0, the base address is 0.
- If the rA field is not 0, the contents of register rA are used as the base address.

The least-significant byte of register rS is stored into the byte referenced by EA.

Pseudocode

$$EA \leftarrow (rA|0) + \text{EXTS}(d)$$

$$\text{MS}(EA, 1) \leftarrow (rS)_{24:31}$$
Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

stbu

Store Byte with Update

stbu rS, d(rA)

D Instruction Form

39	rS	rA	d
0	6	1	1
		1	6
			3
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- The contents of register rA are used as the base address.

The least-significant byte of register rS is stored into the byte referenced by EA. The EA is loaded into rA.

Pseudocode

```
EA      ← (rA) + EXTS(d)
MS(EA, 1) ← (rS)24:31
(rA)    ← EA
```

Registers Altered

- rA.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- rA=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

stbux

Store Byte with Update Indexed

stbux rS, rA, rB

X Instruction Form

31	rS	rA	rB	247	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- The contents of register **rA** are used as the base address.

The least-significant byte of register **rS** is stored into the byte referenced by EA. The EA is loaded into **rA**.

Pseudocode

```

EA      ← (rA) + (rB)
MS(EA, 1) ← (rS)24:31
(rA)    ← EA

```

Registers Altered

- **rA**.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- **rA**=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

stbx

Store Byte Indexed

stbx rS, rA, rB

X Instruction Form

31	rS	rA	rB	215	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The least-significant byte of register **rS** is stored into the byte referenced by EA.

Pseudocode

$$EA \leftarrow (rA|0) + (rB)$$

$$MS(EA, 1) \leftarrow (rS)_{24:31}$$

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

sth

Store Halfword

sth rS, d(rA)

D Instruction Form

44	rS	rA	d
0	6	1	1
		1	6
			3
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- If the rA field is 0, the base address is 0.
- If the rA field is not 0, the contents of register rA are used as the base address.

The least-significant halfword of register rS is stored into the halfword referenced by EA.

Pseudocode

$$EA \leftarrow (rA|0) + \text{EXTS}(d)$$

$$\text{MS}(EA, 2) \leftarrow (rS)_{16:31}$$

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

sthbrx

Store Halfword Byte-Reverse Indexed

sthbrx rS, rA, rB

X Instruction Form

31	rS	rA	rB	918	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The least-significant halfword of register **rS** is byte-reversed and stored into the halfword referenced by EA as follows:

- **rS**[24:31] are stored into the byte referenced by EA.
- **rS**[16:23] are stored into the byte referenced by EA+1.

Pseudocode

$$\begin{aligned} \text{EA} &\leftarrow (\text{rA}|0) + (\text{rB}) \\ \text{MS}(\text{EA}, 2) &\leftarrow (\text{rS})_{24:31} \parallel (\text{rS})_{16:23} \end{aligned}$$

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

sth

Store Halfword with Update

sth rS, d(rA)

D Instruction Form

45	rS	rA	d
0	6	1	1
		1	6
			3
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- The contents of register rA are used as the base address.

The least-significant halfword of register rS is stored into the halfword referenced by EA. The EA is loaded into rA.

Pseudocode

```
EA      ← (rA) + EXTS(d)
MS(EA, 2) ← (rS)16:31
(rA)    ← EA
```

Registers Altered

- rA.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- rA=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

sthux

Store Halfword with Update Indexed

sthux rS, rA, rB

X Instruction Form

31	rS	rA	rB	439	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- The contents of register **rA** are used as the base address.

The least-significant halfword of register **rS** is stored into the halfword referenced by EA. The EA is loaded into **rA**.

Pseudocode

```
EA      ← (rA) + (rB)
MS(EA, 2) ← (rS)16:31
(rA)    ← EA
```

Registers Altered

- **rA**.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- **rA**=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

sthx

Store Halfword Indexed

sthx rS, rA, rB

X Instruction Form

31	rS	rA	rB	407	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The least-significant halfword of register **rS** is stored into the halfword referenced by EA.

Pseudocode

$$EA \leftarrow (rA|0) + (rB)$$

$$MS(EA, 2) \leftarrow (rS)_{16:31}$$

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

stmw

Store Multiple Word

stmw rS, d(rA)

D Instruction Form

47	rS	rA	d
0	6	1	1
		1	6
			3
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- If the rA field is 0, the base address is 0.
- If the rA field is not 0, the contents of register rA are used as the base address.

Let $n = 32 - rS$.

GPRs rS through r31 are stored into n consecutive words starting at the memory address referenced by EA.

Pseudocode

```

EA      ← (rA|0) + EXTS(d)
r       ← rS
do while r ≤ 31
    MS(EA, 4) ← (GPR(r))
    r       ← r + 1
    EA      ← EA + 4

```

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

stswi

Store String Word Immediate

stswi rS, rA, NB

X Instruction Form

31	rS	rA	NB	725	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is determined by the rA field as follows:

- If the rA field is 0, the EA is 0.
- If the rA field is not 0, the contents of register rA are used as the EA.

Let n specify the byte count. If the NB field is 0, n is 32. Otherwise, n is equal to NB.

Let nr specify the number of registers to supply data. $nr = \text{CEIL}(n \div 4)$.

GPRs rS through rS + $nr - 1$ are stored into n consecutive bytes starting at the memory address referenced by EA. The sequence of registers wraps around to r0 if necessary. The bytes within each register are stored beginning with the most-significant byte and ending with the least-significant byte, until the byte count is satisfied.

Pseudocode

```

EA ← (rA|0)
if NB = 0
  then n ← 32
  else n ← NB
r ← rS - 1
i ← 0
do while n > 0
  if i = 0
    then r ← r + 1
  if r = 32
    then r ← 0
  MS(EA,1) ← (GPR(r))i:i+7
  i ← i + 8
  if i = 32
    then i ← 0
  EA ← EA + 1
  n ← n - 1

```

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.

- No-access-allowed zone protection applies only to accesses in user mode.
- Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

stswx

Store String Word Indexed

stswx **rS, rA, rB****X Instruction Form**

31	rS	rA	rB	661	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

Let n specify the byte count contained in **XER[TBC]**.

Let nr specify the number of registers to load with data. $nr = \text{CEIL}(n \div 4)$.

GPRs **rS** through **rS** + $nr - 1$ are stored into n consecutive bytes starting at the memory address referenced by EA. The sequence of registers wraps around to **r0** if necessary. The bytes within each register are stored beginning with the most-significant byte and ending with the least-significant byte, until the byte count is satisfied.

If **XER[TBC]** = 0, **stswx** is treated as a no-operation.

Pseudocode

```

EA  ← (rA[0] + (rB))
n   ← XER[TBC]
r   ← rS - 1
i   ← 0
do while n > 0
  if i = 0
    then r ← r + 1
  if r = 32
    then r ← 0
  MS(EA, 1) ← (GPR(r)[i:i+7])
  i ← i + 8
  if i = 32
    then i ← 0
  EA ← EA + 1
  n  ← n - 1

```

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

If XER[TBC]=0, data-storage and data TLB-miss exceptions do not occur. However, a data machine-check exception can occur when XER[TBC]=0 if the following conditions are true:

- The instruction access passes all protection checks.
- The data address is cachable.
- Access of the data address causes a data-cacheline fill request due to a miss.
- The data-cacheline fill request encounters some form of bus error.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

stw

Store Word

stw rS, d(rA)

D Instruction Form

36	rS	rA	d
0	6	1	3
		1	1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- If the rA field is 0, the base address is 0.
- If the rA field is not 0, the contents of register rA are used as the base address.

The contents of register rS are stored into the word referenced by EA.

Pseudocode

$$EA \leftarrow (rA[0] + \text{EXTS}(d))$$

$$MS(EA, 4) \leftarrow (rS)$$

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

stwbrx

Store Word Byte-Reverse Indexed

stwbrx rS, rA, rB

X Instruction Form

31	rS	rA	rB	662	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The least-significant halfword of register **rS** is byte-reversed and stored into the halfword referenced by EA as follows:

- **rS**[24:31] are stored into the byte referenced by EA.
- **rS**[16:23] are stored into the byte referenced by EA+1.
- **rS**[8:15] are stored into the byte referenced by EA+2.
- **rS**[0:7] are stored into the byte referenced by EA+3.

Pseudocode

$$EA \leftarrow (rA|0) + (rB)$$

$$MS(EA, 4) \leftarrow (rS)_{24:31} \parallel (rS)_{16:23} \parallel (rS)_{8:15} \parallel (rS)_{0:7}$$

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

stwcx.

Store Word Conditional Indexed

stwcx. rS, rA, rB

X Instruction Form

31	rS	rA	rB	150	1
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

If the reservation bit internal to the processor is set to 1 when the instruction is executed, the contents of register **rS** are stored into the word referenced by EA. If the reservation bit is cleared to 0 when the instruction is executed, no store operation is performed. Execution of this instruction always clears the reservation bit.

CR[CR0] is updated as follows:

- CR[CR0]_{LT,GT} are cleared to 0.
- CR[CR0]_{EQ} is set to the state of the reservation bit before the instruction is executed.
- CR[CR0]_{SO} is set to the contents of the XER[SO] bit.

The **lwarx** and the **stwcx.** instructions should be paired in a loop to create the effect of an atomic memory operation when accessing a semaphore. See **Semaphore Synchronization**, [page 462](#) for more information.

Pseudocode

```
EA ← (rA|0) + (rB)
if RESERVE = 1
then
    MS(EA, 4) ← (rS)
    RESERVE ← 0
    (CR[CR0]) ← 0b00 || 1 || XERSO
else
    (CR[CR0]) ← 0b00 || 0 || XERSO
```

Registers Altered

- CR[CR0]_{LT,GT,EQ,SO}

Exceptions

- Alignment—if the EA is not aligned on a word boundary.

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

stwu

Store Word with Update

stwu rS, d(rA)

D Instruction Form

37	rS	rA	d
0	6	1	3
		1	6
			1

Description

An effective address (EA) is calculated by adding a displacement to a base address, which are formed as follows:

- The displacement is formed by sign-extending the 16-bit d instruction field to 32 bits.
- The contents of register rA are used as the base address.

The contents of register rS are stored into the word referenced by EA. The EA is loaded into rA.

Pseudocode

```
EA      ← (rA) + EXTS(d)
MS(EA, 4) ← (rS)
(rA)    ← EA
```

Registers Altered

- rA.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- rA=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

stwux

Store Word with Update Indexed

stwux rS, rA, rB**X Instruction Form**

31	rS	rA	rB	183	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- The contents of register **rA** are used as the base address.

The contents of register **rS** are stored into the word referenced by EA. The EA is loaded into **rA**.

Pseudocode

```
EA      ← (rA) + (rB)
MS(EA, 4) ← (rS)
(rA)    ← EA
```

Registers Altered

- **rA**.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- **rA**=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

stwx

Store Word Indexed

stwx rS, rA, rB

X Instruction Form

31	rS	rA	rB	151	0
0	6	1	1	2	3
		1	6	1	1

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The contents of register **rS** are stored into the word referenced by EA.

Pseudocode

```
EA      ← (rA|0) + (rB)
MS(EA,4) ← (rS)
```

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
 - No-access-allowed zone protection applies only to accesses in user mode.
 - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

subf

Subtract From

subf	rD, rA, rB	(OE=0, Rc=0)
subf.	rD, rA, rB	(OE=0, Rc=1)
subfo	rD, rA, rB	(OE=1, Rc=0)
subfo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	40	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The contents of register **rA** are subtracted from the contents of register **rB**, producing a two's-complement result that is loaded into register **rD**. The subtraction operation is equivalent to adding the contents of register **rB** to the one's complement of register **rA** and adding 1 to the result.

Simplified mnemonics defined for this instruction are described in **Subtract Instructions**, [page 867](#).

Pseudocode

$$(rD) \leftarrow \neg(rA) + (rB) + 1$$

Registers Altered

- **rD**.
- **CR[CR0]_{LT,GT,EQ,SO}** if **Rc=1**.
- **XER[SO,OV]** if **OE=1**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

subfc

Subtract from Carrying

subfc	rD, rA, rB	(OE=0, Rc=0)
subfc.	rD, rA, rB	(OE=0, Rc=1)
subfco	rD, rA, rB	(OE=1, Rc=0)
subfco.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	8	Rc
0	6	1	1	2	2	3
		1	6	1	2	1

Description

The contents of register **rA** are subtracted from the contents of register **rB**, producing a two's-complement result that is loaded into register **rD**. The subtraction operation is equivalent to adding the contents of register **rB** to the one's complement of register **rA** and adding 1 to the result.

XER[CA] is updated to reflect the unsigned magnitude of the result.

Simplified mnemonics defined for this instruction are described in **Subtract Instructions**, [page 867](#).

Pseudocode

```

(rD) ← ¬(rA) + (rB) + 1
if (rD) > 232 - 1
then XER[CA] ← 1
else XER[CA] ← 0

```

Registers Altered

- rD.
- XER[CA].
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

subfe

Subtract from Extended

subfe	rD, rA, rB	(OE=0, Rc=0)
subfe.	rD, rA, rB	(OE=0, Rc=1)
subfeo	rD, rA, rB	(OE=1, Rc=0)
subfeo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	136	Rc
0	6	1	1	2 2		3
		1	6	1 2		1

Description

The contents of register **rB** are added to the one's complement of register **rA**. The contents of **XER[CA]** are added to the result. The result is loaded into register **rD**.

XER[CA] is updated to reflect the unsigned magnitude of the result.

The subtract-from extended instructions can be used to perform subtraction on integers larger than 32 bits, as described on [page 428](#).

Pseudocode

```

(rD) ← ¬(rA) + (rB) + XER[CA]
if (rD) > 232 - 1
then XER[CA] ← 1
else XER[CA] ← 0

```

Registers Altered

- **rD**.
- **XER[CA]**.
- **CR[CR0]_{LT,GT,EQ,SO}** if **Rc=1**.
- **XER[SO,OV]** if **OE=1**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

subfic

Subtract from Immediate Carrying

subfic rD, rA, SIMM

D Instruction Form

8	rD	rA	SIMM
0	6	1	1
		1	6
			3
			1

Description

The contents of register **rA** are subtracted from the sign-extended **SIMM** field, producing a two's-complement result that is loaded into register **rD**. The subtraction operation is equivalent to adding the contents of the **SIMM** field (sign-extended to 32 bits) to the one's complement of register **rA** and adding 1 to the result.

XER[CA] is updated to reflect the unsigned magnitude of the result.

Pseudocode

```

(rD) ← ¬(rA) + EXTS(SIMM) + 1
if (rD) > 232 - 1
  then XER[CA] ← 1
  else XER[CA] ← 0

```

Registers Altered

- **rD**.
- **XER[CA]**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

subfme

Subtract from Minus One Extended

subfme	rD, rA	(OE=0, Rc=0)
subfme.	rD, rA	(OE=0, Rc=1)
subfmeo	rD, rA	(OE=1, Rc=0)
subfmeo.	rD, rA	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	0	0	0	0	0	OE	232	Rc
0	6	1	1					2 2		3
		1	6					1 2		1

Description

The value -1 is added to the one's complement of register **rA**. The contents of XER[CA] are added to the result. The result is loaded into register **rD**.

XER[CA] is updated to reflect the unsigned magnitude of the result.

The subtract-from extended instructions can be used to perform subtraction on integers larger than 32 bits, as described on [page 428](#).

Pseudocode

```

(rD) ← ¬(rA) + 0xFFFF_FFFF + XER[CA]
if (rD) ≥ 232 - 1
  then XER[CA] ← 1
  else XER[CA] ← 0

```

Registers Altered

- **rD**.
- XER[CA].
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.
- XER[SO, OV] if OE=1.

Exceptions

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

subfze

Subtract from Zero Extended

subfze	rD, rA	(OE=0, Rc=0)
subfze.	rD, rA	(OE=0, Rc=1)
subfzeo	rD, rA	(OE=1, Rc=0)
subfzeo.	rD, rA	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	0	0	0	0	0	OE	200	Rc
0	6	1	1					2 2		3
		1	6					1 2		1

Description

The one's complement of register **rA** is added to **XER[CA]** and the result is loaded into register **rD**.

XER[CA] is updated to reflect the unsigned magnitude of the result.

The subtract-from extended instructions can be used to perform subtraction on integers larger than 32 bits, as described on [page 428](#).

Pseudocode

```

(rD) ← ¬(rA) + XER[CA]
if (rD) > 232 - 1
  then XER[CA] ← 1
  else XER[CA] ← 0

```

Registers Altered

- **rD**.
- **XER[CA]**.
- **CR[CR0]_{LT, GT, EQ, SO}** if **Rc=1**.
- **XER[SO, OV]** if **OE=1**.

Exceptions

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

sync

Synchronize

sync

X Instruction Form

31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	598	0
0	6		2		3
			1		1

Description

The **sync** instruction is execution synchronizing. It enforces ordering of all instructions executed by the processor. It ensures that all instructions preceding **sync** in program order complete before **sync** completes. Accesses to main memory caused by instructions preceding the **sync** are completed before the **sync** instruction is completed.

Instructions following the **sync** are not started until the **sync** completes execution. Unlike the **isync** instruction, prefetched instructions are not discarded by the execution of **sync**.

The **sync** instruction can be used to guarantee ordering of both instruction completion and storage access. The **eieio** instruction orders memory access, not instruction completion.

Non-memory instructions following **eieio** can complete before the memory operations ordered by **eieio**. The PPC405, however, implements **eieio** and **sync** identically.

Programmers should use the appropriate ordering instruction to maximize the performance of software that is portable between various PowerPC implementations.

Pseudocode

Synchronize execution

Registers Altered

- None.

Exceptions

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

tlbia

TLB Invalidate All

tlbia

X Instruction Form

31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	370	0
0	6		2		3
			1		1

Description

This is a privileged instruction.

All TLB entries are invalidated. The instruction invalidates a TLB entry by clearing the valid (V) bit in the TLBHI portion of the entry. No other field within the TLB entry is modified by this instruction.

The TLB is invalidated regardless of whether address translation is enabled. A context-synchronizing instruction should follow the **tlbia** instruction to guarantee that the effect of invalidating the TLB is visible to subsequent instructions.

Registers Altered

- None.

Exceptions

- Program—Attempted execution of this instruction from user mode.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined as optional by the operating-environment architecture level (OEA) of the PowerPC architecture and the PowerPC embedded-environment architecture. Because it is optional it is not implemented by all PowerPC processors.

tlbre

TLB Read Entry

tlbre rD, rA, WS

X Instruction Form

31	rD	rA	WS	946	0
0	6	1	1	2	3
		1	6	1	1

Description

This is a privileged instruction.

This instruction reads an entry from the TLB. $rA_{26:31}$ contains an index which is used to select an entry in the TLB. The WS field specifies which portion of the TLB entry is loaded into rD. If WS=0, the tag portion (TLBHI) is loaded into rD and the PID is updated with the TLBHI[TID] field. If WS=1, the data portion (TLBLO) is loaded into rD and the PID is not modified.

See **TLB Entries**, page 512 for a description of the TLB-entry format.

The TLB entry is read regardless of whether address translation is enabled.

Simplified mnemonics defined for this instruction are described in **TLB-Management Instructions**, page 868.

Pseudocode

```

tlb_entry = (rA26:31)
if WS4 = 1
    then (rD) ← TLBLO[tlb_entry]
    else (rD) ← TLBHI[tlb_entry]
    (PID) ← TID from TLB[tlb_entry]

```

Registers Altered

- rD.
- PID if WS=0.

Exceptions

- Program—Attempted execution of this instruction from user mode.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- WS value greater than 1.

Compatibility

This instruction is defined as optional by the operating-environment architecture level (OEA) of the PowerPC embedded-environment architecture and the PowerPC Book-E

architecture. Because it is optional and not defined by the PowerPC architecture it is not implemented by all PowerPC processors.

tlbsx

TLB Search Indexed

tlbsx	rD, rA, rB	(Rc=0)
tlbsx.	rD, rA, rB	(Rc=1)

X Instruction Form

31	rD	rA	rB	914	Rc
0	6	1	1	2	3
		1	6	1	1

Description

This is a privileged instruction.

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **rB** are used as the index.
- If the **rA** field is 0, the base address is 0.
- If the **rA** field is not 0, the contents of register **rA** are used as the base address.

The TLB is searched for a valid entry that translates the combination of the EA and current PID (PID_{24:31}). If a valid entry is found, the corresponding TLB index is loaded into **rD**.

The TLB is searched regardless of whether address translation is enabled.

If Rc=1, CR[CR0] is updated to reflect the search result. If a valid entry is found, CR[CR0]_{EQ} is set to 1. If a valid entry is not found, CR[CR0]_{EQ} is cleared to 0.

Pseudocode

```

EA ← (rA[0] + (rB)
if Rc = 1
then CR[CR0]LT ← 0
   CR[CR0]GT ← 0
   CR[CR0]SO ← XER[SO]
if Valid TLB entry matching EA and PID is in the TLB
then (rD) ← Index of matching TLB Entry
   if Rc = 1
   then CR[CR0]EQ ← 1
else (rD) ← Undefined
   if Rc = 1
   then CR[CR0]EQ ← 0

```

Registers Altered

- **rD**.
- CR[CR0]_{LT, GT, EQ, SO} if Rc=1.

Exceptions

- Program—Attempted execution of this instruction from user mode.

Compatibility

This instruction is defined as optional by the operating-environment architecture level (OEA) of the PowerPC embedded-environment architecture and the PowerPC Book-E architecture. Because it is optional and not defined by the PowerPC architecture it is not implemented by all PowerPC processors.

tlbsync

TLB Synchronize

tlbsync

X Instruction Form

31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	566	0
0	6		2		3
			1		1

Description

This is a privileged instruction.

The **tlbsync** instruction is provided by the PowerPC architecture to support TLB synchronization in multi-processor systems. In the PPC405 this instruction performs no operation. It is provided to facilitate code portability.

Pseudocode

No operation

Registers Altered

- None.

Exceptions

- Program—Attempted execution of this instruction from user mode.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined as optional by the operating-environment architecture level (OEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture. Because it is optional it is not implemented by all PowerPC processors.

tlbwe

TLB Write Entry

tlbwe rS, rA, WS

X Instruction Form

31	rS	rA	WS	978	0
0	6	1	1	2	3
		1	6	1	1

Description

This is a privileged instruction.

This instruction writes a new entry into the TLB. $rA_{26:31}$ contains an index which is used to select an entry in the TLB. The WS field specifies which portion of the TLB entry is written from rS. If WS=0, the tag portion (TLBHI) is written from rS and the PID field ($PID_{24:31}$) is written into the TLBHI[TID] field. If WS=1, the data portion (TLBLO) is written from rS.

See **TLB Entries, page 512** for a description of the TLB-entry format.

The TLB entry is written regardless of whether address translation is enabled. A context-synchronizing instruction should follow the **tlbwe** instruction to guarantee that the effect of writing a TLB entry is visible to subsequent instructions.

Simplified mnemonics defined for this instruction are described in **TLB-Management Instructions, page 868**.

Pseudocode

```

tlb_entry = (rA26:31)
if WS4 = 1
    then TLBLO[tlb_entry] ← (rS)
    else TLBHI[tlb_entry] ← (rS)
         TID of TLB[tlb_entry] ← (PID24:31)

```

Registers Altered

- None.

Exceptions

- Program—Attempted execution of this instruction from user mode.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.
- WS value greater than 1.

Compatibility

This instruction is defined as optional by the operating-environment architecture level (OEA) of the PowerPC embedded-environment architecture and the PowerPC Book-E

architecture. Because it is optional and not defined by the PowerPC architecture it is not implemented by all PowerPC processors.

tw

Trap Word

tw TO, rA, rB

X Instruction Form

31	TO	rA	rB	4	0
0	6	1	1	2	3
		1	6	1	1

Description

The TO opcode field specifies the test conditions to be performed on the contents of registers rA and rB. See [Table 3-13, page 413](#) for more information on the TO field. If any test condition is met, a trap occurs as follows:

- If the trap-instruction debug event is not enabled (DBCR[TDE] = 0, or both DBCR[IDM] = 0 and DBCR[EDM] = 0), a program interrupt occurs.
- If the trap-instruction debug event is enabled as an external-debug event (DBCR[TDE] = 1 and DBCR[EDM] = 1), the processor enters the debug stop state. An external debugger is used to control the processor from this state.

Also, if internal-debug events are enabled (DBCR[IDM] = 1) and debug exceptions are disabled (MSR[DE] = 0), an imprecise debug-event is reported by setting DBSR[IDE] to 1.

- If the trap-instruction debug event is enabled as an internal-debug event (DBCR[TDE] = 1, DBCR[IDM] = 1, and DBCR[EDM] = 0), the action taken depends on whether debug exceptions are enabled:
 - If debug exceptions are enabled (MSR[DE] = 1) a debug interrupt occurs.
 - If debug exceptions are disabled (MSR[DE] = 0) a program interrupt occurs. An imprecise debug-event is also reported by setting DBSR[IDE] to 1.

Refer to the following for more information:

- **Program Interrupt (0x0700), page 547.**
- **Debug Interrupt (0x2000), page 557.**
- **Trap-Instruction Debug Event, page 582.**
- **Internal-Debug Mode, page 572.**
- **External-Debug Mode, page 572.**

Simplified mnemonics defined for this instruction are described in [Trap Instructions, page 868](#).

Pseudocode

```

if ((rA) < (rB)) ∧ (TO0 = 1) then trap
if ((rA) > (rB)) ∧ (TO1 = 1) then trap
if ((rA) = (rB)) ∧ (TO2 = 1) then trap
if ((rA) ≤ (rB)) ∧ (TO3 = 1) then trap
if ((rA) ≥ (rB)) ∧ (TO4 = 1) then trap

```

Registers Altered

- None.

Exceptions

- Program—As specified above.
- Debug—As specified above.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors. However, the behavior of the trap as it relates to the debug exception is implementation-specific.

twi

Trap Word Immediate

twi TO, rA, SIMM

D Instruction Form

3	TO	rA	SIMM
0	6	1	1
		1	6
			3
			1

Description

The TO opcode field specifies the test conditions to be performed on the contents of register rA and the sign-extended SIMM field (sign-extended to 32 bits). See [Table 3-13, page 413](#) for more information on the TO field. If any test condition is met, a trap occurs as follows:

- If the trap-instruction debug event is not enabled (DBCR[TDE] = 0, or both DBCR[IDM] = 0 and DBCR[EDM] = 0), a program interrupt occurs.
- If the trap-instruction debug event is enabled as an external-debug event (DBCR[TDE] = 1 and DBCR[EDM] = 1), the processor enters the debug stop state. An external debugger is used to control the processor from this state.

Also, if internal-debug events are enabled (DBCR[IDM] = 1) and debug exceptions are disabled (MSR[DE] = 0), an imprecise debug-event is reported by setting DBSR[IDE] to 1.

- If the trap-instruction debug event is enabled as an internal-debug event (DBCR[TDE] = 1, DBCR[IDM] = 1, and DBCR[EDM] = 0), the action taken depends on whether debug exceptions are enabled:
 - If debug exceptions are enabled (MSR[DE] = 1) a debug interrupt occurs.
 - If debug exceptions are disabled (MSR[DE] = 0) a program interrupt occurs. An imprecise debug-event is also reported by setting DBSR[IDE] to 1.

Refer to the following for more information:

- **Program Interrupt (0x0700), [page 547](#).**
- **Debug Interrupt (0x2000), [page 557](#).**
- **Trap-Instruction Debug Event, [page 582](#).**
- **Internal-Debug Mode, [page 572](#).**
- **External-Debug Mode, [page 572](#).**

Simplified mnemonics defined for this instruction are described in [Trap Instructions, page 868](#).

Pseudocode

```

if ((rA) <  EXT(SIMM)) ^ (TO0 = 1) then trap
if ((rA) >  EXT(SIMM)) ^ (TO1 = 1) then trap
if ((rA) =  EXT(SIMM)) ^ (TO2 = 1) then trap
if ((rA) ≤  EXT(SIMM)) ^ (TO3 = 1) then trap
if ((rA) ≥  EXT(SIMM)) ^ (TO4 = 1) then trap

```

Registers Altered

- None.

Exceptions

- Program—As specified above.
- Debug—As specified above.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors. However, the behavior of the trap as it relates to the debug exception is implementation-specific.

wrtee

Write External Enable

wrtee rS

X Instruction Form

31	rS	0 0 0 0 0 0 0 0 0 0 0	131	0
0	6	1	2	3
		1	1	1

Description

This is a privileged instruction.

MSR[EE] is set to the value specified by bit 16 in register rS.

Pseudocode

$MSR[EE] \leftarrow (rS)_{16}$

Registers Altered

- MSR[EE].

Exceptions

- Program—Attempted execution of this instruction from user mode.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the operating-environment architecture level (OEA) of the the PowerPC embedded-environment architecture and the PowerPC Book-E architecture. Because it is not defined by the PowerPC architecture it is not implemented by all PowerPC processors.

wrteei

Write External Enable Immediate

wrteei E

X Instruction Form

31	0 0 0 0 0 0 0 0 0 0 0 0	E	0 0 0 0	163	0
0	6	1 6	1 7	2 1	3 1

Description

This is a privileged instruction.

MSR[EE] is set to the value specified by the E opcode field.

Pseudocode

MSR[EE] ← E

Registers Altered

- MSR[EE].

Exceptions

- Program—Attempted execution of this instruction from user mode.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the operating-environment architecture level (OEA) of the the PowerPC embedded-environment architecture and the PowerPC Book-E architecture. Because it is not defined by the PowerPC architecture it is not implemented by all PowerPC processors.

Xor

XOR

xor	rA, rS, rB	(Rc=0)
xor.	rA, rS, rB	(Rc=1)

X Instruction Form

31	rS	rA	rB	316	Rc
0	6	1	1	2	3
		1	6	1	1

Description

The contents of register **rS** are XORed with the contents of register **rB** and the result is loaded into register **rA**.

Pseudocode

$$(rA) \leftarrow (rS) \oplus (rB)$$

Registers Altered

- **rA**.
- **CR[CR0]_{LT, GT, EQ, SO}** if **Rc=1**.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

xori

XOR Immediate

xori rA, rS, UIMM

D Instruction Form

26	rS	rA	UIMM
0	6	1	3
		1	6
			1

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the left. The contents of register rS are XORed with the extended UIMM field and the result is loaded into register rA.

Pseudocode

$$(rA) \leftarrow (rS) \oplus (^{16}0 \parallel \text{UIMM})$$

Registers Altered

- rA.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

xoris

XOR Immediate Shifted

xoris rA, rS, UIMM

D Instruction Form

27	rS	rA	UIMM
0	6	1	1
		1	6
			3
			1

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the right. The contents of register rS are XORed with the extended UIMM field and the result is loaded into register rA.

Pseudocode

$$(rA) \leftarrow (rS) \oplus (UIMM \parallel 160)$$

Registers Altered

- rA.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UIA). It is implemented by all PowerPC processors.

Register Summary

This appendix lists the registers supported by the PPC405. Each table following the register cross-reference shows the register name, its descriptive name, the register number, whether the register is privileged (accessible only from privileged mode), the type of access allowed, and the reset value. In these tables, a column headed “Dec” contains Decimal values and a column headed “Hex” contains hexadecimal values.

Register Cross-Reference

Table A-1 provides a cross-reference to detailed information on all registers supported by the PPC405.

Table A-1: PPC405 Register Cross-Reference

Name	Descriptive Name	Cross Reference
r0–r31	General-Purpose Registers 0–31	General-Purpose Registers (GPRs), page 396
CR	Condition Register	Condition Register (CR), page 397
MSR	Machine-State Register	Machine-State Register, page 467
CCR0	Core-Configuration Register 0	Core-Configuration Register, page 495
CTR	Count Register	Count Register (CTR), page 400
DAC1	Data Address-Compare 1	Data Address-Compare Registers, page 579
DAC2	Data Address-Compare 2	
DBCR0	Debug-Control Register 0	Debug-Control Registers, page 574
DBCR1	Debug-Control Register 1	
DBSR	Debug-Status Register	Debug-Status Register, page 577
DCCR	Data-Cache Cacheability Register	Data-Cache Cacheability Register (DCCR), page 490
DCWR	Data-Cache Write-Through Register	Data-Cache Write-Through Register (DCWR), page 489
DEAR	Data-Error Address Register	Data Exception-Address Register, page 538
DVC1	Data Value-Compare 1	Data Value-Compare Registers, page 579
DVC2	Data Value-Compare 2	
ESR	Exception-Syndrom Register	Exception-Syndrom Register, page 536
EVPR	Exception-Vector Prefix Register	Exception-Vector Prefix Register, page 536

Table A-1: PPC405 Register Cross-Reference (Continued)

Name	Descriptive Name	Cross Reference
IAC1	Instruction Address-Compare 1	Instruction Address-Compare Registers, page 578
IAC2	Instruction Address-Compare 2	
IAC3	Instruction Address-Compare 3	
IAC4	Instruction Address-Compare 4	
ICCR	Instruction-Cache Cacheability Register	Instruction-Cache Cacheability Register (ICCR), page 490
ICDBDR	Instruction-Cache Debug-Data Register	icread Instruction, page 504
LR	Link Register	Link Register (LR), page 399
PID	Process ID Register	Process-ID Register, page 510
PIT	Programmable-Interval Timer	Programmable-Interval Timer Register, page 563
PVR	Processor-Version Register	Processor-Version Register, page 469
SGR	Storage Guarded Register	Storage Guarded Register (SGR), page 491
SLER	Storage Little-Endian Register	Storage Little-Endian Register (SLER), page 491
SPRG0	SPR General-Purpose Register 0	SPR General-Purpose Registers, page 468
SPRG1	SPR General-Purpose Register 1	
SPRG2	SPR General-Purpose Register 2	
SPRG3	SPR General-Purpose Register 3	
SPRG4	SPR General-Purpose Register 4	
SPRG5	SPR General-Purpose Register 5	
SPRG6	SPR General-Purpose Register 6	
SPRG7	SPR General-Purpose Register 7	
SRR0	Save/Restore Register 0	Save/Restore Registers 0 and 1, page 534
SRR1	Save/Restore Register 1	
SRR2	Save/Restore Register 2	Save/Restore Registers 2 and 3, page 535
SRR3	Save/Restore Register 3	
SU0R	Storage User-Defined 0 Register	Storage User-Defined 0 Register (SU0R), page 491
TBL	Time-Base Lower	Time Base, page 560
TBU	Time-Base Upper	
TCR	Timer-Control Register	Timer-Control Register, page 564
TSR	Timer-Status Register	Timer-Status Register, page 565
USPRG0	User SPR General-Purpose Register 0	User-SPR General-Purpose Register, page 400
XER	Fixed-Point Exception Register	Fixed-Point Exception Register (XER), page 399
ZPR	Zone-Protection Register	Zone Protection, page 518

General-Purpose Registers

Table A-2 lists the general-purpose registers (GPRs). A binary version of the register number is shown to assist in interpreting instruction encodings often found in machine-code listings.

Table A-2: General-Purpose Registers

Name	Descriptive Name	Register Number			Privileged	Access	Reset Value
		Dec	Hex	Binary			
r0	General-Purpose Register 0	0	0x00	0b00000	No	Read/Write	Undefined
r1	General-Purpose Register 1	1	0x01	0b00001	No	Read/Write	Undefined
r2	General-Purpose Register 2	2	0x02	0b00010	No	Read/Write	Undefined
r3	General-Purpose Register 3	3	0x03	0b00011	No	Read/Write	Undefined
r4	General-Purpose Register 4	4	0x04	0b00100	No	Read/Write	Undefined
r5	General-Purpose Register 5	5	0x05	0b00101	No	Read/Write	Undefined
r6	General-Purpose Register 6	6	0x06	0b00110	No	Read/Write	Undefined
r7	General-Purpose Register 7	7	0x07	0b00111	No	Read/Write	Undefined
r8	General-Purpose Register 8	8	0x08	0b01000	No	Read/Write	Undefined
r9	General-Purpose Register 9	9	0x09	0b01001	No	Read/Write	Undefined
r10	General-Purpose Register 10	10	0x0A	0b01010	No	Read/Write	Undefined
r11	General-Purpose Register 11	11	0x0B	0b01011	No	Read/Write	Undefined
r12	General-Purpose Register 12	12	0x0C	0b01100	No	Read/Write	Undefined
r13	General-Purpose Register 13	13	0x0D	0b01101	No	Read/Write	Undefined
r14	General-Purpose Register 14	14	0x0E	0b01110	No	Read/Write	Undefined
r15	General-Purpose Register 15	15	0x0F	0b01111	No	Read/Write	Undefined
r16	General-Purpose Register 16	16	0x10	0b10000	No	Read/Write	Undefined
r17	General-Purpose Register 17	17	0x11	0b10001	No	Read/Write	Undefined
r18	General-Purpose Register 18	18	0x12	0b10010	No	Read/Write	Undefined
r19	General-Purpose Register 19	19	0x13	0b10011	No	Read/Write	Undefined
r20	General-Purpose Register 20	20	0x14	0b10100	No	Read/Write	Undefined
r21	General-Purpose Register 21	21	0x15	0b10101	No	Read/Write	Undefined
r22	General-Purpose Register 22	22	0x16	0b10110	No	Read/Write	Undefined
r23	General-Purpose Register 23	23	0x17	0b10111	No	Read/Write	Undefined
r24	General-Purpose Register 24	24	0x18	0b11000	No	Read/Write	Undefined
r25	General-Purpose Register 25	25	0x19	0b11001	No	Read/Write	Undefined
r26	General-Purpose Register 26	26	0x1A	0b11010	No	Read/Write	Undefined
r27	General-Purpose Register 27	27	0x1B	0b11011	No	Read/Write	Undefined
r28	General-Purpose Register 28	28	0x1C	0b11100	No	Read/Write	Undefined
r29	General-Purpose Register 29	29	0x1D	0b11101	No	Read/Write	Undefined
r30	General-Purpose Register 30	30	0x1E	0b11110	No	Read/Write	Undefined
r31	General-Purpose Register 31	31	0x1F	0b11111	No	Read/Write	Undefined

Machine-State Register and Condition Register

Table A-3 lists the machine-state and condition registers. These registers are accessed using special instructions and do not have register numbers associated with them.

Table A-3: Machine-State and Condition Registers

Name	Descriptive Name	Register Number	Privileged	Access	Reset Value
CR	Condition Register	Not Applicable	No	Read/Write	Undefined
MSR	Machine-State Register	Not Applicable	Yes	Read/Write	0x0000_0000

Special-Purpose Registers

Table A-4 lists the special-purpose registers sorted by name. The SPRN is the SPR number that appears in the assembler syntax. The SPRF is the split-field version of the SPRN that appears in the instruction encoding. **Table A-5, page 808** lists the special-purpose registers sorted by SPRN and **Table A-6, page 809** lists the special-purpose registers sorted by SPRF.

The following notes apply to the “Reset Value” column in these tables:

Notes:

- The most-recent reset bits are set as follows:
00—No reset occurred. This is the value of WRS if the watchdog timer *did not* cause the reset.
01—A processor-only reset occurred.
10—A chip reset occurred.
11—A system reset occurred.
All remaining bits are undefined.
- WRC is cleared, disabling watchdog time-out resets. All remaining bits are undefined.

Table A-4: Special-Purpose Registers Sorted by Name

Name	Descriptive Name	SPRN		SPRF		Privileged	Access	Reset Value
		Dec	Hex	Hex	Binary			
CCR0	Core-Configuration Register 0	947	0x3B3	0x27D	0b10011_11101	Yes	Read/Write	Undefined
CTR	Count Register	9	0x009	0x120	0b01001_00000	No	Read/Write	Undefined
DAC1	Data Address-Compare 1	1014	0x3F6	0x2DF	0b10110_11111	Yes	Read/Write	Undefined
DAC2	Data Address-Compare 2	1015	0x3F7	0x2FF	0b10111_11111	Yes	Read/Write	Undefined
DBCR0	Debug-Control Register 0	1010	0x3F2	0x25F	0b10010_11111	Yes	Read/Write	0x0000_0000
DBCR1	Debug-Control Register 1	957	0x3BD	0x3BD	0b11101_11101	Yes	Read/Write	0x0000_0000
DBSR	Debug-Status Register	1008	0x3F0	0x21F	0b10000_11111	Yes	Read/Clear	Undefined ¹
DCCR	Data-Cache Cacheability Register	1018	0x3FA	0x35F	0b11010_11111	Yes	Read/Write	0x0000_0000
DCWR	Data-Cache Write-Through Register	954	0x3BA	0x35D	0b11010_11101	Yes	Read/Write	Undefined
DEAR	Data-Error Address Register	981	0x3D5	0x2BE	0b10101_11110	Yes	Read/Write	Undefined
DVC1	Data Value-Compare 1	950	0x3B6	0x2DD	0b10110_11101	Yes	Read/Write	Undefined
DVC2	Data Value-Compare 2	951	0x3B7	0x2FD	0b10111_11101	Yes	Read/Write	Undefined
ESR	Exception-Syndrome Register	980	0x3D4	0x29E	0b10100_11110	Yes	Read/Write	0x0000_0000
EVPR	Exception-Vector Prefix Register	982	0x3D6	0x2DE	0b10110_11110	Yes	Read/Write	Undefined
IAC1	Instruction Address-Compare 1	1012	0x3F4	0x29F	0b10100_11111	Yes	Read/Write	Undefined
IAC2	Instruction Address-Compare 2	1013	0x3F5	0x2B5	0b10101_11111	Yes	Read/Write	Undefined

Table A-4: Special-Purpose Registers Sorted by Name (Continued)

Name	Descriptive Name	SPRN		SPRF		Privileged	Access	Reset Value
		Dec	Hex	Hex	Binary			
IAC3	Instruction Address-Compare 3	948	0x3B4	0x29D	0b10100_11101	Yes	Read/Write	Undefined
IAC4	Instruction Address-Compare 4	949	0x3B5	0x2BD	0b10101_11101	Yes	Read/Write	Undefined
ICCR	Instruction-Cache Cacheability Register	1019	0x3FB	0x37F	0b11011_11111	Yes	Read/Write	0x0000_0000
ICDBDR	Instruction-Cache Debug-Data Register	979	0x3D3	0x27E	0b10011_11110	Yes	Read-Only	Undefined
LR	Link Register	8	0x008	0x100	0b01000_00000	No	Read/Write	Undefined
PID	Process ID Register	945	0x3B1	0x23D	0b10001_11101	Yes	Read/Write	Undefined
PIT	Programmable-Interval Timer	987	0x3DB	0x37E	0b11011_11110	Yes	Read/Write	Undefined
PVR	Processor-Version Register	287	0x11F	0x3E8	0b11111_01000	Yes	Read-Only	0x2001_0820
SGR	Storage Guarded Register	953	0x3B9	0x33D	0b11001_11101	Yes	Read/Write	0xFFFF_FFFF
SLER	Storage Little-Endian Register	955	0x3BB	0x37D	0b11011_11101	Yes	Read/Write	0x0000_0000
SPRG0	SPR General-Purpose Register 0	272	0x110	0x208	0b10000_01000	Yes	Read/Write	Undefined
SPRG1	SPR General-Purpose Register 1	273	0x111	0x228	0b10001_01000	Yes	Read/Write	Undefined
SPRG2	SPR General-Purpose Register 2	274	0x112	0x248	0b10010_01000	Yes	Read/Write	Undefined
SPRG3	SPR General-Purpose Register 3	275	0x113	0x268	0b10011_01000	Yes	Read/Write	Undefined
SPRG4	SPR General-Purpose Register 4	260	0x104	0x088	0b00100_01000	No	Read-Only	Undefined
SPRG4	SPR General-Purpose Register 4	276	0x114	0x288	0b10100_01000	Yes	Read/Write	Undefined
SPRG5	SPR General-Purpose Register 5	261	0x105	0x0A8	0b00101_01000	No	Read-Only	Undefined
SPRG5	SPR General-Purpose Register 5	277	0x115	0x2A8	0b10101_01000	Yes	Read/Write	Undefined
SPRG6	SPR General-Purpose Register 6	262	0x106	0x0C8	0b00110_01000	No	Read-Only	Undefined
SPRG6	SPR General-Purpose Register 6	278	0x116	0x2C8	0b10110_01000	Yes	Read/Write	Undefined
SPRG7	SPR General-Purpose Register 7	263	0x107	0x0E8	0b00111_01000	No	Read-Only	Undefined
SPRG7	SPR General-Purpose Register 7	279	0x117	0x2E8	0b10111_01000	Yes	Read/Write	Undefined
SRR0	Save/Restore Register 0	26	0x01A	0x340	0b11010_00000	Yes	Read/Write	Undefined
SRR1	Save/Restore Register 1	27	0x01B	0x360	0b11011_00000	Yes	Read/Write	Undefined
SRR2	Save/Restore Register 2	990	0x3DE	0x3DE	0b11110_11110	Yes	Read/Write	Undefined
SRR3	Save/Restore Register 3	991	0x3DF	0x3FE	0b11111_11110	Yes	Read/Write	Undefined
SU0R	Storage User-Defined 0 Register	956	0x3BC	0x39D	0b11100_11101	Yes	Read/Write	0x0000_0000
TBL	Time-Base Lower	284	0x11C	0x388	0b11100_01000	Yes	Write-Only	Undefined
TBU	Time-Base Upper	285	0x11D	0x3A8	0b11101_01000	Yes	Write-Only	Undefined
TCR	Timer-Control Register	986	0x3DA	0x35E	0b11010_11110	Yes	Read/Write	Undefined ²
TSR	Timer-Status Register	984	0x3D8	0x31E	0b11000_11110	Yes	Read/Clear	Undefined ¹
USPRG0	User SPR General-Purpose Register 0	256	0x100	0x008	0b00000_01000	No	Read/Write	Undefined
XER	Fixed-Point Exception Register	1	0x001	0x020	0b00001_00000	No	Read/Write	Undefined
ZPR	Zone-Protection Register	944	0x3B0	0x21D	0b10000_11101	Yes	Read/Write	Undefined

Table A-5 lists the special-purpose registers sorted by the SPRN. The SPRN is the SPR number that appears in the assembler syntax. This table is useful in interpreting assembler listings.

Table A-5: Special-Purpose Registers Sorted by SPRN

Name	Descriptive Name	SPRN		SPRF		Privileged	Access	Reset Value
		Dec	Hex	Hex	Binary			
XER	Fixed-Point Exception Register	1	0x001	0x020	0b00001_00000	No	Read/Write	Undefined
LR	Link Register	8	0x008	0x100	0b01000_00000	No	Read/Write	Undefined
CTR	Count Register	9	0x009	0x120	0b01001_00000	No	Read/Write	Undefined
SRR0	Save/Restore Register 0	26	0x01A	0x340	0b11010_00000	Yes	Read/Write	Undefined
SRR1	Save/Restore Register 1	27	0x01B	0x360	0b11011_00000	Yes	Read/Write	Undefined
USPRG0	User SPR General-Purpose Register 0	256	0x100	0x008	0b00000_01000	No	Read/Write	Undefined
SPRG4	SPR General-Purpose Register 4	260	0x104	0x088	0b00100_01000	No	Read-Only	Undefined
SPRG5	SPR General-Purpose Register 5	261	0x105	0x0A8	0b00101_01000	No	Read-Only	Undefined
SPRG6	SPR General-Purpose Register 6	262	0x106	0x0C8	0b00110_01000	No	Read-Only	Undefined
SPRG7	SPR General-Purpose Register 7	263	0x107	0x0E8	0b00111_01000	No	Read-Only	Undefined
SPRG0	SPR General-Purpose Register 0	272	0x110	0x208	0b10000_01000	Yes	Read/Write	Undefined
SPRG1	SPR General-Purpose Register 1	273	0x111	0x228	0b10001_01000	Yes	Read/Write	Undefined
SPRG2	SPR General-Purpose Register 2	274	0x112	0x248	0b10010_01000	Yes	Read/Write	Undefined
SPRG3	SPR General-Purpose Register 3	275	0x113	0x268	0b10011_01000	Yes	Read/Write	Undefined
SPRG4	SPR General-Purpose Register 4	276	0x114	0x288	0b10100_01000	Yes	Read/Write	Undefined
SPRG5	SPR General-Purpose Register 5	277	0x115	0x2A8	0b10101_01000	Yes	Read/Write	Undefined
SPRG6	SPR General-Purpose Register 6	278	0x116	0x2C8	0b10110_01000	Yes	Read/Write	Undefined
SPRG7	SPR General-Purpose Register 7	279	0x117	0x2E8	0b10111_01000	Yes	Read/Write	Undefined
TBL	Time-Base Lower	284	0x11C	0x388	0b11100_01000	Yes	Write-Only	Undefined
TBU	Time-Base Upper	285	0x11D	0x3A8	0b11101_01000	Yes	Write-Only	Undefined
PVR	Processor-Version Register	287	0x11F	0x3E8	0b11111_01000	Yes	Read-Only	0x2001_0820
ZPR	Zone-Protection Register	944	0x3B0	0x21D	0b10000_11101	Yes	Read/Write	Undefined
PID	Process ID Register	945	0x3B1	0x23D	0b10001_11101	Yes	Read/Write	Undefined
CCR0	Core-Configuration Register 0	947	0x3B3	0x27D	0b10011_11101	Yes	Read/Write	Undefined
IAC3	Instruction Address-Compare 3	948	0x3B4	0x29D	0b10100_11101	Yes	Read/Write	Undefined
IAC4	Instruction Address-Compare 4	949	0x3B5	0x2BD	0b10101_11101	Yes	Read/Write	Undefined
DVC1	Data Value-Compare 1	950	0x3B6	0x2DD	0b10110_11101	Yes	Read/Write	Undefined
DVC2	Data Value-Compare 2	951	0x3B7	0x2FD	0b10111_11101	Yes	Read/Write	Undefined
SGR	Storage Guarded Register	953	0x3B9	0x33D	0b11001_11101	Yes	Read/Write	0xFFFF_FFFF
DCWR	Data-Cache Write-Through Register	954	0x3BA	0x35D	0b11010_11101	Yes	Read/Write	Undefined
SLER	Storage Little-Endian Register	955	0x3BB	0x37D	0b11011_11101	Yes	Read/Write	0x0000_0000
SUOR	Storage User-Defined 0 Register	956	0x3BC	0x39D	0b11100_11101	Yes	Read/Write	0x0000_0000
DBCR1	Debug-Control Register 1	957	0x3BD	0x3BD	0b11101_11101	Yes	Read/Write	0x0000_0000
ICDBDR	Instruction-Cache Debug-Data Register	979	0x3D3	0x27E	0b10011_11110	Yes	Read-Only	Undefined
ESR	Exception-Syndrome Register	980	0x3D4	0x29E	0b10100_11110	Yes	Read/Write	0x0000_0000
DEAR	Data-Error Address Register	981	0x3D5	0x2BE	0b10101_11110	Yes	Read/Write	Undefined

Table A-5: Special-Purpose Registers Sorted by SPRN (Continued)

Name	Descriptive Name	SPRN		SPRF		Privileged	Access	Reset Value
		Dec	Hex	Hex	Binary			
EVPR	Exception-Vector Prefix Register	982	0x3D6	0x2DE	0b10110_11110	Yes	Read/Write	Undefined
TSR	Timer-Status Register	984	0x3D8	0x31E	0b11000_11110	Yes	Read/Clear	Undefined ¹
TCR	Timer-Control Register	986	0x3DA	0x35E	0b11010_11110	Yes	Read/Write	Undefined ²
PIT	Programmable-Interval Timer	987	0x3DB	0x37E	0b11011_11110	Yes	Read/Write	Undefined
SRR2	Save/Restore Register 2	990	0x3DE	0x3DE	0b11110_11110	Yes	Read/Write	Undefined
SRR3	Save/Restore Register 3	991	0x3DF	0x3FE	0b11111_11110	Yes	Read/Write	Undefined
DBSR	Debug-Status Register	1008	0x3F0	0x21F	0b10000_11111	Yes	Read/Clear	Undefined ¹
DBCR0	Debug-Control Register 0	1010	0x3F2	0x25F	0b10010_11111	Yes	Read/Write	0x0000_0000
IAC1	Instruction Address-Compare 1	1012	0x3F4	0x29F	0b10100_11111	Yes	Read/Write	Undefined
IAC2	Instruction Address-Compare 2	1013	0x3F5	0x2B5	0b10101_11111	Yes	Read/Write	Undefined
DAC1	Data Address-Compare 1	1014	0x3F6	0x2DF	0b10110_11111	Yes	Read/Write	Undefined
DAC2	Data Address-Compare 2	1015	0x3F7	0x2FF	0b10111_11111	Yes	Read/Write	Undefined
DCCR	Data-Cache Cacheability Register	1018	0x3FA	0x35F	0b11010_11111	Yes	Read/Write	0x0000_0000
ICCR	Instruction-Cache Cacheability Register	1019	0x3FB	0x37F	0b11011_11111	Yes	Read/Write	0x0000_0000

Table A-6 lists the special-purpose registers sorted by SPRF. The SPRF is the split-field version of the SPRN that appears in the instruction encoding. This table is useful in interpreting machine-code listings.

Table A-6: Special-Purpose Registers Sorted by SPRF

Name	Descriptive Name	SPRN		SPRF		Privileged	Access	Reset Value
		Dec	Hex	Hex	Binary			
USPRG0	User SPR General-Purpose Register 0	256	0x100	0x008	0b00000_01000	No	Read/Write	Undefined
XER	Fixed-Point Exception Register	1	0x001	0x020	0b00001_00000	No	Read/Write	Undefined
SPRG4	SPR General-Purpose Register 4	260	0x104	0x088	0b00100_01000	No	Read-Only	Undefined
SPRG5	SPR General-Purpose Register 5	261	0x105	0x0A8	0b00101_01000	No	Read-Only	Undefined
SPRG6	SPR General-Purpose Register 6	262	0x106	0x0C8	0b00110_01000	No	Read-Only	Undefined
SPRG7	SPR General-Purpose Register 7	263	0x107	0x0E8	0b00111_01000	No	Read-Only	Undefined
LR	Link Register	8	0x008	0x100	0b01000_00000	No	Read/Write	Undefined
CTR	Count Register	9	0x009	0x120	0b01001_00000	No	Read/Write	Undefined
SPRG0	SPR General-Purpose Register 0	272	0x110	0x208	0b10000_01000	Yes	Read/Write	Undefined
ZPR	Zone-Protection Register	944	0x3B0	0x21D	0b10000_11101	Yes	Read/Write	Undefined
DBSR	Debug-Status Register	1008	0x3F0	0x21F	0b10000_11111	Yes	Read/Clear	Undefined ¹
SPRG1	SPR General-Purpose Register 1	273	0x111	0x228	0b10001_01000	Yes	Read/Write	Undefined
PID	Process ID Register	945	0x3B1	0x23D	0b10001_11101	Yes	Read/Write	Undefined
SPRG2	SPR General-Purpose Register 2	274	0x112	0x248	0b10010_01000	Yes	Read/Write	Undefined
DBCR0	Debug-Control Register 0	1010	0x3F2	0x25F	0b10010_11111	Yes	Read/Write	0x0000_0000
SPRG3	SPR General-Purpose Register 3	275	0x113	0x268	0b10011_01000	Yes	Read/Write	Undefined

Table A-6: Special-Purpose Registers Sorted by SPRF (Continued)

Name	Descriptive Name	SPRN		SPRF		Privileged	Access	Reset Value
		Dec	Hex	Hex	Binary			
CCR0	Core-Configuration Register 0	947	0x3B3	0x27D	0b10011_11101	Yes	Read/Write	Undefined
ICDBDR	Instruction-Cache Debug-Data Register	979	0x3D3	0x27E	0b10011_11110	Yes	Read-Only	Undefined
SPRG4	SPR General-Purpose Register 4	276	0x114	0x288	0b10100_01000	Yes	Read/Write	Undefined
IAC3	Instruction Address-Compare 3	948	0x3B4	0x29D	0b10100_11101	Yes	Read/Write	Undefined
ESR	Exception-Syndrome Register	980	0x3D4	0x29E	0b10100_11110	Yes	Read/Write	0x0000_0000
IAC1	Instruction Address-Compare 1	1012	0x3F4	0x29F	0b10100_11111	Yes	Read/Write	Undefined
SPRG5	SPR General-Purpose Register 5	277	0x115	0x2A8	0b10101_01000	Yes	Read/Write	Undefined
IAC2	Instruction Address-Compare 2	1013	0x3F5	0x2B5	0b10101_11111	Yes	Read/Write	Undefined
IAC4	Instruction Address-Compare 4	949	0x3B5	0x2BD	0b10101_11101	Yes	Read/Write	Undefined
DEAR	Data-Error Address Register	981	0x3D5	0x2BE	0b10101_11110	Yes	Read/Write	Undefined
SPRG6	SPR General-Purpose Register 6	278	0x116	0x2C8	0b10110_01000	Yes	Read/Write	Undefined
DVC1	Data Value-Compare 1	950	0x3B6	0x2DD	0b10110_11101	Yes	Read/Write	Undefined
EVPR	Exception-Vector Prefix Register	982	0x3D6	0x2DE	0b10110_11110	Yes	Read/Write	Undefined
DAC1	Data Address-Compare 1	1014	0x3F6	0x2DF	0b10110_11111	Yes	Read/Write	Undefined
SPRG7	SPR General-Purpose Register 7	279	0x117	0x2E8	0b10111_01000	Yes	Read/Write	Undefined
DVC2	Data Value-Compare 2	951	0x3B7	0x2FD	0b10111_11101	Yes	Read/Write	Undefined
DAC2	Data Address-Compare 2	1015	0x3F7	0x2FF	0b10111_11111	Yes	Read/Write	Undefined
TSR	Timer-Status Register	984	0x3D8	0x31E	0b11000_11110	Yes	Read/Clear	Undefined ¹
SGR	Storage Guarded Register	953	0x3B9	0x33D	0b11001_11101	Yes	Read/Write	0xFFFF_FFFF
SRR0	Save/Restore Register 0	26	0x01A	0x340	0b11010_00000	Yes	Read/Write	Undefined
DCWR	Data-Cache Write-Through Register	954	0x3BA	0x35D	0b11010_11101	Yes	Read/Write	Undefined
TCR	Timer-Control Register	986	0x3DA	0x35E	0b11010_11110	Yes	Read/Write	Undefined ²
DCCR	Data-Cache Cacheability Register	1018	0x3FA	0x35F	0b11010_11111	Yes	Read/Write	0x0000_0000
SRR1	Save/Restore Register 1	27	0x01B	0x360	0b11011_00000	Yes	Read/Write	Undefined
SLER	Storage Little-Endian Register	955	0x3BB	0x37D	0b11011_11101	Yes	Read/Write	0x0000_0000
PIT	Programmable-Interval Timer	987	0x3DB	0x37E	0b11011_11110	Yes	Read/Write	Undefined
ICCR	Instruction-Cache Cacheability Register	1019	0x3FB	0x37F	0b11011_11111	Yes	Read/Write	0x0000_0000
TBL	Time-Base Lower	284	0x11C	0x388	0b11100_01000	Yes	Write-Only	Undefined
SU0R	Storage User-Defined 0 Register	956	0x3BC	0x39D	0b11100_11101	Yes	Read/Write	0x0000_0000
TBU	Time-Base Upper	285	0x11D	0x3A8	0b11101_01000	Yes	Write-Only	Undefined
DBCR1	Debug-Control Register 1	957	0x3BD	0x3BD	0b11101_11101	Yes	Read/Write	0x0000_0000
SRR2	Save/Restore Register 2	990	0x3DE	0x3DE	0b11110_11110	Yes	Read/Write	Undefined
PVR	Processor-Version Register	287	0x11F	0x3E8	0b11111_01000	Yes	Read-Only	0x2001_0820
SRR3	Save/Restore Register 3	991	0x3DF	0x3FE	0b11111_11110	Yes	Read/Write	Undefined

Time-Base Registers

Table A-7 lists the time-base registers accessed (read) using the **mftb** instruction. These registers can be written using the **mtspr** instruction (see **Special-Purpose Registers**, [page 806](#) for information on the time-base SPRs). The TBRN is the time-base number that appears in the assembler syntax. The TBRF is the split-field version of the TBRN that appears in the instruction encoding.

Table A-7: Time-Base Registers

Name	Descriptive Name	TBRN		TBRF		Privileged	Access	Reset Value
		Dec	Hex	Hex	Binary			
TBL	Time-Base Lower	268	0x10C	0x188	0b01100_01000	No	Read-Only	Undefined
TBU	Time-Base Upper	269	0x10D	0x1A8	0b01101_01000	No	Read-Only	Undefined

Device Control Registers

Device control registers (DCRs) are not architecturally part of the PPC405. DCRs are used to control, configure, and record status for functional units implemented outside the PPC405 processor but on the same chip. Although the PPC405 does not contain DCRs, the **mfdcr** and **mtdcr** instructions are used by privileged software to access their contents.

Instruction Summary

This appendix lists the PPC405 instruction set sorted by mnemonic, opcode, function, and form. A reference table containing general instruction information such as the architecture level, privilege level, and compatibility is also provided.

In the following tables, reserved fields are shaded gray and contain a value of zero.

Instructions Sorted by Mnemonic

Table B-1 lists the PPC405 instruction set in alphabetical order by mnemonic.

Table B-1: Instructions Sorted by Mnemonic

	0	6	9	11 12	14	16 17	20 21 22	26	30 31	
add	31	rD		rA		rB	OE	266	Rc	
addc	31	rD		rA		rB	OE	10	Rc	
adde	31	rD		rA		rB	OE	138	Rc	
addi	14	rD		rA	SIMM					
addic	12	rD		rA	SIMM					
addic.	13	rD		rA	SIMM					
addis	15	rD		rA	SIMM					
addme	31	rD		rA		00000	OE	234	Rc	
addze	31	rD		rA		00000	OE	202	Rc	
and	31	rS		rA		rB	28		Rc	
andc	31	rS		rA		rB	60		Rc	
andi.	28	rS		rA	UIMM					
andis.	29	rS		rA	UIMM					
b	18	LI							AA	LK
bc	16	BO		BI		BD			AA	LK
bcctr	19	BO		BI		00000		528		LK
bclr	19	BO		BI		00000		16		LK
cmp	31	crfD	00	rA		rB	0		0	
cmpi	11	crfD	00	rA	SIMM					
cmpl	31	crfD	00	rA		rB	32		0	
cmpli	10	crfD	00	rA	SIMM					

Table B-1: Instructions Sorted by Mnemonic (Continued)

	0	6	9	11 12	14	16 17	20 21 22	26	30 31
cntlzw	31	rS	rA	00000				26	Rc
crand	19	crbD	crbA	crbB				257	0
crandc	19	crbD	crbA	crbB				129	0
creqv	19	crbD	crbA	crbB				289	0
crnand	19	crbD	crbA	crbB				225	0
crnor	19	crbD	crbA	crbB				33	0
cror	19	crbD	crbA	crbB				449	0
crorc	19	crbD	crbA	crbB				417	0
crxor	19	crbD	crbA	crbB				193	0
dcba	31	00000	rA	rB				758	0
dcbf	31	00000	rA	rB				86	0
dcbi	31	00000	rA	rB				470	0
dcbst	31	00000	rA	rB				54	0
dcbt	31	00000	rA	rB				278	0
dcbtst	31	00000	rA	rB				246	0
dcbz	31	00000	rA	rB				1014	0
dccci	31	00000	rA	rB				454	0
dcread	31	rD	rA	rB				486	0
divw	31	rD	rA	rB	OE			491	Rc
divwu	31	rD	rA	rB	OE			459	Rc
eieio	31	00000	00000	00000				854	0
eqv	31	rS	rA	rB				284	Rc
extsb	31	rS	rA	00000				954	Rc
extsh	31	rS	rA	00000				922	Rc
icbi	31	00000	rA	rB				982	0
icbt	31	00000	rA	rB				262	0
iccci	31	00000	rA	rB				966	0
icread	31	00000	rA	rB				998	0
isync	19	00000	00000	00000				150	0
lbz	34	rD	rA				d		
lbzu	35	rD	rA				d		
lbzux	31	rD	rA	rB			119		0
lbzx	31	rD	rA	rB			87		0
lha	42	rD	rA				d		
lhau	43	rD	rA				d		
lhaux	31	rD	rA	rB			375		0
lhax	31	rD	rA	rB			343		0
lhbrx	31	rD	rA	rB			790		0

Table B-1: Instructions Sorted by Mnemonic (Continued)

	0	6	9	11	12	14	16	17	20	21	22	26	30	31
lh _z	40	rD				rA						d		
lh _{zu}	41	rD				rA						d		
lh _{zux}	31	rD				rA		rB				311		0
lh _{zx}	31	rD				rA		rB				279		0
lm _w	46	rD				rA						d		
ls _{wi}	31	rD				rA		NB				597		0
ls _{wx}	31	rD				rA		rB				533		0
lw _{arx}	31	rD				rA		rB				20		0
lw _{brx}	31	rD				rA		rB				534		0
lw _z	32	rD				rA						d		
lw _{zu}	33	rD				rA						d		
lw _{zux}	31	rD				rA		rB				55		0
lw _{zx}	31	rD				rA		rB				23		0
macchw	4	rD				rA		rB	OE			172		Rc
macchws	4	rD				rA		rB	OE			236		Rc
macchwsu	4	rD				rA		rB	OE			204		Rc
macchwu	4	rD				rA		rB	OE			140		Rc
machhw	4	rD				rA		rB	OE			44		Rc
machhws	4	rD				rA		rB	OE			108		Rc
machhwsu	4	rD				rA		rB	OE			76		Rc
machhwu	4	rD				rA		rB	OE			12		Rc
maclhw	4	rD				rA		rB	OE			428		Rc
maclhws	4	rD				rA		rB	OE			492		Rc
maclhwsu	4	rD				rA		rB	OE			460		Rc
maclhwu	4	rD				rA		rB	OE			396		Rc
mcrf	19	crfD	00		crfS	00		00000				0		0
mcrxr	31	crfD	00		00000			00000				512		0
mfcr	31	rD			00000			00000				19		0
mf _{dcr}	31	rD						DCRF				323		0
mf _{msr}	31	rD			00000			00000				83		0
mf _{spr}	31	rD						SPRF				339		0
mf _{tb}	31	rD						TBRF				371		0
mtcrf	31	rS	0					CRM		0		144		0
mt _{dcr}	31	rS						DCRF				451		0
mt _{msr}	31	rS			00000			00000				146		0
mt _{spr}	31	rS						SPRF				467		0
mulchw	4	rD				rA		rB				168		Rc
mulchwu	4	rD				rA		rB				136		Rc

Table B-1: Instructions Sorted by Mnemonic (Continued)

	0	6	9	11 12	14	16 17	20 21 22	26	30 31
mulhww	4	rD	rA	rB	40				Rc
mulhwwu	4	rD	rA	rB	8				Rc
mulhw	31	rD	rA	rB	0	75			Rc
mulhwu	31	rD	rA	rB	0	11			Rc
mullhw	4	rD	rA	rB	424				Rc
mullhwu	4	rD	rA	rB	392				Rc
mulli	7	rD	rA	SIMM					
mullw	31	rD	rA	rB	OE	235			Rc
nand	31	rS	rA	rB	476				Rc
neg	31	rD	rA	00000	OE	104			Rc
nmacchw	4	rD	rA	rB	OE	174			Rc
nmacchws	4	rD	rA	rB	OE	238			Rc
nmachhw	4	rD	rA	rB	OE	46			Rc
nmachhws	4	rD	rA	rB	OE	110			Rc
nmachlw	4	rD	rA	rB	OE	430			Rc
nmachlws	4	rD	rA	rB	OE	494			Rc
nor	31	rS	rA	rB	124				Rc
or	31	rS	rA	rB	444				Rc
orc	31	rS	rA	rB	412				Rc
ori	24	rS	rA	UIMM					
oris	25	rS	rA	UIMM					
rfci	19	00000	00000	00000	51				0
rfi	19	00000	00000	00000	50				0
rlwimi	20	rS	rA	SH	MB	ME			Rc
rlwinm	21	rS	rA	SH	MB	ME			Rc
rlwnm	23	rS	rA	rB	MB	ME			Rc
sc	17	00000	00000	00000	00000	0000	1	0	
slw	31	rS	rA	rB	24				Rc
sraw	31	rS	rA	rB	792				Rc
srawi	31	rS	rA	SH	824				Rc
srw	31	rS	rA	rB	536				Rc
stb	38	rS	rA	d					
stbu	39	rS	rA	d					
stbux	31	rS	rA	rB	247				0
stbx	31	rS	rA	rB	215				0
sth	44	rS	rA	d					
sthbrx	31	rS	rA	rB	918				0
sthu	45	rS	rA	d					

Table B-1: Instructions Sorted by Mnemonic (Continued)

	0	6	9	11 12	14	16 17	20 21 22	26	30 31
sthux	31	rS	rA	rB	439				0
sthx	31	rS	rA	rB	407				0
stmw	47	rS	rA	d					
stswi	31	rS	rA	NB	725				0
stswx	31	rS	rA	rB	661				0
stw	36	rS	rA	d					
stwbrx	31	rS	rA	rB	662				0
stwcx.	31	rS	rA	rB	150				1
stwu	37	rS	rA	d					
stwux	31	rS	rA	rB	183				0
stwx	31	rS	rA	rB	151				0
subf	31	rD	rA	rB	OE	40			Rc
subfc	31	rD	rA	rB	OE	8			Rc
subfe	31	rD	rA	rB	OE	136			Rc
subfic	8	rD	rA	SIMM					
subfme	31	rD	rA	00000	OE	232			Rc
subfze	31	rD	rA	00000	OE	200			Rc
sync	31	00000	00000	00000	598				0
tlbia	31	00000	00000	00000	370				0
tlbre	31	rD	rA	WS	946				0
tlbsx	31	rD	rA	rB	914				Rc
tlbsync	31	00000	00000	00000	566				0
tlbwe	31	rS	rA	WS	978				0
tw	31	TO	rA	rB	4				0
twi	3	TO	rA	SIMM					
wrttee	31	rS	00000	00000	131				0
wrtteei	31	00000	00000	E	0000	163			0
xor	31	rS	rA	rB	316				Rc
xori	26	rS	rA	UIMM					
xoris	27	rS	rA	UIMM					

Instructions Sorted by Opcode

Table B-2 lists the PPC405 instruction set in numeric order by primary and secondary opcode.

Table B-2: Instructions Sorted by Opcode

	0	6	9	11 12	14	16 17	20 21 22	26	30 31		
twi	3	TO		rA		SIMM					
mulhhuw	4	rD		rA		rB		8		Rc	
machhuw	4	rD		rA		rB		OE	12		Rc
mulhuw	4	rD		rA		rB		40			Rc
machhuw	4	rD		rA		rB		OE	44		Rc
nmachhuw	4	rD		rA		rB		OE	46		Rc
machhuwsu	4	rD		rA		rB		OE	76		Rc
machhuws	4	rD		rA		rB		OE	108		Rc
nmachhuws	4	rD		rA		rB		OE	110		Rc
mulchwu	4	rD		rA		rB		136			Rc
macchwu	4	rD		rA		rB		OE	140		Rc
mulchw	4	rD		rA		rB		168			Rc
macchw	4	rD		rA		rB		OE	172		Rc
nmacchw	4	rD		rA		rB		OE	174		Rc
macchwsu	4	rD		rA		rB		OE	204		Rc
macchws	4	rD		rA		rB		OE	236		Rc
nmacchws	4	rD		rA		rB		OE	238		Rc
mullhuw	4	rD		rA		rB		392			Rc
macldhuw	4	rD		rA		rB		OE	396		Rc
mullhw	4	rD		rA		rB		424			Rc
macldhw	4	rD		rA		rB		OE	428		Rc
nmacldhw	4	rD		rA		rB		OE	430		Rc
macldhuwsu	4	rD		rA		rB		OE	460		Rc
macldhuws	4	rD		rA		rB		OE	492		Rc
nmacldhuws	4	rD		rA		rB		OE	494		Rc
mulli	7	rD		rA		SIMM					
subfic	8	rD		rA		SIMM					
cmpli	10	crfD	00	rA		SIMM					
cmpi	11	crfD	00	rA		SIMM					
addic	12	rD		rA		SIMM					
addic.	13	rD		rA		SIMM					
addi	14	rD		rA		SIMM					
addis	15	rD		rA		SIMM					
bc	16	BO		BI		BD			AA	LK	
sc	17	00000		00000		00000		00000		0000	1 0
b	18	LI								AA	LK
mcrf	19	crfD	00	crfS	00	00000		0			0

Table B-2: Instructions Sorted by Opcode (Continued)

	0	6	9	11 12	14	16 17	20 21 22	26	30 31	
bclr	19	BO		BI		00000	16			LK
crnor	19	crbD		crbA		crbB	33			0
rfi	19	00000		00000		00000	50			0
rfci	19	00000		00000		00000	51			0
crandc	19	crbD		crbA		crbB	129			0
isync	19	00000		00000		00000	150			0
crxor	19	crbD		crbA		crbB	193			0
crnand	19	crbD		crbA		crbB	225			0
crand	19	crbD		crbA		crbB	257			0
creqv	19	crbD		crbA		crbB	289			0
crorc	19	crbD		crbA		crbB	417			0
cror	19	crbD		crbA		crbB	449			0
bcctr	19	BO		BI		00000	528			LK
rlwimi	20	rS		rA		SH	MB	ME		Rc
rlwinm	21	rS		rA		SH	MB	ME		Rc
rlwnm	23	rS		rA		rB	MB	ME		Rc
ori	24	rS		rA		UIMM				
oris	25	rS		rA		UIMM				
xori	26	rS		rA		UIMM				
xoris	27	rS		rA		UIMM				
andi.	28	rS		rA		UIMM				
andis.	29	rS		rA		UIMM				
cmp	31	crfD	00	rA		rB	0			0
tw	31	TO		rA		rB	4			0
subfc	31	rD		rA		rB	OE	8		Rc
addc	31	rD		rA		rB	OE	10		Rc
mulhwu	31	rD		rA		rB	0	11		Rc
mfcrl	31	rD		00000		00000	19			0
lwarx	31	rD		rA		rB	20			0
lwzx	31	rD		rA		rB	23			0
slw	31	rS		rA		rB	24			Rc
cntlzw	31	rS		rA		00000	26			Rc
and	31	rS		rA		rB	28			Rc
cmpl	31	crfD	00	rA		rB	32			0
subf	31	rD		rA		rB	OE	40		Rc
dcbst	31	00000		rA		rB	54			0
lwzux	31	rD		rA		rB	55			0
andc	31	rS		rA		rB	60			Rc

Table B-2: Instructions Sorted by Opcode (Continued)

	0	6	9	11 12	14	16 17	20 21 22	26	30 31
mulhw	31	rD	rA		rB		0	75	Rc
mfmsr	31	rD	00000		00000		83		0
dcbf	31	00000	rA		rB		86		0
lbzx	31	rD	rA		rB		87		0
neg	31	rD	rA		00000		OE	104	Rc
lbzux	31	rD	rA		rB		119		0
nor	31	rS	rA		rB		124		Rc
wrtree	31	rS	00000		00000		131		0
subfe	31	rD	rA		rB		OE	136	Rc
adde	31	rD	rA		rB		OE	138	Rc
mtcrf	31	rS	0	CRM			0	144	0
mtmsr	31	rS	00000		00000		146		0
stwcx.	31	rS	rA		rB		150		1
stwx	31	rS	rA		rB		151		0
wrtreei	31	00000	00000		E	0000	163		0
stwux	31	rS	rA		rB		183		0
subfze	31	rD	rA		00000		OE	200	Rc
addze	31	rD	rA		00000		OE	202	Rc
stbx	31	rS	rA		rB		215		0
subfme	31	rD	rA		00000		OE	232	Rc
addme	31	rD	rA		00000		OE	234	Rc
mullw	31	rD	rA		rB		OE	235	Rc
dcbtst	31	00000	rA		rB		246		0
stbux	31	rS	rA		rB		247		0
icbt	31	00000	rA		rB		262		0
add	31	rD	rA		rB		OE	266	Rc
dcbt	31	00000	rA		rB		278		0
lhzx	31	rD	rA		rB		279		0
eqv	31	rS	rA		rB		284		Rc
lhzux	31	rD	rA		rB		311		0
xor	31	rS	rA		rB		316		Rc
mfdcr	31	rD	DCRF				323		0
mfspr	31	rD	SPRF				339		0
lhax	31	rD	rA		rB		343		0
tlbia	31	00000	00000		00000		370		0
mftb	31	rD	TBRF				371		0
lhaux	31	rD	rA		rB		375		0
sthx	31	rS	rA		rB		407		0

Table B-2: Instructions Sorted by Opcode (Continued)

	0	6	9	11 12	14	16 17	20 21 22	26	30 31
orc	31	rS		rA		rB		412	Rc
sthux	31	rS		rA		rB		439	0
or	31	rS		rA		rB		444	Rc
mtdcr	31	rS		DCRF				451	0
dccci	31	00000		rA		rB		454	0
divwu	31	rD		rA		rB	OE	459	Rc
mtspr	31	rS		SPRF				467	0
dcbi	31	00000		rA		rB		470	0
nand	31	rS		rA		rB		476	Rc
dcread	31	rD		rA		rB		486	0
divw	31	rD		rA		rB	OE	491	Rc
mcrxr	31	crfD	00	00000		00000		512	0
lswx	31	rD		rA		rB		533	0
lwbrx	31	rD		rA		rB		534	0
srw	31	rS		rA		rB		536	Rc
tlbsync	31	00000		00000		00000		566	0
lswi	31	rD		rA		NB		597	0
sync	31	00000		00000		00000		598	0
stswx	31	rS		rA		rB		661	0
stwbrx	31	rS		rA		rB		662	0
stswi	31	rS		rA		NB		725	0
dcba	31	00000		rA		rB		758	0
lhbrx	31	rD		rA		rB		790	0
sraw	31	rS		rA		rB		792	Rc
srawi	31	rS		rA		SH		824	Rc
eieio	31	00000		00000		00000		854	0
tlbsx	31	rD		rA		rB		914	Rc
sthbrx	31	rS		rA		rB		918	0
extsh	31	rS		rA		00000		922	Rc
tlbre	31	rD		rA		WS		946	0
extsb	31	rS		rA		00000		954	Rc
iccci	31	00000		rA		rB		966	0
tlbwe	31	rS		rA		WS		978	0
icbi	31	00000		rA		rB		982	0
icread	31	00000		rA		rB		998	0
dcbz	31	00000		rA		rB		1014	0
lwz	32	rD		rA		d			
lwzu	33	rD		rA		d			

Table B-2: Instructions Sorted by Opcode (Continued)

	0	6	9	11 12	14	16 17	20 21 22	26	30 31
lbz	34	rD	rA					d	
lbzu	35	rD	rA					d	
stw	36	rS	rA					d	
stwu	37	rS	rA					d	
stb	38	rS	rA					d	
stbu	39	rS	rA					d	
lhz	40	rD	rA					d	
lhzu	41	rD	rA					d	
lha	42	rD	rA					d	
lhau	43	rD	rA					d	
sth	44	rS	rA					d	
sthu	45	rS	rA					d	
lmw	46	rD	rA					d	
stmw	47	rS	rA					d	

Instructions Grouped by Function

Table B-3 through Table B-22 list the PPC405 instruction set grouped by function. Within each table, instructions are sorted in alphabetical order by mnemonic.

Table B-3: Integer Add and Subtract Instructions

	0	6	11	16	21 22	31
add	31	rD	rA	rB	OE	266 Rc
addc	31	rD	rA	rB	OE	10 Rc
adde	31	rD	rA	rB	OE	138 Rc
addi	14	rD	rA			SIMM
addic	12	rD	rA			SIMM
addic.	13	rD	rA			SIMM
addis	15	rD	rA			SIMM
addme	31	rD	rA	00000	OE	234 Rc
addze	31	rD	rA	00000	OE	202 Rc
neg	31	rD	rA	00000	OE	104 Rc
subf	31	rD	rA	rB	OE	40 Rc
subfc	31	rD	rA	rB	OE	8 Rc
subfe	31	rD	rA	rB	OE	136 Rc
subfic	8	rD	rA			SIMM
subfme	31	rD	rA	00000	OE	232 Rc
subfze	31	rD	rA	00000	OE	200 Rc

Table B-4: Integer Divide and Multiply Instructions

	0	6	11	16	21	22	31
divw	31	rD	rA	rB	OE	491	Rc
divwu	31	rD	rA	rB	OE	459	Rc
mulchw	4	rD	rA	rB		168	Rc
mulchwu	4	rD	rA	rB		136	Rc
mulhww	4	rD	rA	rB		40	Rc
mulhwwu	4	rD	rA	rB		8	Rc
mulhw	31	rD	rA	rB	0	75	Rc
mulhwu	31	rD	rA	rB	0	11	Rc
mullhw	4	rD	rA	rB		424	Rc
mullhwu	4	rD	rA	rB		392	Rc
mulli	7	rD	rA	SIMM			
mullw	31	rD	rA	rB	OE	235	Rc

Table B-5: Integer Multiply-Accumulate Instructions

	0	6	11	16	21	22	31
macchw	4	rD	rA	rB	OE	172	Rc
macchws	4	rD	rA	rB	OE	236	Rc
macchwsu	4	rD	rA	rB	OE	204	Rc
macchwu	4	rD	rA	rB	OE	140	Rc
machhw	4	rD	rA	rB	OE	44	Rc
machhws	4	rD	rA	rB	OE	108	Rc
machhwsu	4	rD	rA	rB	OE	76	Rc
machhwu	4	rD	rA	rB	OE	12	Rc
maclhw	4	rD	rA	rB	OE	428	Rc
maclhws	4	rD	rA	rB	OE	492	Rc
maclhwsu	4	rD	rA	rB	OE	460	Rc
maclhwu	4	rD	rA	rB	OE	396	Rc
nmacchw	4	rD	rA	rB	OE	174	Rc
nmacchws	4	rD	rA	rB	OE	238	Rc
nmachhw	4	rD	rA	rB	OE	46	Rc
nmachhws	4	rD	rA	rB	OE	110	Rc
nmaclhw	4	rD	rA	rB	OE	430	Rc
nmaclhws	4	rD	rA	rB	OE	494	Rc

Table B-6: Integer Compare Instructions

	0	6	9	11	16	21	31
cmp	31	crfD	00	rA	rB	0	0
cmpi	11	crfD	00	rA	SIMM		
cmpl	31	crfD	00	rA	rB	32	0
cmpli	10	crfD	00	rA	SIMM		

Table B-7: Integer Logical Instructions

	0	6	11	16	21	31
and	31	rS	rA	rB	28	Rc
andc	31	rS	rA	rB	60	Rc
andi.	28	rS	rA	UIMM		
andis.	29	rS	rA	UIMM		
cntlzw	31	rS	rA	00000	26	Rc
eqv	31	rS	rA	rB	284	Rc
extsb	31	rS	rA	00000	954	Rc
extsh	31	rS	rA	00000	922	Rc
nand	31	rS	rA	rB	476	Rc
nor	31	rS	rA	rB	124	Rc
or	31	rS	rA	rB	444	Rc
orc	31	rS	rA	rB	412	Rc
ori	24	rS	rA	UIMM		
oris	25	rS	rA	UIMM		
xor	31	rS	rA	rB	316	Rc
xori	26	rS	rA	UIMM		
xoris	27	rS	rA	UIMM		

Table B-8: Integer Rotate Instructions

	0	6	11	16	21	26	31
rlwimi	20	rS	rA	SH	MB	ME	Rc
rlwinm	21	rS	rA	SH	MB	ME	Rc
rlwnm	23	rS	rA	rB	MB	ME	Rc

Table B-9: Integer Shift Instructions

	0	6	11	16	21	31
slw	31	rS	rA	rB	24	Rc

Table B-9: Integer Shift Instructions (Continued)

	0	6	11	16	21	31
sraw	31	rS	rA	rB	792	Rc
srawi	31	rS	rA	SH	824	Rc
srw	31	rS	rA	rB	536	Rc

Table B-10: Integer Load Instructions

	0	6	11	16	21	31
lbz	34	rD	rA	d		
lbzu	35	rD	rA	d		
lbzux	31	rD	rA	rB	119	0
lbzx	31	rD	rA	rB	87	0
lha	42	rD	rA	d		
lhau	43	rD	rA	d		
lhaux	31	rD	rA	rB	375	0
lhax	31	rD	rA	rB	343	0
lhz	40	rD	rA	d		
lhzu	41	rD	rA	d		
lhzux	31	rD	rA	rB	311	0
lhzx	31	rD	rA	rB	279	0
lwz	32	rD	rA	d		
lwzu	33	rD	rA	d		
lwzux	31	rD	rA	rB	55	0
lwzx	31	rD	rA	rB	23	0

Table B-11: Integer Store Instructions

	0	6	11	16	21	31
stb	38	rS	rA	d		
stbu	39	rS	rA	d		
stbux	31	rS	rA	rB	247	0
stbx	31	rS	rA	rB	215	0
sth	44	rS	rA	d		
sthu	45	rS	rA	d		
sthux	31	rS	rA	rB	439	0
sthx	31	rS	rA	rB	407	0
stw	36	rS	rA	d		
stwu	37	rS	rA	d		
stwux	31	rS	rA	rB	183	0
stwx	31	rS	rA	rB	151	0

Table B-12: Integer Load and Store with Byte Reverse Instructions

	0	6	11	16	21	31
lhbrx	31	rD	rA	rB	790	0
lwbrx	31	rD	rA	rB	534	0
sthbrx	31	rS	rA	rB	918	0
stwbrx	31	rS	rA	rB	662	0

Table B-13: Integer Load and Store Multiple Instructions

	0	6	11	16	31
lmw	46	rD	rA	d	
stmw	47	rS	rA	d	

Table B-14: Integer Load and Store String Instructions

	0	6	11	16	21	31
lswi	31	rD	rA	NB	597	0
lswx	31	rD	rA	rB	533	0
stswi	31	rS	rA	NB	725	0
stswx	31	rS	rA	rB	661	0

Table B-15: Branch Instructions

	0	6	11	16	21	30	31
b	18	LI				AA	LK
bc	16	BO	BI	BD		AA	LK
bcctr	19	BO	BI	00000	528		LK
bclr	19	BO	BI	00000	16		LK

Table B-16: Condition Register Logical Instructions

	0	6	9	11	14	16	21	31
crand	19	crbD		crbA		crbB	257	0
crandc	19	crbD		crbA		crbB	129	0
creqv	19	crbD		crbA		crbB	289	0
crnand	19	crbD		crbA		crbB	225	0
crnor	19	crbD		crbA		crbB	33	0
cror	19	crbD		crbA		crbB	449	0
crorc	19	crbD		crbA		crbB	417	0
crxor	19	crbD		crbA		crbB	193	0
mcrf	19	crfD	00	crfS	00	00000	0	0

Table B-17: System Linkage Instructions

	0	6	11	16	21	26	30	31
rfci	19	00000	00000	00000		51		0
rfi	19	00000	00000	00000		50		0
sc	17	00000	00000	00000	00000	0000	1	0

Table B-18: Trap Instructions

	0	6	11	16	21	31
tw	31	TO	rA	rB	4	0
twi	3	TO	rA	SIMM		

Table B-19: Synchronization Instructions

	0	6	11	16	21	31
eieio	31	00000	00000	00000	854	0
isync	19	00000	00000	00000	150	0
lwarx	31	rD	rA	rB	20	0
stwcx.	31	rS	rA	rB	150	1
sync	31	00000	00000	00000	598	0

Table B-20: Processor Control Instructions

	0	6	9	11	12	16	17	20	21	31
mcrxr	31	crfD	00	00000	00000	512				0
mfcrr	31	rD		00000	00000	19				0
mfddr	31	rD		DCRF			323			0
mfmsr	31	rD		00000	00000	83				0
mfsprr	31	rD		SPRF			339			0
mftb	31	rD		TBRF			371			0
mtcrf	31	rS	0	CRM			0	144		0
mtddr	31	rS		DCRF			451			0
mtmsr	31	rS		00000	00000	146				0
mtspr	31	rS		SPRF			467			0
wrttee	31	rS		00000	00000	131				0
wrtteei	31	00000		00000	E	0000		163		0

Table B-21: Cache Management Instructions

	0	6	11	16	21	31
dcba	31	00000	rA	rB	758	0
dcbf	31	00000	rA	rB	86	0
dcbi	31	00000	rA	rB	470	0
dcbst	31	00000	rA	rB	54	0
dcbt	31	00000	rA	rB	278	0
dcbtst	31	00000	rA	rB	246	0
dcbz	31	00000	rA	rB	1014	0
dccci	31	00000	rA	rB	454	0
dcread	31	rD	rA	rB	486	0
icbi	31	00000	rA	rB	982	0
icbt	31	00000	rA	rB	262	0
iccci	31	00000	rA	rB	966	0
icread	31	00000	rA	rB	998	0

Table B-22: TLB Management Instructions

	0	6	11	16	21	31
tlbia	31	00000	00000	00000	370	0
tlbre	31	rD	rA	WS	946	0
tlbsx	31	rD	rA	rB	914	Rc
tlbsync	31	00000	00000	00000	566	0
tlbwe	31	rS	rA	WS	978	0

Instructions Grouped by Form

Table B-23 through Table B-31 list the PPC405 instruction set grouped by form. Within each table, instructions are sorted in numeric order by primary and secondary opcode.

Table B-23: B Form

	0	6	11	16	30	31
bc	16	BO	BI	BD	AA	LK

Table B-24: D Form

	0	6	9	11	16	31
twi	3	TO	rA	SIMM		
mulld	7	rD	rA	SIMM		
subfcd	8	rD	rA	SIMM		
cmpld	10	crfD	00	rA	SIMM	

Table B-24: D Form (Continued)

	0	6	9	11	16	31
cmpi	11	crfD	00	rA	SIMM	
addic	12	rD		rA	SIMM	
addic.	13	rD		rA	SIMM	
addi	14	rD		rA	SIMM	
addis	15	rD		rA	SIMM	
ori	24	rS		rA	UIMM	
oris	25	rS		rA	UIMM	
xori	26	rS		rA	UIMM	
xoris	27	rS		rA	UIMM	
andi.	28	rS		rA	UIMM	
andis.	29	rS		rA	UIMM	
lwz	32	rD		rA	d	
lwzu	33	rD		rA	d	
lbz	34	rD		rA	d	
lbzu	35	rD		rA	d	
stw	36	rS		rA	d	
stwu	37	rS		rA	d	
stb	38	rS		rA	d	
stbu	39	rS		rA	d	
lhz	40	rD		rA	d	
lhzu	41	rD		rA	d	
lha	42	rD		rA	d	
lhau	43	rD		rA	d	
sth	44	rS		rA	d	
sthu	45	rS		rA	d	
lmw	46	rD		rA	d	
stmw	47	rS		rA	d	

Table B-25: I Form

	0	6	30	31
b	18	LI	AA	LK

Table B-26: M Form

	0	6	11	16	21	26	31
rlwimi	20	rS	rA	SH	MB	ME	Rc
rlwinm	21	rS	rA	SH	MB	ME	Rc
rlwnm	23	rS	rA	rB	MB	ME	Rc

Table B-27: SC Form

	0	6	11	16	21	26	30	31
sc	17	00000	00000	00000	00000	0000	1	0

Table B-28: X Form

	0	6	9	11	16	17	21	31
mulhww	4	rD		rA	rB		8	Rc
mulhw	4	rD		rA	rB		40	Rc
mulchw	4	rD		rA	rB		136	Rc
mulchw	4	rD		rA	rB		168	Rc
mullww	4	rD		rA	rB		392	Rc
mullhw	4	rD		rA	rB		424	Rc
cmp	31	crfD	00	rA	rB		0	0
tw	31	TO		rA	rB		4	0
mfcrr	31	rD		00000	00000		19	0
lwarx	31	rD		rA	rB		20	0
lwzx	31	rD		rA	rB		23	0
slw	31	rS		rA	rB		24	Rc
cntlzw	31	rS		rA	00000		26	Rc
and	31	rS		rA	rB		28	Rc
cmpl	31	crfD	00	rA	rB		32	0
dcbst	31	00000		rA	rB		54	0
lwzux	31	rD		rA	rB		55	0
andc	31	rS		rA	rB		60	Rc
mfmrr	31	rD		00000	00000		83	0
dcbf	31	00000		rA	rB		86	0
lbzx	31	rD		rA	rB		87	0
lbzux	31	rD		rA	rB		119	0
nor	31	rS		rA	rB		124	Rc
wrttee	31	rS		00000	00000		131	0
mtmrr	31	rS		00000	00000		146	0
stwcx.	31	rS		rA	rB		150	1
stwx	31	rS		rA	rB		151	0
wrtteei	31	00000		00000	E	0000	163	0
stwux	31	rS		rA	rB		183	0
stbx	31	rS		rA	rB		215	0
dcbtst	31	00000		rA	rB		246	0
stbux	31	rS		rA	rB		247	0
icbt	31	00000		rA	rB		262	0

Table B-28: X Form (Continued)

	0	6	9	11	16 17	21	31
dcbt	31	00000		rA	rB	278	0
lhzx	31	rD		rA	rB	279	0
eqv	31	rS		rA	rB	284	Rc
lhzux	31	rD		rA	rB	311	0
xor	31	rS		rA	rB	316	Rc
lhax	31	rD		rA	rB	343	0
tlbia	31	00000	00000	00000		370	0
lhaux	31	rD		rA	rB	375	0
sthx	31	rS		rA	rB	407	0
orc	31	rS		rA	rB	412	Rc
sthux	31	rS		rA	rB	439	0
or	31	rS		rA	rB	444	Rc
dccci	31	00000		rA	rB	454	0
dcbi	31	00000		rA	rB	470	0
nand	31	rS		rA	rB	476	Rc
dcread	31	rD		rA	rB	486	0
mcrxr	31	crfD	00	00000	00000	512	0
lswx	31	rD		rA	rB	533	0
lwbrx	31	rD		rA	rB	534	0
srw	31	rS		rA	rB	536	Rc
tlbsync	31	00000	00000	00000		566	0
lswi	31	rD		rA	NB	597	0
sync	31	00000	00000	00000		598	0
stswx	31	rS		rA	rB	661	0
stwbrx	31	rS		rA	rB	662	0
stswi	31	rS		rA	NB	725	0
dcba	31	00000		rA	rB	758	0
lhbrx	31	rD		rA	rB	790	0
sraw	31	rS		rA	rB	792	Rc
srawi	31	rS		rA	SH	824	Rc
eiio	31	00000	00000	00000		854	0
tlbsx	31	rD		rA	rB	914	Rc
sthbrx	31	rS		rA	rB	918	0
extsh	31	rS		rA	00000	922	Rc
tlbre	31	rD		rA	WS	946	0
extsb	31	rS		rA	00000	954	Rc
iccci	31	00000		rA	rB	966	0
tlbwe	31	rS		rA	WS	978	0

Table B-28: X Form (Continued)

	0	6	9	11	16	17	21	31
icbi	31	00000		rA		rB	982	0
icread	31	00000		rA		rB	998	0
dcbz	31	00000		rA		rB	1014	0

Table B-29: XFX Form

	0	6	11	12	20	21	31
mtcrf	31	rS	0	CRM	0	144	0
mfocr	31	rD		DCRF		323	0
mfopr	31	rD		SPRF		339	0
mftb	31	rD		TBRF		371	0
mtocr	31	rS		DCRF		451	0
mtopr	31	rS		SPRF		467	0

Table B-30: XL Form

	0	6	9	11	14	16	21	31
mcrf	19	crfD	00	crfS	00	00000	0	0
bclr	19	BO		BI		00000	16	LK
crnor	19	crbD		crbA		crbB	33	0
rfi	19	00000		00000		00000	50	0
rfci	19	00000		00000		00000	51	0
crandc	19	crbD		crbA		crbB	129	0
isync	19	00000		00000		00000	150	0
crxor	19	crbD		crbA		crbB	193	0
crnand	19	crbD		crbA		crbB	225	0
crand	19	crbD		crbA		crbB	257	0
creqv	19	crbD		crbA		crbB	289	0
crorc	19	crbD		crbA		crbB	417	0
cror	19	crbD		crbA		crbB	449	0
bcctr	19	BO		BI		00000	528	LK

Table B-31: XO Form

	0	6	11	16	21	22	31
machhwu	4	rD	rA	rB	OE	12	Rc
machhw	4	rD	rA	rB	OE	44	Rc
nmachhw	4	rD	rA	rB	OE	46	Rc
machhwsu	4	rD	rA	rB	OE	76	Rc
machhws	4	rD	rA	rB	OE	108	Rc

Table B-31: XO Form (Continued)

	0	6	11	16	21	22	31
nmacchw	4	rD	rA	rB	OE	110	Rc
macchw	4	rD	rA	rB	OE	140	Rc
nmacchw	4	rD	rA	rB	OE	172	Rc
macchw	4	rD	rA	rB	OE	174	Rc
nmacchw	4	rD	rA	rB	OE	204	Rc
macchw	4	rD	rA	rB	OE	236	Rc
nmacchw	4	rD	rA	rB	OE	238	Rc
macchw	4	rD	rA	rB	OE	396	Rc
nmacchw	4	rD	rA	rB	OE	428	Rc
macchw	4	rD	rA	rB	OE	430	Rc
nmacchw	4	rD	rA	rB	OE	460	Rc
macchw	4	rD	rA	rB	OE	492	Rc
nmacchw	4	rD	rA	rB	OE	494	Rc
subfc	31	rD	rA	rB	OE	8	Rc
addc	31	rD	rA	rB	OE	10	Rc
mulhw	31	rD	rA	rB	0	11	Rc
subf	31	rD	rA	rB	OE	40	Rc
mulhw	31	rD	rA	rB	0	75	Rc
neg	31	rD	rA	00000	OE	104	Rc
subfe	31	rD	rA	rB	OE	136	Rc
adde	31	rD	rA	rB	OE	138	Rc
subfze	31	rD	rA	00000	OE	200	Rc
addze	31	rD	rA	00000	OE	202	Rc
subfme	31	rD	rA	00000	OE	232	Rc
addme	31	rD	rA	00000	OE	234	Rc
mullw	31	rD	rA	rB	OE	235	Rc
add	31	rD	rA	rB	OE	266	Rc
divwu	31	rD	rA	rB	OE	459	Rc
divw	31	rD	rA	rB	OE	491	Rc

Instruction Set Information

Table B-32 classifies general information about the PPC405 instruction set. A lower-case "x" within a cell indicates the instruction is a member of the class specified by the column heading.

Table B-32: Instruction Set Information

Mnemonic	PowerPC Architecture	PowerPC Embedded Environment Architecture	PowerPC Book-E Architecture	Implementation Specific	Architecture Level	Privileged	Optional	Form
add	x	x	x		UISA			XO
addc	x	x	x		UISA			XO
adde	x	x	x		UISA			XO
addi	x	x	x		UISA			D
addic	x	x	x		UISA			D
addic.	x	x	x		UISA			D
addis	x	x	x		UISA			D
addme	x	x	x		UISA			XO
addze	x	x	x		UISA			XO
and	x	x	x		UISA			X
andc	x	x	x		UISA			X
andi.	x	x	x		UISA			D
andis.	x	x	x		UISA			D
b	x	x	x		UISA			I
bc	x	x	x		UISA			B
bcctr	x	x	x		UISA			XL
bclr	x	x	x		UISA			XL
cmp	x	x	x		UISA			X
cmpi	x	x	x		UISA			D
cmpl	x	x	x		UISA			X
cmpli	x	x	x		UISA			D
cntlzw	x	x	x		UISA			X
crand	x	x	x		UISA			XL
crandc	x	x	x		UISA			XL
creqv	x	x	x		UISA			XL
crnand	x	x	x		UISA			XL
crnor	x	x	x		UISA			XL
cror	x	x	x		UISA			XL
crorc	x	x	x		UISA			XL
crxor	x	x	x		UISA			XL
dcba	x	x	x		VEA		x	X
dcbf	x	x	x		VEA			X
dcbi	x	x	x		OEA	x		X
dcbst	x	x	x		VEA			X
dcbt	x	x	x		VEA			X

Table B-32: Instruction Set Information (Continued)

Mnemonic	PowerPC Architecture	PowerPC Embedded Environment Architecture	PowerPC Book-E Architecture	Implementation Specific	Architecture Level	Privileged	Optional	Form
dcbtst	x	x	x		VEA			X
dcbz	x	x	x		VEA			X
dccci				x	OEA	x		X
dcread				x	OEA	x		X
divw	x	x	x		UISA			XO
divwu	x	x	x		UISA			XO
eieio	x	x			VEA			X
eqv	x	x	x		UISA			X
extsb	x	x	x		UISA			X
extsh	x	x	x		UISA			X
icbi	x	x	x		VEA			X
icbt		x	x		VEA			X
iccci				x	OEA	x		X
icread				x	OEA	x		X
isync	x	x	x		VEA			XL
lbz	x	x	x		UISA			D
lbzu	x	x	x		UISA			D
lbzux	x	x	x		UISA			X
lbzx	x	x	x		UISA			X
lha	x	x	x		UISA			D
lhau	x	x	x		UISA			D
lhaux	x	x	x		UISA			X
lhax	x	x	x		UISA			X
lhbrx	x	x	x		UISA			X
lhz	x	x	x		UISA			D
lhzu	x	x	x		UISA			D
lhzux	x	x	x		UISA			X
lhzx	x	x	x		UISA			X
lmw	x	x	x		UISA			D
lswi	x	x	x		UISA			X
lswx	x	x	x		UISA			X
lwarx	x	x	x		UISA			X
lwbrx	x	x	x		UISA			X
lwz	x	x	x		UISA			D
lwzu	x	x	x		UISA			D
lwzux	x	x	x		UISA			X

Table B-32: Instruction Set Information (Continued)

Mnemonic	PowerPC Architecture	PowerPC Embedded Environment Architecture	PowerPC Book-E Architecture	Implementation Specific	Architecture Level	Privileged	Optional	Form
lwzx	x	x	x		UISA			X
macchw				x	UISA			XO
macchws				x	UISA			XO
macchwsu				x	UISA			XO
macchwu				x	UISA			XO
machhw				x	UISA			XO
machhws				x	UISA			XO
machhwsu				x	UISA			XO
machhwu				x	UISA			XO
macldw				x	UISA			XO
macldws				x	UISA			XO
macldwsu				x	UISA			XO
macldwu				x	UISA			XO
mcrf	x	x	x		UISA			XL
mcrxr	x	x	x		UISA			X
mfcrl	x	x	x		UISA			X
mfcdl		x	x		OEA	x		XFx
mfmsr	x	x	x		OEA	x		X
mfspr	x	x	x		UISA			XFx
					OEA	x ¹		
mftb	x	x			VEA			XFx
mtcrf	x	x	x		UISA			XFx
mtcdl		x	x		OEA	x		XFx
mtmsr	x	x	x		OEA	x		X
mtspr	x	x	x		UISA			XFx
					OEA	x ¹		
mulchw				x	UISA			X
mulchwu				x	UISA			X
mulhhw				x	UISA			X
mulhhwu				x	UISA			X
mulhw	x	x	x		UISA			XO
mulhwu	x	x	x		UISA			XO
mullhw				x	UISA			X
mullhwu				x	UISA			X
mulli	x	x	x		UISA			D
mullw	x	x	x		UISA			XO

Table B-32: Instruction Set Information (Continued)

Mnemonic	PowerPC Architecture	PowerPC Embedded Environment Architecture	PowerPC Book-E Architecture	Implementation Specific	Architecture Level	Privileged	Optional	Form
nand	x	x	x		UISA			X
neg	x	x	x		UISA			XO
nmacchw				x	UISA			XO
nmacchws				x	UISA			XO
nmachhw				x	UISA			XO
nmachhws				x	UISA			XO
nmacldhw				x	UISA			XO
nmacldhws				x	UISA			XO
nor	x	x	x		UISA			X
or	x	x	x		UISA			X
orc	x	x	x		UISA			X
ori	x	x	x		UISA			D
oris	x	x	x		UISA			D
rfdi		x	x		OEA	x		XL
rfdi	x	x	x		OEA	x		XL
rlwimi	x	x	x		UISA			M
rlwinm	x	x	x		UISA			M
rlwnm	x	x	x		UISA			M
sc	x	x	x		UISA			SC
slw	x	x	x		UISA			X
sraw	x	x	x		UISA			X
srawi	x	x	x		UISA			X
srw	x	x	x		UISA			X
stb	x	x	x		UISA			D
stbu	x	x	x		UISA			D
stbux	x	x	x		UISA			X
stbx	x	x	x		UISA			X
sth	x	x	x		UISA			D
sthbrx	x	x	x		UISA			X
sthu	x	x	x		UISA			D
sthux	x	x	x		UISA			X
sthx	x	x	x		UISA			X
stmw	x	x	x		UISA			D
stswi	x	x	x		UISA			X
stswx	x	x	x		UISA			X
stw	x	x	x		UISA			D

Table B-32: Instruction Set Information (Continued)

Mnemonic	PowerPC Architecture	PowerPC Embedded Environment Architecture	PowerPC Book-E Architecture	Implementation Specific	Architecture Level	Privileged	Optional	Form
stwbrx	x	x	x		UISA			X
stwcx.	x	x	x		UISA			X
stwu	x	x	x		UISA			D
stwux	x	x	x		UISA			X
stwx	x	x	x		UISA			X
subf	x	x	x		UISA			XO
subfc	x	x	x		UISA			XO
subfe	x	x	x		UISA			XO
subfic	x	x	x		UISA			D
subfme	x	x	x		UISA			XO
subfze	x	x	x		UISA			XO
sync	x	x	x		UISA			X
tlbia	x	x			OEA	x	x	X
tlbre		x	x		OEA	x	x ²	X
tlbsx		x	x		OEA	x	x ²	X
tlbsync	x	x	x		OEA	x	x	X
tlbwe		x	x		OEA	x	x ²	X
tw	x	x	x		UISA			X
twi	x	x	x		UISA			D
wrtee		x	x		OEA			X
wrteei		x	x		OEA			X
xor	x	x	x		UISA			X
xori	x	x	x		UISA			D
xoris	x	x	x		UISA			D

Notes:

1. Execution of this instruction can be either privileged or non-privileged, depending on the SPR number.
2. These instructions are not optional if the PowerPC embedded-environment processor or PowerPC Book-E processor includes a translation look-aside buffer (TLB). The presence of a TLB is optional.

List of Mnemonics and Simplified Mnemonics

Table B-33 provides an alphabetic list of all mnemonics and simplified mnemonics described in this document. If the mnemonic is a simplified mnemonic, its equivalent mnemonic is listed in the column headed “Equivalent Mnemonic”. Otherwise, the column is shaded gray.

Table B-33: Complete List of Instruction Mnemonics

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
add	Add		page 608
add.	Add and Record		
addc	Add Carrying		page 609
addc.	Add Carrying and Record		
addco	Add Carrying with Overflow Enabled		
addco.	Add Carrying with Overflow Enabled and Record		
adde	Add Extended		page 610
adde.	Add Extended and Record		
addeo	Add Extended with Overflow Enabled		
addeo.	Add Extended with Overflow Enabled and Record		
addi	Add Immediate		page 611
addic	Add Immediate Carrying		page 612
addic.	Add Immediate Carrying and Record		page 613
addis	Add Immediate Shifted		page 614
addme	Add to Minus One Extended		page 615
addme.	Add to Minus One Extended and Record		
addmeo	Add to Minus One Extended with Overflow Enabled		
addmeo.	Add to Minus One Extended with Overflow Enabled and Record		
addo	Add with Overflow Enabled		page 608
addo.	Add with Overflow Enabled and Record		
addze	Add to Zero Extended		page 616
addze.	Add to Zero Extended and Record		
addzeo	Add to Zero Extended with Overflow Enabled		
addzeo.	Add to Zero Extended with Overflow Enabled and Record		
and	AND		page 617
and.	AND and Record		
andc	AND with Complement		page 618
andc.	AND with Complement and Record		
andi.	AND Immediate and Record		page 619
andis.	AND Immediate Shifted and Record		page 620
b	Branch		page 621
ba	Branch Absolute		
bc	Branch Conditional		page 622
bca	Branch Conditional Absolute		

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
bcctr	Branch Conditional to Count Register		page 624
bcctrl	Branch Conditional to Count Register and Link		
bcl	Branch Conditional and Link		page 622
bcla	Branch Conditional Absolute and Link		
bclr	Branch Conditional to Link Register		page 626
bclrl	Branch Conditional to Link Register and Link		
bctr	Branch to Count Register	bcctr	page 859
bctrl	Branch to Count Register and Link	bcctrl	page 860
bdnz	Branch if Decremental CTR Not Zero	bc	page 858
bdnza	Branch if Decremental CTR Not Zero Absolute	bca	page 858
bdnzf	Branch if Decremental CTR Not Zero and Condition False	bc	page 858
bdnzfa	Branch if Decremental CTR Not Zero and Condition False Absolute	bca	page 858
bdnzfl	Branch if Decremental CTR Not Zero and Condition False and Link	bcl	page 859
bdnzfla	Branch if Decremental CTR Not Zero and Condition False Absolute and Link	bcla	page 859
bdnzflr	Branch if Decremental CTR Not Zero and Condition False to Link Register	bclr	page 859
bdnzflrl	Branch if Decremental CTR Not Zero and Condition False to Link Register and Link	bclrl	page 860
bdnzl	Branch if Decremental CTR Not Zero and Link	bcl	page 859
bdnzla	Branch if Decremental CTR Not Zero Absolute and Link	bcla	page 859
bdnzlr	Branch if Decremental CTR Not Zero to Link Register	bclr	page 859
bdnzlrl	Branch if Decremental CTR Not Zero to Link Register and Link	bclrl	page 860
bdnzt	Branch if Decremental CTR Not Zero and Condition True	bc	page 858
bdnzta	Branch if Decremental CTR Not Zero and Condition True Absolute	bca	page 858
bdnztl	Branch if Decremental CTR Not Zero and Condition True and Link	bcl	page 859
bdnztla	Branch if Decremental CTR Not Zero and Condition True Absolute and Link	bcla	page 859
bdnztlr	Branch if Decremental CTR Not Zero and Condition True to Link Register	bclr	page 859
bdnztlrl	Branch if Decremental CTR Not Zero and Condition True to Link Register and Link	bclrl	page 860
bdz	Branch if Decremental CTR Zero	bc	page 858
bdza	Branch if Decremental CTR Zero Absolute	bca	page 858
bdzf	Branch if Decremental CTR Zero and Condition False	bc	page 858
bdzfa	Branch if Decremental CTR Zero and Condition False Absolute	bca	page 858
bdzfl	Branch if Decremental CTR Zero and Condition False and Link	bcl	page 859
bdzfla	Branch if Decremental CTR Zero and Condition False Absolute and Link	bcla	page 859
bdzflr	Branch if Decremental CTR Zero and Condition False to Link Register	bclr	page 859
bdzflrl	Branch if Decremental CTR Zero and Condition False to Link Register and Link	bclrl	page 860
bdzl	Branch if Decremental CTR Zero and Link	bcl	page 859
bdzla	Branch if Decremental CTR Zero Absolute and Link	bcla	page 859

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
bdzlr	Branch if Decrement CTR Zero to Link Register	bclr	page 859
bdzlrl	Branch if Decrement CTR Zero to Link Register and Link	bclrl	page 860
bdzt	Branch if Decrement CTR Zero and Condition True	bc	page 858
bdzta	Branch if Decrement CTR Zero and Condition True Absolute	bca	page 858
bdztl	Branch if Decrement CTR Zero and Condition True and Link	bcl	page 859
bdztla	Branch if Decrement CTR Zero and Condition True Absolute and Link	bcla	page 859
bdztlr	Branch if Decrement CTR Zero and Condition True to Link Register	bclr	page 859
bdztlrl	Branch if Decrement CTR Zero and Condition True to Link Register and Link	bclrl	page 860
beq	Branch if Equal	bc	page 861
beqa	Branch if Equal Absolute	bca	page 861
beqctr	Branch if Equal to Count Register	bcctr	page 862
beqctrl	Branch if Equal to Count Register and Link	bcctrl	page 863
beql	Branch if Equal and Link	bcl	page 862
beqla	Branch if Equal Absolute and Link	bcla	page 862
beqlr	Branch if Equal to Link Register	bclr	page 862
beqlrl	Branch if Equal to Link Register and Link	bclrl	page 863
bf	Branch if Condition False	bc	page 858
bfa	Branch if Condition False Absolute	bca	page 858
bfcctr	Branch if Condition False to Count Register	bcctr	page 859
bfcctrl	Branch if Condition False to Count Register and Link	bcctrl	page 860
bfl	Branch if Condition False and Link	bcl	page 859
bfla	Branch if Condition False Absolute and Link	bcla	page 859
bflr	Branch if Condition False to Link Register	bclr	page 859
bflrl	Branch if Condition False to Link Register and Link	bclrl	page 860
bge	Branch if Greater Than or Equal	bc	page 861
bgea	Branch if Greater Than or Equal Absolute	bca	page 861
bgectr	Branch if Greater Than or Equal to Count Register	bcctr	page 862
bgectrl	Branch if Greater Than or Equal to Count Register and Link	bcctrl	page 863
bgecl	Branch if Greater Than or Equal and Link	bcl	page 862
bgecla	Branch if Greater Than or Equal Absolute and Link	bcla	page 862
bgeclr	Branch if Greater Than or Equal to Link Register	bclr	page 862
bgeclrl	Branch if Greater Than or Equal to Link Register and Link	bclrl	page 863
bgt	Branch if Greater Than	bc	page 861
bgta	Branch if Greater Than Absolute	bca	page 861
bgtctr	Branch if Greater Than to Count Register	bcctr	page 862
bgtctrl	Branch if Greater Than to Count Register and Link	bcctrl	page 863

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
bgtl	Branch if Greater Than and Link	bcl	page 862
bgtla	Branch if Greater Than Absolute and Link	bcla	page 862
bgtlr	Branch if Greater Than to Link Register	bclr	page 862
bgtlrl	Branch if Greater Than to Link Register and Link	bclrl	page 863
bl	Branch and Link		page 621
bla	Branch Absolute and Link		
ble	Branch if Less Than or Equal	bc	page 861
blea	Branch if Less Than or Equal Absolute	bca	page 861
blectr	Branch if Less Than or Equal to Count Register	bcctr	page 862
blectrl	Branch if Less Than or Equal to Count Register and Link	bcctrl	page 863
blel	Branch if Less Than or Equal and Link	bcl	page 862
blela	Branch if Less Than or Equal Absolute and Link	bcla	page 862
blelr	Branch if Less Than or Equal to Link Register	bclr	page 862
blelrl	Branch if Less Than or Equal to Link Register and Link	bclrl	page 863
blr	Branch to Link Register	bclr	page 859
blrl	Branch to Link Register and Link	bclrl	page 860
blt	Branch if Less Than	bc	page 861
blta	Branch if Less Than Absolute	bca	page 861
bltctr	Branch if Less Than to Count Register	bcctr	page 862
bltctrl	Branch if Less Than to Count Register and Link	bcctrl	page 863
bltl	Branch if Less Than and Link	bcl	page 862
bltla	Branch if Less Than Absolute and Link	bcla	page 862
bltlr	Branch if Less Than to Link Register	bclr	page 862
bltlrl	Branch if Less Than to Link Register and Link	bclrl	page 863
bne	Branch if Not Equal	bc	page 861
bnea	Branch if Not Equal Absolute	bca	page 861
bnctr	Branch if Not Equal to Count Register	bcctr	page 862
bnctrl	Branch if Not Equal to Count Register and Link	bcctrl	page 863
bnel	Branch if Not Equal and Link	bcl	page 862
bnela	Branch if Not Equal Absolute and Link	bcla	page 862
bnelr	Branch if Not Equal to Link Register	bclr	page 862
bnelrl	Branch if Not Equal to Link Register and Link	bclrl	page 863
bng	Branch if Not Greater Than	bc	page 861
bnga	Branch if Not Greater Than Absolute	bca	page 861
bnctr	Branch if Not Greater Than to Count Register	bcctr	page 862
bngctrl	Branch if Not Greater Than to Count Register and Link	bcctrl	page 863

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
bngl	Branch if Not Greater Than and Link	bcl	page 862
bngla	Branch if Not Greater Than Absolute and Link	bcla	page 862
bnglr	Branch if Not Greater Than to Link Register	bclr	page 862
bnglrl	Branch if Not Greater Than to Link Register and Link	bclrl	page 863
bnl	Branch if Not Less Than	bc	page 861
bnla	Branch if Not Less Than Absolute	bca	page 861
bnlctr	Branch if Not Less Than to Count Register	bcctr	page 862
bnlctrl	Branch if Not Less Than to Count Register and Link	bcctrl	page 863
bnll	Branch if Not Less Than and Link	bcl	page 862
bnlla	Branch if Not Less Than Absolute and Link	bcla	page 862
bnllr	Branch if Not Less Than to Link Register	bclr	page 862
bnllrl	Branch if Not Less Than to Link Register and Link	bclrl	page 863
bns	Branch if Not Summary Overflow	bc	page 861
bnsa	Branch if Not Summary Overflow Absolute	bca	page 861
bnsctr	Branch if Not Summary Overflow to Count Register	bcctr	page 862
bnsctrl	Branch if Not Summary Overflow to Count Register and Link	bcctrl	page 863
bnsll	Branch if Not Summary Overflow and Link	bcl	page 862
bnslla	Branch if Not Summary Overflow Absolute and Link	bcla	page 862
bnsllr	Branch if Not Summary Overflow to Link Register	bclr	page 862
bnsllrl	Branch if Not Summary Overflow to Link Register and Link	bclrl	page 863
bso	Branch if Summary Overflow	bc	page 861
bsoa	Branch if Summary Overflow Absolute	bca	page 861
bsocctr	Branch if Summary Overflow to Count Register	bcctr	page 862
bsocctrl	Branch if Summary Overflow to Count Register and Link	bcctrl	page 863
bsol	Branch if Summary Overflow and Link	bcl	page 862
bsola	Branch if Summary Overflow Absolute and Link	bcla	page 862
bsolr	Branch if Summary Overflow to Link Register	bclr	page 862
bsolrl	Branch if Summary Overflow to Link Register and Link	bclrl	page 863
bt	Branch if Condition True	bc	page 858
bta	Branch if Condition True Absolute	bca	page 858
btctr	Branch if Condition True to Count Register	bcctr	page 859
btctrl	Branch if Condition True to Count Register and Link	bcctrl	page 860
btl	Branch if Condition True and Link	bcl	page 859
btla	Branch if Condition True Absolute and Link	bcla	page 859
btlr	Branch if Condition True to Link Register	bclr	page 859
btllrl	Branch if Condition True to Link Register and Link	bclrl	page 860

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
clrlslwi	Clear Left and Shift Left Immediate	rlwinm	page 865
clrlslwi.	Clear Left and Shift Left Immediate and Record	rlwinm.	
clrlwi	Clear Left Immediate	rlwinm	
clrlwi.	Clear Left Immediate and Record	rlwinm.	
clrrwi	Clear Right Immediate	rlwinm	
clrrwi.	Clear Right Immediate and Record	rlwinm.	
cmp	Compare		page 628
cmpi	Compare Immediate		page 629
cmpl	Compare Logical		page 630
cmpli	Compare Logical Immediate		page 631
cmplw	Compare Logical Word	cmpl	page 864
cmplwi	Compare Logical Word Immediate	cmpli	
cmpw	Compare Word	cmp	
cmpwi	Compare Word Immediate	cmpi	
cntlzw	Count Leading Zeros Word		page 632
cntlzw.	Count Leading Zeros Word and Record		
crand	Condition Register AND		page 633
crandc	Condition Register AND with Complement		page 634
crclr	Condition Register Clear	crxor	page 864
creqv	Condition Register Equivalent		page 635
crmove	Condition Register Move	cror	page 864
crnand	Condition Register NAND		page 636
crnor	Condition Register NOR		page 637
crnot	Condition Register Not	crnor	page 864
cror	Condition Register OR		page 638
crorc	Condition Register OR with Complement		page 639
crset	Condition Register Set	creqv	page 864
crxor	Condition Register XOR		page 640
dcba	Data Cache Block Allocate		page 641
dcbf	Data Cache Block Flush		page 643
dcbi	Data Cache Block Invalidate		page 645
dcbst	Data Cache Block Store		page 647
dcbt	Data Cache Block Touch		page 649
dcbtst	Data Cache Block Touch for Store		page 651
dcbz	Data Cache Block Clear to Zero		page 653
dccci	Data Cache Congruence Class Invalidate		page 655

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
dcread	Data Cache Read		page 657
divw	Divide Word		page 659
divw.	Divide Word and Record		
divwo	Divide Word with Overflow Enabled		
divwo.	Divide Word with Overflow Enabled and Record		
divwu	Divide Word Unsigned		page 661
divwu.	Divide Word Unsigned and Record		
divwuo	Divide Word Unsigned with Overflow Enabled		
divwuo.	Divide Word Unsigned with Overflow Enabled and Record		
eieio	Enforce In-Order Execution of I/O		page 662
eqv	Equivalent		page 663
eqv.	Equivalent and Record		
extlwi	Extract and Left Justify Immediate	rlwinm	page 865
extlwi.	Extract and Left Justify Immediate and Record	rlwinm.	
extrwi	Extract and Right Justify Immediate	rlwinm	
extrwi.	Extract and Right Justify Immediate and Record	rlwinm.	
extsb	Extend Sign Byte		page 664
extsb.	Extend Sign Byte and Record		
extsh	Extend Sign Halfword		page 665
extsh.	Extend Sign Halfword and Record		
icbi	Instruction Cache Block Invalidate		page 666
icbt	Instruction Cache Block Touch		page 668
iccci	Instruction Cache Congruence Class Invalidate		page 670
icread	Instruction Cache Read		page 671
inslwi	Insert from Left Immediate	rlwimi	page 865
inslwi.	Insert from Left Immediate and Record	rlwimi.	
insrwi	Insert from Right Immediate	rlwimi	
insrwi.	Insert from Right Immediate and Record	rlwimi.	
isync	Instruction Synchronize		page 673
la	Load Address	addi	page 870
lbz	Load Byte and Zero		page 674
lbzu	Load Byte and Zero with Update		page 675
lbzux	Load Byte and Zero with Update Indexed		page 676
lbzx	Load Byte and Zero Indexed		page 677
lha	Load Halfword Algebraic		page 678
lhau	Load Halfword Algebraic with Update		page 679

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
lhax	Load Halfword Algebraic with Update Indexed		page 680
lhax	Load Halfword Algebraic Indexed		page 681
lhbrx	Load Halfword Byte-Reverse Indexed		page 682
lhz	Load Halfword and Zero		page 683
lhzu	Load Halfword and Zero with Update		page 684
lhzux	Load Halfword and Zero with Update Indexed		page 685
lhzx	Load Halfword and Zero Indexed		page 686
li	Load Immediate	addi	page 870
lis	Load Immediate Shifted	addis	page 870
lmw	Load Multiple Word		page 687
lswi	Load String Word Immediate		page 689
lswx	Load String Word Indexed		page 691
lwarx	Load Word and Reserve Indexed		page 693
lwbrx	Load Word Byte-Reverse Indexed		page 694
lwz	Load Word and Zero		page 695
lwzu	Load Word and Zero with Update		page 696
lwzux	Load Word and Zero with Update Indexed		page 697
lwzx	Load Word and Zero Indexed		page 698
macchw	Multiply Accumulate Cross Halfword to Word Modulo Signed		page 699
macchw.	Multiply Accumulate Cross Halfword to Word Modulo Signed and Record		
macchwo	Multiply Accumulate Cross Halfword to Word Modulo Signed with Overflow Enabled		
macchwo.	Multiply Accumulate Cross Halfword to Word Modulo Signed with Overflow Enabled and Record		
macchws	Multiply Accumulate Cross Halfword to Word Saturate Signed		page 700
macchws.	Multiply Accumulate Cross Halfword to Word Saturate Signed and Record		
macchwso	Multiply Accumulate Cross Halfword to Word Saturate Signed with Overflow Enabled		
macchwso.	Multiply Accumulate Cross Halfword to Word Saturate Signed with Overflow Enabled and Record		
macchwsu	Multiply Accumulate Cross Halfword to Word Saturate Unsigned		page 701
macchwsu.	Multiply Accumulate Cross Halfword to Word Saturate Unsigned and Record		
macchwsuo	Multiply Accumulate Cross Halfword to Word Saturate Unsigned with Overflow Enabled		
macchwsuo.	Multiply Accumulate Cross Halfword to Word Saturate Unsigned with Overflow Enabled and Record		
macchwu	Multiply Accumulate Cross Halfword to Word Modulo Unsigned		page 702
macchwu.	Multiply Accumulate Cross Halfword to Word Modulo Unsigned and Record		
macchwuo	Multiply Accumulate Cross Halfword to Word Modulo Unsigned with Overflow Enabled		
macchwuo.	Multiply Accumulate Cross Halfword to Word Modulo Unsigned with Overflow Enabled and Record		

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
machhw	Multiply Accumulate High Halfword to Word Modulo Signed		page 703
machhw.	Multiply Accumulate High Halfword to Word Modulo Signed and Record		
machhwo	Multiply Accumulate High Halfword to Word Modulo Signed with Overflow Enabled		
machhwo.	Multiply Accumulate High Halfword to Word Modulo Signed with Overflow Enabled and Record		
machhws	Multiply Accumulate High Halfword to Word Saturate Signed		page 704
machhws.	Multiply Accumulate High Halfword to Word Saturate Signed and Record		
machhwso	Multiply Accumulate High Halfword to Word Saturate Signed with Overflow Enabled		
machhwso.	Multiply Accumulate High Halfword to Word Saturate Signed with Overflow Enabled and Record		
machhwsu	Multiply Accumulate High Halfword to Word Saturate Unsigned		page 705
machhwsu.	Multiply Accumulate High Halfword to Word Saturate Unsigned and Record		
machhwsuo	Multiply Accumulate High Halfword to Word Saturate Unsigned with Overflow Enabled		
machhwsuo.	Multiply Accumulate High Halfword to Word Saturate Unsigned with Overflow Enabled and Record		
machhwu	Multiply Accumulate High Halfword to Word Modulo Unsigned		page 706
machhwu.	Multiply Accumulate High Halfword to Word Modulo Unsigned and Record		
machhwuo	Multiply Accumulate High Halfword to Word Modulo Unsigned with Overflow Enabled		
machhwuo.	Multiply Accumulate High Halfword to Word Modulo Unsigned with Overflow Enabled and Record		
maclhw	Multiply Accumulate Low Halfword to Word Modulo Signed		page 707
maclhw.	Multiply Accumulate Low Halfword to Word Modulo Signed and Record		
maclhwo	Multiply Accumulate Low Halfword to Word Modulo Signed with Overflow Enabled		
maclhwo.	Multiply Accumulate Low Halfword to Word Modulo Signed with Overflow Enabled and Record		
maclhws	Multiply Accumulate Low Halfword to Word Saturate Signed		page 708
maclhws.	Multiply Accumulate Low Halfword to Word Saturate Signed and Record		
maclhwso	Multiply Accumulate Low Halfword to Word Saturate Signed with Overflow Enabled		
maclhwso.	Multiply Accumulate Low Halfword to Word Saturate Signed with Overflow Enabled and Record		
maclhwsu	Multiply Accumulate Low Halfword to Word Saturate Unsigned		page 709
maclhwsu.	Multiply Accumulate Low Halfword to Word Saturate Unsigned and Record		
maclhwsuo	Multiply Accumulate Low Halfword to Word Saturate Unsigned with Overflow Enabled		
maclhwsuo.	Multiply Accumulate Low Halfword to Word Saturate Unsigned with Overflow Enabled and Record		
maclhwu	Multiply Accumulate Low Halfword to Word Modulo Unsigned		page 710
maclhwu.	Multiply Accumulate Low Halfword to Word Modulo Unsigned and Record		
maclhwuo	Multiply Accumulate Low Halfword to Word Modulo Unsigned with Overflow Enabled		
maclhwuo.	Multiply Accumulate Low Halfword to Word Modulo Unsigned with Overflow Enabled and Record		

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
mcrf	Move Condition Register Field		page 711
mcrxr	Move to Condition Register from XER		page 712
mfccr0	Move From Core-Configuration Register 0	mfspir	page 866
mfcrr	Move from Condition Register		page 713
mfctr	Move From Count Register	mfspir	page 866
mfdacl	Move From Data Address-Compare 1		
mfdacl2	Move From Data Address-Compare 2		
mfdbcr0	Move From Debug-Control Register 0		
mfdbcr1	Move From Debug-Control Register 1		
mfdbsr	Move From Debug-Status Register		
mfddcr	Move From Data-Cache Cachability Register		
mfddcr	Move from Device Control Register		page 714
mfddcrw	Move From Data-Cache Write-Through Register	mfspir	page 866
mfdear	Move From Data-Error Address Register		
mfddvc1	Move From Data Value-Compare 1		
mfddvc2	Move From Data Value-Compare 2		
mfesr	Move From Exception-Syndrome Register		
mfevpr	Move From Exception-Vector Prefix Register		
mfiacl	Move From Instruction Address-Compare 1		
mfiacl2	Move From Instruction Address-Compare 2		
mfiacl3	Move From Instruction Address-Compare 3		
mfiacl4	Move From Instruction Address-Compare 4		
mficcr	Move From Instruction-Cache Cachability Register		
mficdbdr	Move From Instruction-Cache Debug-Data Register		
mflr	Move From Link Register		
mfmsr	Move from Machine State Register		page 715
mfpid	Move From Process ID Register	mfspir	page 866
mfppit	Move From Programmable-Interval Timer		
mfppvr	Move From Processor-Version Register		
mfsgrr	Move From Storage Guarded Register		
mfslrr	Move From Storage Little-Endian Register		
mfspir	Move from Special Purpose Register		page 716

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
mfspg0	Move From SPR General-Purpose Register 0	mfsp	page 866
mfspg1	Move From SPR General-Purpose Register 1		
mfspg2	Move From SPR General-Purpose Register 2		
mfspg3	Move From SPR General-Purpose Register 3		
mfspg4	Move From SPR General-Purpose Register 4		
mfspg5	Move From SPR General-Purpose Register 5		
mfspg6	Move From SPR General-Purpose Register 6		
mfspg7	Move From SPR General-Purpose Register 7		
mfsrc0	Move From Save/Restore Register 0		
mfsrc1	Move From Save/Restore Register 1		
mfsrc2	Move From Save/Restore Register 2		
mfsrc3	Move From Save/Restore Register 3		
mfsrc0r	Move From Storage User-Defined 0 Register		
mftrb	Move from Time Base Register		page 717
mftrbl	Move From Time-Base Lower	mfsp	page 866
mftrbu	Move From Time-Base Upper		
mftrcr	Move From Timer-Control Register		
mftrsr	Move From Timer-Status Register		
mfusprg0	Move From User SPR General-Purpose Register 0		
mfxc	Move From Fixed-Point Exception Register		
mfzpr	Move From Zone-Protection Register		
mr	Move Register	or or.	page 870
mr.	Move Register and Record		
mtccr0	Move to Core-Configuration Register 0	mtspr	page 866
mtcr	Move to Condition Register	mtcrf	page 871
mtcrf	Move to Condition Register Fields		page 718
mtctr	Move to Count Register	mtspr	page 866
mtdac1	Move to Data Address-Compare 1		
mtdac2	Move to Data Address-Compare 2		
mtdbcr0	Move to Debug-Control Register 0		
mtdbcr1	Move to Debug-Control Register 1		
mtdbsr	Move to Debug-Status Register		
mtdccr	Move to Data-Cache Cachability Register		
mtdcr	Move to Device Control Register		page 720

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
mtdcwr	Move to Data-Cache Write-Through Register	mtspr	page 866
mtdear	Move to Data-Error Address Register		
mtdvc1	Move to Data Value-Compare 1		
mtdvc2	Move to Data Value-Compare 2		
mtesr	Move to Exception-Syndrome Register		
mtvpr	Move to Exception-Vector Prefix Register		
mtiac1	Move to Instruction Address-Compare 1		
mtiac2	Move to Instruction Address-Compare 2		
mtiac3	Move to Instruction Address-Compare 3		
mtiac4	Move to Instruction Address-Compare 4		
mticcr	Move to Instruction-Cache Cachability Register		
mtlr	Move to Link Register		
mtmsr	Move to Machine State Register		page 721
mtpid	Move to Process ID Register	mtspr	page 866
mtpit	Move to Programmable-Interval Timer		
mtsgr	Move to Storage Guarded Register		
mtsler	Move to Storage Little-Endian Register		
mtspr	Move to Special Purpose Register		page 722
mtsprg0	Move to SPR General-Purpose Register 0	mtspr	page 866
mtsprg1	Move to SPR General-Purpose Register 1		
mtsprg2	Move to SPR General-Purpose Register 2		
mtsprg3	Move to SPR General-Purpose Register 3		
mtsprg4	Move to SPR General-Purpose Register 4		
mtsprg5	Move to SPR General-Purpose Register 5		
mtsprg6	Move to SPR General-Purpose Register 6		
mtsprg7	Move to SPR General-Purpose Register 7		
mtsrr0	Move to Save/Restore Register 0		
mtsrr1	Move to Save/Restore Register 1		
mtsrr2	Move to Save/Restore Register 2		
mtsrr3	Move to Save/Restore Register 3		
mtsu0r	Move to Storage User-Defined 0 Register		
mttbl	Move to Time-Base Lower	mtspr	page 866
mttbu	Move to Time-Base Upper		

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
mttcr	Move to Timer-Control Register	mtspr	page 866
mttsr	Move to Timer-Status Register		
mtusprg0	Move to User SPR General-Purpose Register 0		
mtxer	Move to Fixed-Point Exception Register		
mtzpr	Move to Zone-Protection Register		
mulchw	Multiply Cross Halfword to Word Signed		page 723
mulchw.	Multiply Cross Halfword to Word Signed and Record		
mulchwu	Multiply Cross Halfword to Word Unsigned		page 724
mulchwu.	Multiply Cross Halfword to Word Unsigned and Record		
mulhhw	Multiply High Halfword to Word Signed		page 725
mulhhw.	Multiply High Halfword to Word Signed and Record		
mulhhwu	Multiply High Halfword to Word Unsigned		page 726
mulhhwu.	Multiply High Halfword to Word Unsigned and Record		
mulhw	Multiply High Word		page 727
mulhw.	Multiply High Word and Record		
mulhwu	Multiply High Word Unsigned		page 728
mulhwu.	Multiply High Word Unsigned and Record		
mullhw	Multiply Low Halfword to Word Signed		page 729
mullhw.	Multiply Low Halfword to Word Signed and Record		
mullhwu	Multiply Low Halfword to Word Unsigned		page 730
mullhwu.	Multiply Low Halfword to Word Unsigned and Record		
mulli	Multiply Low Immediate		page 731
mullw	Multiply Low Word		page 732
mullw.	Multiply Low Word and Record		
mullwo	Multiply Low Word with Overflow Enabled		page 732
mullwo.	Multiply Low Word with Overflow Enabled and Record		
nand	NAND		page 733
nand.	NAND and Record		
neg	Negate		page 734
neg.	Negate and Record		
nego	Negate with Overflow Enabled		
nego.	Negate with Overflow Enabled and Record		

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
nmacchw	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed		page 735
nmacchw.	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed and Record		
nmacchwso	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed with Overflow Enabled		
nmacchwso.	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed with Overflow Enabled and Record		
nmacchws	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed		page 736
nmacchws.	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed and Record		
nmacchwso	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed with Overflow Enabled		
nmacchwso.	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed with Overflow Enabled and Record		
nmachhw	Negative Multiply Accumulate High Halfword to Word Modulo Signed		page 737
nmachhw.	Negative Multiply Accumulate High Halfword to Word Modulo Signed and Record		
nmachhwo	Negative Multiply Accumulate High Halfword to Word Modulo Signed with Overflow Enabled		
nmachhwo.	Negative Multiply Accumulate High Halfword to Word Modulo Signed with Overflow Enabled and Record		
nmachhws	Negative Multiply Accumulate High Halfword to Word Saturate Signed		page 738
nmachhws.	Negative Multiply Accumulate High Halfword to Word Saturate Signed and Record		
nmachhwso	Negative Multiply Accumulate High Halfword to Word Saturate Signed with Overflow Enabled		
nmachhwso.	Negative Multiply Accumulate High Halfword to Word Saturate Signed with Overflow Enabled and Record		
nmaclhw	Negative Multiply Accumulate Low Halfword to Word Modulo Signed		page 739
nmaclhw.	Negative Multiply Accumulate Low Halfword to Word Modulo Signed and Record		
nmaclhwo	Negative Multiply Accumulate Low Halfword to Word Modulo Signed with Overflow Enabled		
nmaclhwo.	Negative Multiply Accumulate Low Halfword to Word Modulo Signed with Overflow Enabled and Record		
nmaclhws	Negative Multiply Accumulate Low Halfword to Word Saturate Signed		page 740
nmaclhws.	Negative Multiply Accumulate Low Halfword to Word Saturate Signed and Record		
nmaclhwso	Negative Multiply Accumulate Low Halfword to Word Saturate Signed with Overflow Enabled		
nmaclhwso.	Negative Multiply Accumulate Low Halfword to Word Saturate Signed with Overflow Enabled and Record		
nop	No operation	ori	page 870
nor	NOR		page 741
nor.	NOR and Record		
not	Complement (Not) Register	nor	page 871
not.	Complement (Not) Register and Record	nor.	

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
or	OR		page 742
or.	OR and Record		
orc	OR with Complement		page 743
orc.	OR with Complement and Record		
ori	OR Immediate		page 744
oris	OR Immediate Shifted		page 745
rfci	Return from Critical Interrupt		page 746
rfi	Return from Interrupt		page 747
rlwimi	Rotate Left Word Immediate then Mask Insert		page 748
rlwimi.	Rotate Left Word Immediate then Mask Insert and Record		
rlwinm	Rotate Left Word Immediate then AND with Mask		page 749
rlwinm.	Rotate Left Word Immediate then AND with Mask and Record		
rlwnm	Rotate Left Word then AND with Mask		page 750
rlwnm.	Rotate Left Word then AND with Mask and Record		
rotlw	Rotate Left	rlwinm	page 865
rotlw.	Rotate Left and Record	rlwinm.	
rotlwi	Rotate Left Immediate	rlwinm	
rotlwi.	Rotate Left Immediate and Record	rlwinm.	
rotrwi	Rotate Right Immediate	rlwinm	
rotrwi.	Rotate Right Immediate and Record	rlwinm.	
sc	System Call		page 751
slw	Shift Left Word		page 752
slw.	Shift Left Word and Record		
slwi	Shift Left Immediate	rlwinm	page 865
slwi.	Shift Left Immediate and Record	rlwinm.	
sraw	Shift Right Algebraic Word		page 753
sraw.	Shift Right Algebraic Word and Record		
srawi	Shift Right Algebraic Word Immediate		page 754
srawi.	Shift Right Algebraic Word Immediate and Record		
srw	Shift Right Word		page 755
srw.	Shift Right Word and Record		
srwi	Shift Right Immediate	rlwinm	page 865
srwi.	Shift Right Immediate and Record	rlwinm.	
stb	Store Byte		page 756
stbu	Store Byte with Update		page 757
stbux	Store Byte with Update Indexed		page 758

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
stbx	Store Byte Indexed		page 759
sth	Store Halfword		page 760
sthbrx	Store Halfword Byte-Reverse Indexed		page 761
sthu	Store Halfword with Update		page 762
sthux	Store Halfword with Update Indexed		page 763
sthx	Store Halfword Indexed		page 764
stmw	Store Multiple Word		page 765
stswi	Store String Word Immediate		page 766
stswx	Store String Word Indexed		page 768
stw	Store Word		page 770
stwbrx	Store Word Byte-Reverse Indexed		page 771
stwcx.	Store Word Conditional Indexed		page 773
stwu	Store Word with Update		page 775
stwux	Store Word with Update Indexed		page 776
stwx	Store Word Indexed		page 777
sub	Subtract	subf	page 868
sub.	Subtract and Record	subf.	
subc	Subtract Carrying	subfc	page 868
subc.	Subtract Carrying and Record	subfc.	
subco	Subtract Carrying with Overflow Enabled	subfco	page 868
subco.	Subtract Carrying with Overflow Enabled and Record	subfco.	
subf	Subtract from		page 778
subf.	Subtract from and Record		
subfc	Subtract from Carrying		page 779
subfc.	Subtract from Carrying and Record		
subfco	Subtract from Carrying with Overflow Enabled		
subfco.	Subtract from Carrying with Overflow Enabled and Record		
subfe	Subtract from Extended		page 780
subfe.	Subtract from Extended and Record		
subfeo	Subtract from Extended with Overflow Enabled		
subfeo.	Subtract from Extended with Overflow Enabled and Record		
subfic	Subtract from Immediate Carrying		page 781
subfme	Subtract from Minus One Extended		page 782
subfme.	Subtract from Minus One Extended and Record		
subfmeo	Subtract from Minus One Extended with Overflow Enabled		
subfmeo.	Subtract from Minus One Extended with Overflow Enabled and Record		

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
subfo	Subtract from with Overflow Enabled		page 778
subfo.	Subtract from with Overflow Enabled and Record		
subfze	Subtract from Zero Extended		page 783
subfze.	Subtract from Zero Extended and Record		
subfzeo	Subtract from Zero Extended with Overflow Enabled		
subfzeo.	Subtract from Zero Extended with Overflow Enabled and Record		
subi	Subtract Immediate	addi	page 868
subic	Subtract Immediate Carrying	addic	
subic.	Subtract Immediate Carrying and Record	addic.	
subis	Subtract Immediate Shifted	addis	
subo	Subtract with Overflow Enabled	subfo	
subo.	Subtract with Overflow Enabled and Record	subfo.	
sync	Synchronize		page 784
tlbia	TLB Invalidate All		page 785
tlbre	TLB Read Entry		page 786
tlbrehi	Read TLBHI Portion of TLB Entry	tlbre	page 868
tlbrelo	Read TLBLO Portion of TLB Entry		
tlbsx	TLB Search Indexed		page 788
tlbsx.	TLB Search Indexed and Record		
tlbsync	TLB Synchronize		page 790
tlbwe	TLB Write Entry		page 791
tlbwehi	Write TLBHI Portion of TLB Entry	tlbwe	page 868
tlbwelo	Write TLBLO Portion of TLB Entry		
trap	Trap if Unconditional	tw	page 869
tw	Trap Word		page 793
tweq	Trap if Equal	tw	page 869
tweqi	Trap if Equal Immediate	twi	
twge	Trap if Greater Than or Equal	tw	
twgei	Trap if Greater Than or Equal Immediate	twi	
twgt	Trap if Greater Than	tw	
twgti	Trap if Greater Than Immediate	twi	
twi	Trap Word Immediate		page 795

Table B-33: Complete List of Instruction Mnemonics (Continued)

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
twle	Trap if Less Than or Equal	tw	page 869
twlei	Trap if Less Than or Equal Immediate	twi	
twlge	Trap if Logically Greater Than or Equal	tw	
twlgei	Trap if Logically Greater Than or Equal Immediate	twi	
twlgt	Trap if Logically Greater Than	tw	
twlgti	Trap if Logically Greater Than Immediate	twi	
twlle	Trap if Logically Less Than or Equal	tw	
twllei	Trap if Logically Less Than or Equal Immediate	twi	
twllt	Trap if Logically Less Than	tw	
twllti	Trap if Logically Less Than Immediate	twi	
twlng	Trap if Logically Not Greater Than	tw	
twlngi	Trap if Logically Not Greater Than Immediate	twi	
twlnl	Trap if Logically Not Less Than	tw	
twlnli	Trap if Logically Not Less Than Immediate	twi	
twlt	Trap if Less Than	tw	
twlti	Trap if Less Than Immediate	twi	
twne	Trap if Not Equal	tw	
twnei	Trap if Not Equal Immediate	twi	
twng	Trap if Not Greater Than	tw	
twngi	Trap if Not Greater Than Immediate	twi	
twnl	Trap if Not Less Than	tw	
twnli	Trap if Not Less Than Immediate	twi	
wrttee	Write External Enable		page 797
wrtteei	Write External Enable Immediate		page 798
xor	XOR		page 799
xor.	XOR and Record		
xori	XOR Immediate		page 800
xoris	XOR Immediate Shifted		page 801

Simplified Mnemonics

Simplified mnemonics (sometimes referred to as extended mnemonics) define a shorthand used by assemblers for the most-frequently used forms of several instructions.

Branch Instructions

Two classes of simplified branch mnemonics are provided. [Table C-2, page 858](#) summarizes the simplified branch-conditional mnemonics that test if a condition is true or false. The condition tested can include a specific bit (*b*) in the CR, whether or not the contents of the CTR are zero, or both. [Table C-8, page 861](#) summarizes the simplified branch-conditional mnemonics that test a comparison condition. Instructions in that table specify a CR*n* field (*n*) that is checked for a particular comparison result.

True/False Conditional Branches

True/false conditional branches test a condition and branch if the condition is met. The condition tested can include a specific bit (*b*) in the CR, whether or not the contents of the CTR are zero, or both. The simplified mnemonics in [Table C-2](#) through [Table C-6](#) are formed using the following syntax (angle brackets denote an optional field):

b<CTR decrement><CTR test><CR test><LR target><CTR target><LR update><absolute target>

[Table C-1](#) shows the abbreviations used in the formation of the simplified branch mnemonics.

Table C-1: Abbreviations for True/False Conditional Branches

Abbreviation	Description	Mnemonic Field
d	Decrement CTR	CTR decrement
nz	Branch if CTR $\neq 0$	CTR test
z	Branch if CTR = 0	CTR test
f	Branch if condition false ($CR_b=0$)	CR test
t	Branch if condition true ($CR_b=1$)	CR test
lr	Branch to target address in LR	LR target
ctr	Branch to target address in CTR	CTR target
l	Update LR with return address (LK opcode field = 1)	LR update
a	Branch to absolute address (AA opcode field = 1)	absolute target

The detailed instruction syntax for the simplified mnemonics listed in [Table C-2](#) are shown in [Table C-3](#) through [Table C-6](#). A cross-reference to the appropriate table is shown in the column heading of [Table C-2](#).

Table C-2: Simplified Branch-Conditional Mnemonics, True/False Conditions

Operation	LR not Updated				LR Updated			
	Relative	Absolute	to LR	to CTR	Relative	Absolute	to LR	to CTR
	Table C-3		Table C-4		Table C-5		Table C-6	
Branch Unconditionally	—	—	blr	bctr	—	—	blr	bctrl
Branch if Condition True ($CR_b=1$)	bt	bta	btlr	btctr	btl	bta	btlr	btctrl
Branch if Condition False ($CR_b=0$)	bf	bfa	bflr	bfctr	bfl	bfa	bflr	bfctrl
Decrement CTR, Branch if CTR $\neq 0$	bdnz	bdnza	bdnzlr	—	bdnzl	bdnzla	bdnzlr	—
Decrement CTR, Branch if CTR $\neq 0$ and Condition True ($CR_b=1$)	bdnzt	bdnzta	bdnztlr	—	bdnztl	bdnzta	bdnztlr	—
Decrement CTR, Branch if CTR $\neq 0$ and Condition False ($CR_b=0$)	bdnzf	bdnzfa	bdnzflr	—	bdnzfl	bdnzfa	bdnzflr	—
Decrement CTR, Branch if CTR = 0	bdz	bdza	bdzlr	—	bdzl	bdzla	bdzlr	—
Decrement CTR, Branch if CTR = 0 and Condition True ($CR_b=1$)	bdzt	bdzta	bdztlr	—	bdztl	bdzta	bdztlr	—
Decrement CTR, Branch if CTR = 0 and Condition False ($CR_b=0$)	bdzf	bdzfa	bdzflr	—	bdzfl	bdzfa	bdzflr	—

Table C-3 lists the simplified-mnemonic assembler syntax for the branch-conditional relative and branch-conditional absolute instructions (true/false conditions) that do not update the LR. In the following table, *target* represents the target address of the branch.

Table C-3: Branch (True/False) to Relative/Absolute (LK=0)

Operation	LR not Updated			
	Branch Relative		Branch Absolute	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Branch Unconditionally	—	—	—	—
Branch if Condition True ($CR_b=1$)	bt <i>b</i> , target	bc 12, <i>b</i> , target	bta <i>b</i> , target	bca 12, <i>b</i> , target
Branch if Condition False ($CR_b=0$)	bf <i>b</i> , target	bc 4, <i>b</i> , target	bfa <i>b</i> , target	bca 4, <i>b</i> , target
Decrement CTR, Branch if CTR $\neq 0$	bdnz target	bc 16, 0, target	bdnza target	bca 16, 0, target
Decrement CTR, Branch if CTR $\neq 0$ and Condition True ($CR_b=1$)	bdnzt <i>b</i> , target	bc 8, <i>b</i> , target	bdnzta <i>b</i> , target	bca 8, <i>b</i> , target
Decrement CTR, Branch if CTR $\neq 0$ and Condition False ($CR_b=0$)	bdnzf <i>b</i> , target	bc 0, <i>b</i> , target	bdnzfa <i>b</i> , target	bca 0, <i>b</i> , target
Decrement CTR, Branch if CTR = 0	bdz target	bc 18, 0, target	bdza target	bca 18, 0, target
Decrement CTR, Branch if CTR = 0 and Condition True ($CR_b=1$)	bdzt <i>b</i> , target	bc 10, <i>b</i> , target	bdzta <i>b</i> , target	bca 10, <i>b</i> , target
Decrement CTR, Branch if CTR = 0 and Condition False ($CR_b=0$)	bdzf <i>b</i> , target	bc 2, <i>b</i> , target	bdzfa <i>b</i> , target	bca 2, <i>b</i> , target

Table C-4 lists the simplified-mnemonic assembler syntax for the branch-conditional to LR and branch-conditional to CTR instructions (true/false conditions) that do not update the LR.

Table C-4: Branch (True/False) to LR/CTR (LK=0)

Operation	LR not Updated			
	Branch to LR		Branch to CTR	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Branch Unconditionally	blr	bclr 20, 0	bctr	bcctr 20, 0
Branch if Condition True ($CR_b=1$)	btlr <i>b</i>	bclr 12, <i>b</i>	btctr <i>b</i>	bcctr 12, <i>b</i>
Branch if Condition False ($CR_b=0$)	bflr <i>b</i>	bclr 4, <i>b</i>	bfctr <i>b</i>	bcctr 4, <i>b</i>
Decrement CTR, Branch if CTR $\neq 0$	bdnzlr	bclr 16, 0	—	—
Decrement CTR, Branch if CTR $\neq 0$ and Condition True ($CR_b=1$)	bdnztlr <i>b</i>	bclr 8, <i>b</i>	—	—
Decrement CTR, Branch if CTR $\neq 0$ and Condition False ($CR_b=0$)	bdnzflr <i>b</i>	bclr 0, <i>b</i>	—	—
Decrement CTR, Branch if CTR = 0	bdzlr	bclr 18, 0	—	—
Decrement CTR, Branch if CTR = 0 and Condition True ($CR_b=1$)	bdztlr <i>b</i>	bclr 10, <i>b</i>	—	—
Decrement CTR, Branch if CTR = 0 and Condition False ($CR_b=0$)	bdzflr <i>b</i>	bclr 2, <i>b</i>	—	—

Table C-5 lists the simplified-mnemonic assembler syntax for the branch-conditional relative and branch-conditional absolute instructions (true/false conditions) that update the LR. In the following table, *target* represents the target address of the branch.

Table C-5: Branch (True/False) to Relative/Absolute (LK=1)

Operation	LR Updated			
	Branch Relative		Branch Absolute	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Branch Unconditionally	—	—	—	—
Branch if Condition True ($CR_b=1$)	btl <i>b</i>, target	bcl 12, <i>b</i>, target	btla <i>b</i>, target	bcla 12, <i>b</i>, target
Branch if Condition False ($CR_b=0$)	bfl <i>b</i>, target	bcl 4, <i>b</i>, target	bfla <i>b</i>, target	bcla 4, <i>b</i>, target
Decrement CTR, Branch if CTR $\neq 0$	bdnzl target	bcl 16, 0, target	bdnzla target	bcla 16, 0, target
Decrement CTR, Branch if CTR $\neq 0$ and Condition True ($CR_b=1$)	bdnztl <i>b</i>, target	bcl 8, <i>b</i>, target	bdnzla <i>b</i>, target	bcla 8, <i>b</i>, target
Decrement CTR, Branch if CTR $\neq 0$ and Condition False ($CR_b=0$)	bdnzfl <i>b</i>, target	bcl 0, <i>b</i>, target	bdnzfla <i>b</i>, target	bcla 0, <i>b</i>, target
Decrement CTR, Branch if CTR = 0	bdzl target	bcl 18, 0, target	bdzla target	bcla 18, 0, target
Decrement CTR, Branch if CTR = 0 and Condition True ($CR_b=1$)	bdztl <i>b</i>, target	bcl 10, <i>b</i>, target	bdzla <i>b</i>, target	bcla 10, <i>b</i>, target

Table C-5: Branch (True/False) to Relative/Absolute (LK=1) (Continued)

Operation	LR Updated			
	Branch Relative		Branch Absolute	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Decrement CTR, Branch if CTR = 0 and Condition False (CR _b =0)	bdzfl <i>b</i> , target	bcl 2, <i>b</i> , target	bdzfla <i>b</i> , target	bcla 2, <i>b</i> , target

Table C-6 lists the simplified-mnemonic assembler syntax for the branch-conditional to LR and branch-conditional to CTR instructions (true/false conditions) that update the LR.

Table C-6: Branch (True/False) to LR/CTR (LK=1)

Operation	LR Updated			
	Branch to LR		Branch to CTR	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Branch Unconditionally	blrl	bclrl 20, 0	bctrl	bcctrl 20, 0
Branch if Condition True (CR _b =1)	btirl <i>b</i>	bclrl 12, <i>b</i>	btctrl <i>b</i>	bcctrl 12, <i>b</i>
Branch if Condition False (CR _b =0)	bfirl <i>b</i>	bclrl 4, <i>b</i>	bfctrl <i>b</i>	bcctrl 4, <i>b</i>
Decrement CTR, Branch if CTR ≠ 0	bdnzlrl	bclrl 16, 0	—	—
Decrement CTR, Branch if CTR ≠ 0 and Condition True (CR _b =1)	bdnztlrl <i>b</i>	bclrl 8, <i>b</i>	—	—
Decrement CTR, Branch if CTR ≠ 0 and Condition False (CR _b =0)	bdnzflrl <i>b</i>	bclrl 0, <i>b</i>	—	—
Decrement CTR, Branch if CTR = 0	bdzlrl	bclrl 18, 0	—	—
Decrement CTR, Branch if CTR = 0 and Condition True (CR _b =1)	bdztlrl <i>b</i>	bclrl 10, <i>b</i>	—	—
Decrement CTR, Branch if CTR = 0 and Condition False (CR _b =0)	bdzflrl <i>b</i>	bclrl 2, <i>b</i>	—	—

Comparison Conditional Branches

Comparison conditional branches examine the specified field in the CR register and branch if the comparison outcome is met. The CR field can be omitted from the assembler syntax if the CR0 field is used. The simplified mnemonics in Table C-8 through Table C-12 are formed using the following syntax (angle brackets denote an optional field):

b<comparison><LR target><CTR target><LR update><absolute target>

Table C-7 shows the abbreviations for the comparison operations used in the formation of the simplified branch mnemonics. The remaining fields are abbreviated as shown in Table C-1, page 857.

Table C-7: Abbreviations for Comparison Conditional Branches

Abbreviation	Description
lt	Less than
le	Less than or equal
e	Equal

Table C-7: Abbreviations for Comparison Conditional Branches (Continued)

Abbreviation	Description
ge	Greater than or equal
gt	Greater than
nl	Not less than
ne	Not equal
ng	Not greater than
so	Summary overflow
ns	Not summary overflow

Table C-8 summarizes the simplified branch-conditional mnemonics that test a comparison condition. Instructions in that table specify a CR n field (n) that is checked for a particular comparison result. The CR field defaults to CR0 if omitted. The detailed instruction syntax for the simplified mnemonics listed in Table C-8 are shown in Table C-9 through Table C-12. A cross-reference to the appropriate table is shown in the column heading of Table C-8.

Table C-8: Simplified Branch-Conditional Mnemonics, Comparison Conditions

Operation	LR not Updated				LR Updated			
	Relative	Absolute	to LR	to CTR	Relative	Absolute	to LR	to CTR
	Table C-9		Table C-10		Table C-11		Table C-12	
Branch if Less Than	blt	blta	bltlr	bltctr	bltl	bltla	bltlrl	bltctrl
Branch if Less Than or Equal	ble	blea	blelr	blectr	blel	blela	blelrl	blectrl
Branch if Equal	beq	beqa	beqlr	beqctr	beql	beqla	beqlrl	beqctrl
Branch if Greater Than or Equal	bge	bgea	bgehr	bgectr	bgehl	bgeha	bgehlrl	bgectrl
Branch if Greater Than	bgt	bgtla	bgtlr	bgtctr	bgtl	bgtla	bgtlrl	bgtctrl
Branch if Not Less Than	bnl	bnla	bnllr	bnlctr	bnll	bnlla	bnllrl	bnlctrl
Branch if Not Equal	bne	bnea	bnelr	bnecr	bnel	bnela	bnelrl	bnecr
Branch if Not Greater Than	bng	bnga	bnglr	bngctr	bngl	bngla	bnglrl	bngctrl
Branch if Summary Overflow	bso	bsoa	bsolr	bsoctr	bsol	bsola	bsolrl	bsoctr
Branch if Not Summary Overflow	bns	bnsa	bnslr	bnsctr	bnsl	bnsla	bnsrl	bnsctrl

Table C-9 lists the simplified-mnemonic assembler syntax for the branch-conditional relative and branch-conditional absolute instructions (comparison conditions) that do not update the LR. In the following table, *target* represents the target address of the branch.

Table C-9: Branch (Comparison) to Relative/Absolute (LK=0)

Operation	LR not Updated			
	Branch Relative		Branch Absolute	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Branch if Less Than	blt n , target	bc 12, $4 \times n + 0$, target	blta n , target	bca 12, $4 \times n + 0$, target
Branch if Less Than or Equal	ble n , target	bc 4, $4 \times n + 1$, target	blea n , target	bca 4, $4 \times n + 1$, target
Branch if Equal	beq n , target	bc 12, $4 \times n + 2$, target	beqa n , target	bca 12, $4 \times n + 2$, target

Table C-9: Branch (Comparison) to Relative/Absolute (LK=0) (Continued)

Operation	LR not Updated			
	Branch Relative		Branch Absolute	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Branch if Greater Than or Equal	bge <i>n</i> , target	bc 4, 4× <i>n</i> +0, target	bgea <i>n</i> , target	bca 4, 4× <i>n</i> +0, target
Branch if Greater Than	bgt <i>n</i> , target	bc 12, 4× <i>n</i> +1, target	bgt a <i>n</i> , target	bca 12, 4× <i>n</i> +1, target
Branch if Not Less Than	bnl <i>n</i> , target	bc 4, 4× <i>n</i> +0, target	bnla <i>n</i> , target	bca 4, 4× <i>n</i> +0, target
Branch if Not Equal	bne <i>n</i> , target	bc 4, 4× <i>n</i> +2, target	bnea <i>n</i> , target	bca 4, 4× <i>n</i> +2, target
Branch if Not Greater Than	bng <i>n</i> , target	bc 4, 4× <i>n</i> +1, target	bnga <i>n</i> , target	bca 4, 4× <i>n</i> +1, target
Branch if Summary Overflow	bso <i>n</i> , target	bc 12, 4× <i>n</i> +3, target	bsoa <i>n</i> , target	bca 12, 4× <i>n</i> +3, target
Branch if Not Summary Overflow	bns <i>n</i> , target	bc 4, 4× <i>n</i> +3, target	bnsa <i>n</i> , target	bca 4, 4× <i>n</i> +3, target

Table C-10 lists the simplified-mnemonic assembler syntax for the branch-conditional to LR and branch-conditional to CTR instructions (comparison conditions) that do not update the LR.

Table C-10: Branch (Comparison) to LR/CTR (LK=0)

Operation	LR not Updated			
	Branch to LR		Branch to CTR	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Branch if Less Than	bltlr <i>n</i>	bclr 12, 4× <i>n</i> +0	bltctr <i>n</i>	bcctr 12, 4× <i>n</i> +0
Branch if Less Than or Equal	blelr <i>n</i>	bclr 4, 4× <i>n</i> +1	blectr <i>n</i>	bcctr 4, 4× <i>n</i> +1
Branch if Equal	beqlr <i>n</i>	bclr 12, 4× <i>n</i> +2	beqctr <i>n</i>	bcctr 12, 4× <i>n</i> +2
Branch if Greater Than or Equal	bgehr <i>n</i>	bclr 4, 4× <i>n</i> +0	bgectr <i>n</i>	bcctr 4, 4× <i>n</i> +0
Branch if Greater Than	bgtlr <i>n</i>	bclr 12, 4× <i>n</i> +1	bgtctr <i>n</i>	bcctr 12, 4× <i>n</i> +1
Branch if Not Less Than	bnllr <i>n</i>	bclr 4, 4× <i>n</i> +0	bnlctr <i>n</i>	bcctr 4, 4× <i>n</i> +0
Branch if Not Equal	bnelr <i>n</i>	bclr 4, 4× <i>n</i> +2	bnctr <i>n</i>	bcctr 4, 4× <i>n</i> +2
Branch if Not Greater Than	bnglr <i>n</i>	bclr 4, 4× <i>n</i> +1	bngctr <i>n</i>	bcctr 4, 4× <i>n</i> +1
Branch if Summary Overflow	bsolr <i>n</i>	bclr 12, 4× <i>n</i> +3	bsocctr <i>n</i>	bcctr 12, 4× <i>n</i> +3
Branch if Not Summary Overflow	bnslr <i>n</i>	bclr 4, 4× <i>n</i> +3	bnsctr <i>n</i>	bcctr 4, 4× <i>n</i> +3

Table C-11 lists the simplified-mnemonic assembler syntax for the branch-conditional relative and branch-conditional absolute instructions (comparison conditions) that update the LR. In the following table, *target* represents the target address of the branch.

Table C-11: Branch (Comparison) to Relative/Absolute (LK=1)

Operation	LR Updated			
	Branch Relative		Branch Absolute	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Branch if Less Than	bltl <i>n</i> , target	bcl 12, 4× <i>n</i> +0, target	bltla <i>n</i> , target	bcla 12, 4× <i>n</i> +0, target
Branch if Less Than or Equal	blel <i>n</i> , target	bcl 4, 4× <i>n</i> +1, target	blela <i>n</i> , target	bcla 4, 4× <i>n</i> +1, target

Table C-11: Branch (Comparison) to Relative/Absolute (LK=1) (Continued)

Operation	LR Updated			
	Branch Relative		Branch Absolute	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Branch if Equal	beql <i>n</i> , target	bcl 12, 4× <i>n</i> +2, target	beqla <i>n</i> , target	bcla 12, 4× <i>n</i> +2, target
Branch if Greater Than or Equal	bgel <i>n</i> , target	bcl 4, 4× <i>n</i> +0, target	bgela <i>n</i> , target	bcla 4, 4× <i>n</i> +0, target
Branch if Greater Than	bgtl <i>n</i> , target	bcl 12, 4× <i>n</i> +1, target	bgtla <i>n</i> , target	bcla 12, 4× <i>n</i> +1, target
Branch if Not Less Than	bnll <i>n</i> , target	bcl 4, 4× <i>n</i> +0, target	bnlla <i>n</i> , target	bcla 4, 4× <i>n</i> +0, target
Branch if Not Equal	bnel <i>n</i> , target	bcl 4, 4× <i>n</i> +2, target	bnela <i>n</i> , target	bcla 4, 4× <i>n</i> +2, target
Branch if Not Greater Than	bngr <i>n</i> , target	bcl 4, 4× <i>n</i> +1, target	bngra <i>n</i> , target	bcla 4, 4× <i>n</i> +1, target
Branch if Summary Overflow	bsol <i>n</i> , target	bcl 12, 4× <i>n</i> +3, target	bsola <i>n</i> , target	bcla 12, 4× <i>n</i> +3, target
Branch if Not Summary Overflow	bsnl <i>n</i> , target	bcl 4, 4× <i>n</i> +3, target	bsnla <i>n</i> , target	bcla 4, 4× <i>n</i> +3, target

Table C-12 lists the simplified-mnemonic assembler syntax for the branch-conditional to LR and branch-conditional to CTR instructions (comparison conditions) that update the LR.

Table C-12: Branch (Comparison) to LR/CTR (LK=1)

Operation	LR Updated			
	Branch to LR		Branch to CTR	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Branch if Less Than	bltlr <i>n</i>	bclr 12, 4× <i>n</i> +0	bltctr <i>n</i>	bcctr 12, 4× <i>n</i> +0
Branch if Less Than or Equal	blelr <i>n</i>	bclr 4, 4× <i>n</i> +1	blectr <i>n</i>	bcctr 4, 4× <i>n</i> +1
Branch if Equal	beqlr <i>n</i>	bclr 12, 4× <i>n</i> +2	beqctr <i>n</i>	bcctr 12, 4× <i>n</i> +2
Branch if Greater Than or Equal	bgelr <i>n</i>	bclr 4, 4× <i>n</i> +0	bgectr <i>n</i>	bcctr 4, 4× <i>n</i> +0
Branch if Greater Than	bgtlr <i>n</i>	bclr 12, 4× <i>n</i> +1	bgctr <i>n</i>	bcctr 12, 4× <i>n</i> +1
Branch if Not Less Than	bnllr <i>n</i>	bclr 4, 4× <i>n</i> +0	bnlctr <i>n</i>	bcctr 4, 4× <i>n</i> +0
Branch if Not Equal	bnelr <i>n</i>	bclr 4, 4× <i>n</i> +2	bnectr <i>n</i>	bcctr 4, 4× <i>n</i> +2
Branch if Not Greater Than	bngrl <i>n</i>	bclr 4, 4× <i>n</i> +1	bngrctr <i>n</i>	bcctr 4, 4× <i>n</i> +1
Branch if Summary Overflow	bsolr <i>n</i>	bclr 12, 4× <i>n</i> +3	bsoctr <i>n</i>	bcctr 12, 4× <i>n</i> +3
Branch if Not Summary Overflow	bsnlr <i>n</i>	bclr 4, 4× <i>n</i> +3	bsnctr <i>n</i>	bcctr 4, 4× <i>n</i> +3

Branch Prediction

The low-order bit (*y* bit) of the BO field in branch-conditional instructions provides a hint to the processor about whether the branch is likely to be taken. See **Specifying Branch-Prediction Behavior**, page 406 for more information on the *y* bit. Assemblers should clear this bit to 0 unless otherwise directed. Clearing the *y* bit specifies the following default action:

- A conditional branch with a negative displacement field is predicted taken.
- A conditional branch with a non-negative displacement field is predicted not taken (fall through).
- A conditional branch to an address in the LR or CTR is predicted not taken (fall through).

If the likely outcome (branch or fall through) of a conditional-branch instruction is known, a suffix can be added to the mnemonic that tells the assembler how to set the *y* bit, as follows:

- + indicates that the branch should be predicted taken.
- – indicates that the branch should be predicted not taken.

The suffix can be added to any branch-conditional mnemonic, including simplified mnemonics. For example, “**blt+** target” indicates the *branch to target if CR0 is less than* instruction should be predicted taken.

For relative and absolute branches, the default value of the *y* bit depends on whether the displacement field is negative or non-negative. With these instructions, the prediction override has the following effect:

- For negative displacement fields:
 - A “+” suffix clears the *y* bit to 0.
 - A “–” suffix sets the *y* bit to 1.
- For non-negative displacement fields:
 - A “+” suffix sets the *y* bit to 1.
 - A “–” suffix clears the *y* bit to 0.

For branches to an address in the LR or CTR, the prediction override has the following effect:

- A “+” suffix sets the *y* bit to 1.
- A “–” suffix clears the *y* bit to 0.

Compare Instructions

The PowerPC compare instructions include an L opcode field that specifies whether the comparison is performed on a word or doubleword operand. In 32-bit implementations like the PPC405, only word comparisons are supported. Simplified mnemonics are shown in **Table C-13** that dispense with the need to encode the L field in the instruction syntax.

The **crfD** field can be omitted if the comparison result is placed into the CR0 field. Otherwise, the target CR field must be specified as the first operand.

Table C-13: Simplified Mnemonics for Compare Instructions

Operation	Simplified Mnemonic	Equivalent Mnemonic
Compare Word Immediate	cmpwi crfD, rA, SIMM	cmpi crfD, 0, rA, SIMM
Compare Word	cmpw crfD, rA, rB	cmp crfD, 0, rA, rB
Compare Logical Word Immediate	cmplwi crfD, rA, UIMM	cmpli crfD, 0, rA, UIMM
Compare Logical Word	cmplw crfD, rA, rB	cmpl crfD, 0, rA, rB

CR-Logical Instructions

The condition register logical instructions, are used to set, clear, copy, or invert a specific condition register bit. The simplified mnemonics in **Table C-14** provide a shorthand for several common operations. The variables *bx* and *by* are used to specify individual CR bits.

Table C-14: Simplified Mnemonics for CR-Logical Instructions

Operation	Simplified Mnemonic	Equivalent Mnemonic
Condition Register Set	crset bx	creqv bx, bx, bx

Table C-14: Simplified Mnemonics for CR-Logical Instructions (Continued)

Operation	Simplified Mnemonic	Equivalent Mnemonic
Condition Register Clear	crclr <i>bx</i>	crxor <i>bx, bx, bx</i>
Condition Register Move	crmove <i>bx, by</i>	cror <i>bx, by, by</i>
Condition Register Not	crnot <i>bx, by</i>	crnor <i>bx, by, by</i>

Rotate and Shift Instructions

Although the rotate and shift instructions provide powerful and general ways to manipulate register contents, they can be difficult to understand. The simplified mnemonics in Table C-15 are provided for the following types of operations:

- *Extract*—Select a field of n bits starting at bit position b from the source register. Left or right justify this field in the target register. Clear all other bits of the target register.
- *Insert*—Select a left-justified or right-justified field of n bits from the source register. Insert this field in the target register starting at bit position b , leaving all other bits in the target register unchanged.
- *Rotate*—Rotate the contents of a register right or left by n bits without masking.
- *Shift*—Shift the contents of a register right or left by n bits, clearing vacated bits (logical shift).
- *Clear*—Clear the left-most or right-most n bits of a register.
- *Clear left and shift left*—Clear the left-most b bits of a register and shift the register left by n bits. This operation can be used to scale a known non-negative array index by the width of an element.

Table C-15: Simplified Mnemonics for Rotate and Shift Instructions

Operation	Simplified Mnemonic	Equivalent Mnemonic
Extract and Left Justify Immediate	extlwi <i>rA, rS, n, b (n > 0)</i>	rlwinm <i>rA, rS, b, 0, n-1</i>
	extlwi. <i>rA, rS, n, b (n > 0)</i>	rlwinm. <i>rA, rS, b, 0, n-1</i>
Extract and Right Justify Immediate	extrwi <i>rA, rS, n, b (n > 0)</i>	rlwinm <i>rA, rS, b+n, 32-n, 31</i>
	extrwi. <i>rA, rS, n, b (n > 0)</i>	rlwinm. <i>rA, rS, b+n, 32-n, 31</i>
Insert from Left Immediate	inslwi <i>rA, rS, n, b (n > 0)</i>	rlwimi <i>rA, rS, 32-b, b, (b+n)-1</i>
	inslwi. <i>rA, rS, n, b (n > 0)</i>	rlwimi. <i>rA, rS, 32-b, b, (b+n)-1</i>
Insert from Right Immediate	insrwi <i>rA, rS, n, b (n > 0)</i>	rlwimi <i>rA, rS, 32-(b+n), b, (b+n)-1</i>
	insrwi. <i>rA, rS, n, b (n > 0)</i>	rlwimi. <i>rA, rS, 32-(b+n), b, (b+n)-1</i>
Rotate Left Immediate	rotlwi <i>rA, rS, n</i>	rlwinm <i>rA, rS, n, 0, 31</i>
	rotlwi. <i>rA, rS, n</i>	rlwinm. <i>rA, rS, n, 0, 31</i>
Rotate Right Immediate	rotrwi <i>rA, rS, n</i>	rlwinm <i>rA, rS, 32-n, 0, 31</i>
	rotrwi. <i>rA, rS, n</i>	rlwinm. <i>rA, rS, 32-n, 0, 31</i>
Rotate Left	rotlw <i>rA, rS, rB</i>	rlwnm <i>rA, rS, rB, 0, 31</i>
	rotlw. <i>rA, rS, rB</i>	rlwnm. <i>rA, rS, rB, 0, 31</i>
Shift Left Immediate	slwi <i>rA, rS, n (n < 32)</i>	rlwinm <i>rA, rS, n, 0, 31-n</i>
	slwi. <i>rA, rS, n (n < 32)</i>	rlwinm. <i>rA, rS, n, 0, 31-n</i>
Shift Right Immediate	srwi <i>rA, rS, n (n < 32)</i>	rlwinm <i>rA, rS, 32-n, n, 31</i>
	srwi. <i>rA, rS, n (n < 32)</i>	rlwinm. <i>rA, rS, 32-n, n, 31</i>

Table C-15: Simplified Mnemonics for Rotate and Shift Instructions (Continued)

Operation	Simplified Mnemonic	Equivalent Mnemonic
Clear Left Immediate	clrlwi rA, rS, n ($n < 32$)	rlwinm rA, rS, 0, n, 31
	clrlwi. rA, rS, n ($n < 32$)	rlwinm. rA, rS, 0, n, 31
Clear Right Immediate	clrrwi rA, rS, n ($n < 32$)	rlwinm rA, rS, 0, 0, 31-n
	clrrwi. rA, rS, n ($n < 32$)	rlwinm. rA, rS, 0, 0, 31-n
Clear Left and Shift Left Immediate	clrlslwi rA, rS, b, n ($n \leq b \leq 31$)	rlwinm rA, rS, b-n, 31-n
	clrlslwi. rA, rS, b, n ($n \leq b \leq 31$)	rlwinm. rA, rS, b-n, 31-n

Special-Purpose Registers

Special-purpose register instructions use the SPR number (SPRN) to specify the register being read or written. The simplified mnemonics in Table C-16 encode the SPR name as part of the mnemonic rather than requiring a numeric SPRN operand.

Table C-16: Simplified Mnemonics for Special-Purpose Register Instructions

Special-Purpose Register	Move to SPR		Move from SPR	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Core-Configuration Register 0	mtccr0 rS	mtspr 947, rS	mfccr0 rD	mspr rD, 947
Count Register	mtctr rS	mtspr 9, rS	mfctr rD	mspr rD, 9
Data Address-Compare 1	mtdac1 rS	mtspr 1014, rS	mfdacl rD	mspr rD, 1014
Data Address-Compare 2	mtdac2 rS	mtspr 1015, rS	mfdacl rD	mspr rD, 1015
Debug-Control Register 0	mtdbc0 rS	mtspr 1010, rS	mfdbc0 rD	mspr rD, 1010
Debug-Control Register 1	mtdbc1 rS	mtspr 957, rS	mfdbc1 rD	mspr rD, 957
Debug-Status Register	mtdbsr rS ¹	mtspr 1008, rS ¹	mfdbsr rD	mspr rD, 1008
Data-Cache Cachability Register	mtdccr rS	mtspr 1018, rS	mfdccr rD	mspr rD, 1018
Data-Cache Write-Through Register	mtdcwr rS	mtspr 954, rS	mfdcwr rD	mspr rD, 954
Data-Error Address Register	mtdear rS	mtspr 981, rS	mfdear rD	mspr rD, 981
Data Value-Compare 1	mtdvc1 rS	mtspr 950, rS	mfsvc1 rD	mspr rD, 950
Data Value-Compare 2	mtdvc2 rS	mtspr 951, rS	mfsvc2 rD	mspr rD, 951
Exception-Syndrome Register	mtesr rS	mtspr 980, rS	mfesr rD	mspr rD, 980
Exception-Vector Prefix Register	mtcvpr rS	mtspr 982, rS	mfcvpr rD	mspr rD, 982
Instruction Address-Compare 1	mtiac1 rS	mtspr 1012, rS	mfiacl rD	mspr rD, 1012
Instruction Address-Compare 2	mtiac2 rS	mtspr 1013, rS	mfiacl rD	mspr rD, 1013
Instruction Address-Compare 3	mtiac3 rS	mtspr 948, rS	mfiacl rD	mspr rD, 948
Instruction Address-Compare 4	mtiac4 rS	mtspr 949, rS	mfiacl rD	mspr rD, 949
Instruction-Cache Cachability Register	mticcr rS	mtspr 1019, rS	mficcr rD	mspr rD, 1019
Instruction-Cache Debug-Data Register	—	—	mficdbdr rD	mspr rD, 979
Link Register	mtlir rS	mtspr 8, rS	mflr rD	mspr rD, 8

Notes:

1. Performs a clear to zero operation.

Table C-16: Simplified Mnemonics for Special-Purpose Register Instructions (Continued)

Special-Purpose Register	Move to SPR		Move from SPR	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Process ID Register	mtpid rS	mtspr 945, rS	mfpid rD	mfspir rD, 945
Programmable-Interval Timer	mtpit rS	mtspr 987, rS	mfpit rD	mfspir rD, 987
Processor-Version Register	—	—	mfpvr rD	mfspir rD, 287
Storage Guarded Register	mtsgr rS	mtspr 953, rS	mfsg rD	mfspir rD, 953
Storage Little-Endian Register	mtsler rS	mtspr 955, rS	mfsler rD	mfspir rD, 955
SPR General-Purpose Register 0	mtsprg0 rS	mtspr 272, rS	mfsprg0 rD	mfspir rD, 272
SPR General-Purpose Register 1	mtsprg1 rS	mtspr 273, rS	mfsprg1 rD	mfspir rD, 273
SPR General-Purpose Register 2	mtsprg2 rS	mtspr 274, rS	mfsprg2 rD	mfspir rD, 274
SPR General-Purpose Register 3	mtsprg3 rS	mtspr 275, rS	mfsprg3 rD	mfspir rD, 275
SPR General-Purpose Register 4	—	—	mfsprg4 rD	mfspir rD, 260
SPR General-Purpose Register 4	mtsprg4 rS	mtspr 276, rS	—	—
SPR General-Purpose Register 5	—	—	mfsprg5 rD	mfspir rD, 261
SPR General-Purpose Register 5	mtsprg5 rS	mtspr 277, rS	—	—
SPR General-Purpose Register 6	—	—	mfsprg6 rD	mfspir rD, 262
SPR General-Purpose Register 6	mtsprg6 rS	mtspr 278, rS	—	—
SPR General-Purpose Register 7	—	—	mfsprg7 rD	mfspir rD, 263
SPR General-Purpose Register 7	mtsprg7 rS	mtspr 279, rS	—	—
Save/Restore Register 0	mtsrr0 rS	mtspr 26, rS	mfsrr0 rD	mfspir rD, 26
Save/Restore Register 1	mtsrr1 rS	mtspr 27, rS	mfsrr1 rD	mfspir rD, 27
Save/Restore Register 2	mtsrr2 rS	mtspr 990, rS	mfsrr2 rD	mfspir rD, 990
Save/Restore Register 3	mtsrr3 rS	mtspr 991, rS	mfsrr3 rD	mfspir rD, 991
Storage User-Defined 0 Register	mtsu0r rS	mtspr 956, rS	mfsu0r rD	mfspir rD, 956
Time-Base Lower	mttbl rS	mtspr 284, rS	mftbl rD	mftb rD, 268
Time-Base Upper	mttbu rS	mtspr 285, rS	mftbu rD	mftb rD, 269
Timer-Control Register	mttcr rS	mtspr 986, rS	mftcr rD	mfspir rD, 986
Timer-Status Register	mttsr rS ¹	mtspr 984, rS ¹	mftsr rD	mfspir rD, 984
User SPR General-Purpose Register 0	mtusprg0 rS	mtspr 256, rS	mfsprg0 rD	mfspir rD, 256
Fixed-Point Exception Register	mtxer rS	mtspr 1, rS	mfxer rD	mfspir rD, 1
Zone-Protection Register	mtzpr rS	mtspr 944, rS	mfpzpr rD	mfspir rD, 944
Notes: 1. Performs a clear to zero operation.				

Subtract Instructions

The subtract-from instructions subtract the second operand (rA) from the third operand (rB). The simplified mnemonics in [Table C-17](#) use the order in which the third operand is subtracted from the second operand.

The effect of a subtract-immediate instruction can be achieved by using an add-immediate instruction with a negative immediate operand. In the following table, *value* represents a signed immediate operand.

Table C-17: Simplified Mnemonics for Subtract Instructions

Operation	Simplified Mnemonic	Equivalent Mnemonic
Subtract ($rA - rB$)	sub rD, rA, rB	subf rD, rB, rA
	sub. rD, rA, rB	subf. rD, rB, rA
	subo rD, rA, rB	subfo rD, rB, rA
	subo. rD, rA, rB	subfo. rD, rB, rA
Subtract Carrying ($rA - rB$)	subc rD, rA, rB	subfc rD, rB, rA
	subc. rD, rA, rB	subfc. rD, rB, rA
	subco rD, rA, rB	subfco rD, rB, rA
	subco. rD, rA, rB	subfco. rD, rB, rA
Subtract Immediate ($rA - \text{value}$)	subi rD, rA, value	addi rD, rA, -value
Subtract Immediate Shifted ($rA - \text{value} \ll 16_0$)	subis rD, rA, value	addis rD, rA, -value
Subtract Immediate Carrying ($rA - \text{value}$)	subic rD, rA, value	addic rD, rA, -value
Subtract Immediate Carrying and Record ($rA - \text{value}$)	subic. rD, rA, value	addic. rD, rA, -value

TLB-Management Instructions

The simplified mnemonics for TLB-management instructions are listed in [Table C-18](#).

Table C-18: Simplified Mnemonics for TLB-Management Instructions

Operation	Simplified Mnemonic	Equivalent Mnemonic
Read TLBHI Portion of TLB Entry	tlbrehi rD, rA	tlbre rD, rA, 0
Read TLBLO Portion of TLB Entry	tlbrelo rD, rA	tlbre rD, rA, 1
Write TLBHI Portion of TLB Entry	tlbwehi rD, rA	tlbwe rD, rA, 0
Write TLBLO Portion of TLB Entry	tlbwelo rD, rA	tlbwe rD, rA, 1

Trap Instructions

System-trap instructions use the TO opcode field to specify the trap condition. Simplified trap mnemonics are provided for the most common encodings of TO. These mnemonics encode the trap condition as part of the mnemonic rather than as a numeric operand. [Table C-19](#) shows the abbreviations for the comparison operations used in the formation of the simplified trap mnemonics. In this table, the column headed "<U" indicates an unsigned less-than comparison and the column headed ">U" indicates an unsigned greater-than comparison

Table C-19: Abbreviations for Trap Comparison Conditions

Abbreviation	Description	TO Encoding	<	>	=	<U	>U
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
—	Unconditional	31	1	1	1	1	1

Table C-20 lists the simplified mnemonics for the system-trap instructions.

Table C-20: Simplified Mnemonics for Trap Instructions

Operation	Trap Word		Trap Word Immediate	
	Simplified Mnemonic	Equivalent Mnemonic	Simplified Mnemonic	Equivalent Mnemonic
Trap if less than	twlt rA, rB	tw 16, rA, rB	twlti rA, SIMM	twi 16, rA, SIMM
Trap if less than or equal	twle rA, rB	tw 20, rA, rB	twlei rA, SIMM	twi 20, rA, SIMM
Trap if equal	tweq rA, rB	tw 4, rA, rB	tweqi rA, SIMM	twi 4, rA, SIMM
Trap if greater than or equal	twge rA, rB	tw 12, rA, rB	twgei rA, SIMM	twi 12, rA, SIMM
Trap if greater than	twgt rA, rB	tw 8, rA, rB	twgti rA, SIMM	twi 8, rA, SIMM
Trap if not less than	twnl rA, rB	tw 12, rA, rB	twnli rA, SIMM	twi 12, rA, SIMM
Trap if not equal	twne rA, rB	tw 24, rA, rB	twnei rA, SIMM	twi 24, rA, SIMM
Trap if not greater than	twng rA, rB	tw 20, rA, rB	twngi rA, SIMM	twi 20, rA, SIMM
Trap if logically less than	twllt rA, rB	tw 2, rA, rB	twllti rA, SIMM	twi 2, rA, SIMM
Trap if logically less than or equal	twlle rA, rB	tw 6, rA, rB	twllei rA, SIMM	twi 6, rA, SIMM
Trap if logically greater than or equal	twlge rA, rB	tw 5, rA, rB	twlgei rA, SIMM	twi 5, rA, SIMM
Trap if logically greater than	twlgt rA, rB	tw 1, rA, rB	twlgti rA, SIMM	twi 1, rA, SIMM
Trap if logically not less than	twlnl rA, rB	tw 5, rA, rB	twlnli rA, SIMM	twi 5, rA, SIMM
Trap if logically not greater than	twlng rA, rB	tw 6, rA, rB	twlngi rA, SIMM	twi 6, rA, SIMM
Trap if unconditional	trap	tw 31, rA, rB	—	twi 31, rA, SIMM

Other Simplified Mnemonics

No Operation

The preferred form of the no-operation instruction (no-op) is shown in Table C-21.

Table C-21: Simplified Mnemonic for No-op

Operation	Simplified Mnemonic	Equivalent Mnemonic
No operation	<code>nop</code>	<code>ori 0, 0, 0</code>

Load Immediate

The simplified mnemonics in Table C-22 provide a shorthand for loading an immediate signed value into a register.

Table C-22: Simplified Mnemonics for Load Immediate

Operation	Simplified Mnemonic	Equivalent Mnemonic
Load Immediate	<code>li rD, SIMM</code>	<code>addi rD, 0, SIMM</code>
Load Immediate Shifted	<code>lis rD, SIMM</code>	<code>addis rD, 0, SIMM</code>

Load Address

The load-address simplified mnemonic in Table C-23 computes the value of a base-displacement operand (register-indirect with immediate index addressing). This mnemonic is useful for obtaining the address of a variable specified by name. The assembler substitutes the name *variable* with the appropriate values of *rA* and *d* in the address syntax *d(rA)*.

Table C-23: Simplified Mnemonic for Load Address

Operation	Simplified Mnemonic	Equivalent Mnemonic
Load Address	<code>la rD, d(rA)</code>	<code>addi rD, rA, d</code>
	<code>la rD, variable</code>	<code>addi rD, rA, d</code> (<i>rA, d</i> substitution by assembler)

Move Register

The simplified mnemonics in Table C-24 provide a shorthand for moving the contents of a GPR to another GPR.

Table C-24: Simplified Mnemonics for Move Register

Operation	Simplified Mnemonic	Equivalent Mnemonic
Move Register	<code>mr rA, rS</code>	<code>or rA, rS, rS</code>
	<code>mr. rA, rS</code>	<code>or. rA, rS, rS</code>

Complement Register

The simplified mnemonics in Table C-25 provide a shorthand for complementing the contents of a GPR.

Table C-25: Simplified Mnemonics for Complement Register

Operation	Simplified Mnemonic	Equivalent Mnemonic
Complement (Not) Register	not rA, rS	nor rA, rS, rS
	not. rA, rS	nor. rA, rS, rS

Move to Condition Register

The simplified mnemonic in Table C-26 provides a shorthand for copying the contents of a GPR into the CR.

Table C-26: Simplified Mnemonic for Move to Condition Register

Operation	Simplified Mnemonic	Equivalent Mnemonic
Move to Condition Register	mtcr rS	mtcrf 0xFF, rS

Programming Considerations

This appendix provides programming examples that can be useful in embedded applications.

Synchronization Examples

The following provides general guidelines for using the **lwarx** and **stwcx** instructions:

- The **lwarx** and **stwcx** instructions should be paired and use the same effective address (EA).
- An unpaired **stwcx** instruction to an arbitrary EA (scratch address) can be used to clear any reservation held by the processor.
- An **lwarx** instruction can be left unpaired when executing certain synchronization primitives if the value loaded by the **lwarx** is not zero. **Test and Set**, page 874 provides such an example.
- Minimizing the looping on an **lwarx/stwcx** pair increases the likelihood that forward progress is made. The sequence shown in **Test and Set**, page 874 provides such an example. This example tests the old value before attempting the store. If the order is reversed (store before load), more **stwcx** instructions are executed and reservations are more likely to be lost between the **lwarx** and the **stwcx** instructions.
- Performance can be improved by minimizing looping on an **lwarx** instruction that fails to return a desired value. Performance can also be improved by using an ordinary load instruction to do the initial value check, as follows:


```

loop: lwz    r5,0(r3) #load the word
      cmpwi  r5,0     #compare word to 0
      bne-   loop     #loop back if word not equal to 0
      lwarx  r5,0,r3  #try reserving again
      cmpwi  r5,0     #compare likely to succeed
      bne    loop
      stwcx. r4,0,r3  #try to store nonzero
      bne-   loop     #loop if reservation lost
      
```
- Livelock is a state where no progress is made in a multiprocessor environment due to the interaction of the processors. Livelock is possible if a loop containing an **lwarx/stwcx** pair also contains an ordinary store instruction that affects one or more bytes in the reservation granule. For example, the first code sequence shown in **List Insertion**, page 876 can cause livelock if two list elements have next element pointers in the same reservation granule.

The examples in this appendix show how synchronization instructions are used to emulate various synchronization primitives and how more complex forms of synchronization can be implemented. Each example assumes that a similar instruction sequence is used by all processes requiring synchronization of the accessed data. The examples show a conditional sequence that begins with an **lwarx** instruction. This can be followed by memory accesses and/or computations on the loaded value. The sequence ends with a

stwcx. instruction. In most of the examples, failure of the **stwcx.** instruction causes a branch back to the **lwarx** for a repeated attempt. The examples are optimized for the case where the **stwcx.** instruction succeeds by having the conditional-branch prediction bit set appropriately.

Fetch and No-Op

The *fetch and no-op* primitive atomically loads the current value in a memory word. This example assumes that the address of the memory word is in **r3** and the data is loaded into **r4**.

```
loop: lwarx  r4,0,r3 #load and reserve
      stwcx. r4,0,r3 #store old value if still reserved
      bne-   loop   #loop if reservation lost
```

If the **stwcx.** succeeds, the destination location is updated with the same value that was loaded by the preceding **lwarx**. Although this store is unnecessary with respect to the value in the memory location, its success ensures that the value loaded by the **lwarx** was the most current value.

Fetch and Store

The *fetch and store* primitive atomically loads and replaces a memory word. This example assumes that the address of the memory word is in **r3**, the new data is stored from **r4**, and the old data is loaded into **r5**.

```
loop: lwarx  r5,0,r3 #load and reserve
      stwcx. r4,0,r3 #store new value if still reserved
      bne-   loop   #loop if reservation lost
```

Fetch and Add

The *fetch and add* primitive atomically increments a memory word. This example assumes that the incremented (new) data is stored from **r0**, the address of the memory word to be incremented is in **r3**, the increment value is contained in **r4**, and the data to be incremented is loaded into **r5**.

```
loop: lwarx  r5,0,r3 #load and reserve
      add    r0,r4,r5 #increment word
      stwcx. r0,0,r3 #store new value if still reserved
      bne-   loop   #loop if reservation lost
```

Fetch and AND

The *fetch and AND* primitive atomically ANDs a value into a memory word. This example assumes that the ANDed (new) data is stored from **r0**, the address of the memory word to be ANDed is in **r3**, the AND value is contained in **r4**, and the data to be ANDed is loaded into **r5**.

```
loop: lwarx  r5,0,r3 #load and reserve
      and    r0,r4,r5 #AND word
      stwcx. r0,0,r3 #store new value if still reserved
      bne-   loop   #loop if reservation lost
```

The above sequence can be changed to perform any atomic boolean operation on a memory word.

Test and Set

This version of the *test and set* primitive atomically loads a word from memory, ensures that the memory word is a nonzero value, and updates **CR0[EQ]** according to whether the value loaded is zero. This example assumes that the address of the memory word is in **r3**, the new (nonzero) data is stored from **r4**, and the old data is loaded into **r5**.

```

loop:  lwarx  r5,0,r3  #load and reserve
      cmpwi  r5, 0    #compare with 0
      bne    $+12     #branch if not equal to 0
      stwcx. r4,0,r3  #try to store non-zero
      bne-   loop     #loop if reservation lost

```

Compare and Swap

The *compare and swap* primitive atomically compares a value in a first register with a memory word. If they are equal, it stores a value from a second register into the memory word. If they are unequal, it moves the word from memory into the first register and updates CR0[EQ] to reflect the comparison result. This example assumes that the address of the memory word is in r3, the compare value is contained in r4, the new data is stored from r5, and the old data is loaded into r6.

```

loop:  lwarx  r6,0,r3  #load and reserve
      cmpw   r4,r6    #compare load value with first register
      bne-   exit     #skip if not equal
      stwcx. r5,0,r3  #store second register if still reserved
      bne-   loop     #loop if reservation lost
exit:  mr     r4,r6    #move load value into first register

```

The following applies to the above example:

- The semantics are based on the IBM System/370™ *compare and swap* instruction. Some architectures define this primitive differently.
- A *compare and swap* instruction is useful on machines that lack the synchronization capability provided by the **lwarx** and **stwcx.** instructions. Although such an instruction is atomic, it checks only whether the current value matches the old value. An error can occur if the value is changed and restored before being tested.
- In some applications, the second **bne-** instruction and/or the **mr** instruction can be omitted. The second **bne-** is used only to indicate that the original values in r4 and r6 were not equal by exiting the primitive with CR0[EQ]=0. If this indication is not required by the application, the second **bne-** can be omitted. The **mr** is used only when the application requires that the memory word be loaded into the compare register (rather than into a third register) if the compared values are not equal. The resulting compare and swap primitive does not obey the IBM System/370 semantics if either or both of these instructions are omitted.

Lock Acquisition and Release

This example provides a locking algorithm that demonstrates the use of an atomic read/modify/write synchronization operation. The argument of the lock and unlock procedures is the address of a shared memory location (stored in r3). This argument points to a lock that controls access to some shared resource, such as a data structure. The lock is open when its value is zero and it is locked when its value is one. Before accessing the shared resource, the processor sets the lock by having the lock procedure call **test_and_set** (the procedure executes the code sequence in **Test and Set**, page 874). This atomically updates the old value of the lock with the new value (1) contained in r4. The old value is returned in r5 (not shown in the following example). CR0[EQ] is updated by **test_and_set** to indicate whether the value returned in r5 is zero. The lock procedure repeats the **test_and_set** procedure until it successfully changes the lock value from zero to one.

The processor does not access the shared resource until it sets the lock. After the **bne** instruction checks for the successful test and set operation, the processor executes the **isync** instruction. This synchronizes program context. The **sync** instruction could be used but performance would be degraded because the **sync** instruction waits for all outstanding memory accesses to complete with respect to other processors. This is not required by the procedure.

```

lock:  li    r4,1          #obtain new lock
loop:  bl    test_and_set  #test and set
      bne-   loop         #retry until old lock = 0
      isync                #synchronize context
      blr                    #return

```

The unlock procedure writes a zero to the lock location. If access to the shared resource includes write operations, most applications require a **sync** instruction to make the shared resource modifications visible to all processors before releasing the lock.

```

unlock: sync                #delay until prior stores finish
      li    r1,0
      stw   r1,0(r3) #store zero to lock location
      blr                    #return

```

List Insertion

The following example shows how the **lwarx** and **stwcx** instructions are used to implement simple LIFO (last-in-first-out) insertion into a singly linked list. If multiple values must be changed atomically or the correct order of insertion depends on the element contents, insertion cannot be implemented as shown below and instead requires a more complicated strategy (such as lock synchronization).

In this example, list elements are data structures that contain pointers to the next element in the list. A new element is inserted after an existing (parent) element. The next element pointer in the parent element is copied (stored) unconditionally into the new element. A pointer to the new element is stored conditionally into the parent element.

In this example, it is assumed that the parent element address is in **r3**, the new element address is in **r4**, and the next element pointers are at offset zero in the respective element data structure. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements.

```

loop:  lwarx  r2,0,r3 #get next pointer
      stw    r2,0(r4) #store in new element
      sync                   #synchronize memory (can omit if not MP)
      stwcx. r4,0,r3 #add new element to list
      bne-   loop      #loop if reservation lost

```

In the preceding example, livelock can occur in a multiprocessor system if two list elements have next element pointers within the same reservation granule. If it is not possible to allocate list elements such that next element pointers are in different reservation granules, livelock can be avoided by using the following sequence:

```

      lwz    r2,0(r3) #get next pointer
loop1: mr    r5,r2     #keep a copy
      stw    r2,0(r4) #store in new element
      sync                   #synchronize memory
loop2: lwarx  r2,0,r3 #get next pointer again
      cmpw   r2,r5     #loop if changed
      bne-   loop1    #(updated by another processor)
      stwcx. r4,0,r3 #add new element to list
      bne-   loop2    #loop if reservation lost

```

Multiple-Precision Shifts

Following are programming examples for multiple-precision shifts. A multiple-precision shift is a shift of an n -word quantity, where $n > 1$. The quantity to be shifted is contained in n registers. The shift amount is specified either by an immediate value in the instruction or by bits 27:31 of a register.

The following examples distinguish between the cases $n = 2$ and $n > 2$. If $n > 2$, the examples yield the desired result only when the shift amount is restricted to the range 0–31. When $n > 2$, the number of instructions required is $2n - 1$ (immediate shifts) or $3n - 1$ (non-

immediate shifts). The examples shown for $n > 2$ use $n = 3$. Extending those examples to larger values of n or reducing them to the case $n = 2$ is straightforward when the shift amount restriction is met. This restriction is always met for shifts with immediate shift amounts.

The examples assume GPRs $r2$ and $r3$ (and $r4$ if $n = 3$) contain the quantity to be shifted and that the result is placed into the same registers. For non-immediate shifts, the shift amount is contained in bits 27:31 of GPR $r6$. For immediate shifts, the shift amount is assumed to be greater than zero. GPRs $r0$ and $r31$ are used as scratch registers. The variable sh represents the shift amount.

- Shift-left immediate, $n = 3$ (shift amount < 32)


```
rlwinm r2, r2, sh, 0, 31-sh
rlwimi r2, r3, sh, 32-sh, 31
rlwinm r3, r3, sh, 0, 31-sh
rlwimi r3, r4, sh, 32-sh, 31
rlwinm r4, r4, sh, 0, 31-sh
```
- Shift-left, $n = 2$ (shift amount < 64)


```
subfic r31, r6, 32
slw    r2, r2, r6
srw    r0, r3, r31
or     r2, r2, r0
addi   r31, r6, -32
slw    r0, r3, r31
or     r2, r2, r0
slw    r3, r3, r6
```
- Shift-left, $n = 3$ (shift amount < 32)


```
subfic r31, r6, 32
slw    r2, r2, r6
srw    r0, r3, r31
or     r2, r2, r0
slw    r3, r3, r6
srw    r0, r4, r31
or     r3, r3, r0
slw    r4, r4, r6
```
- Shift-right immediate, $n = 3$ (shift amount < 32)


```
rlwinm r4, r4, 32-sh, sh, 31
rlwimi r4, r3, 32-sh, 0, sh-1
rlwinm r3, r3, 32-sh, sh, 31
rlwimi r3, r2, 32-sh, 0, sh-1
rlwinm r2, r2, 32-sh, sh, 31
```
- Shift-right, $n = 2$ (shift amount < 64)


```
subfic r31, r6, 32
srw    r3, r3, r6
slw    r0, r2, r31
or     r3, r3, r0
addi   r31, r6, -32
srw    r0, r2, r31
or     r3, r3, r0
srw    r2, r2, r6
```
- Shift-right, $n = 3$ (shift amount < 32)


```
subfic r31, r6, -32
srw    r4, r4, r6
slw    r0, r3, r31
or     r4, r4, r0
srw    r3, r3, r6
slw    r0, r2, r31
or     r3, r3, r0
srw    r2, r2, r6
```
- Shift-right algebraic immediate, $n = 3$ (shift amount < 32)

- ```

 rlwinm r4, r4, 32-sh, sh, 31
 rlwimi r4, r3, 32-sh, 0, sh-1
 rlwinm r3, r3, 32-sh, sh, 31
 rlwimi r3, r2, 32-sh, 0, sh-1
 srawi r2, r2, sh

```
- Shift-right algebraic,  $n = 2$  (shift amount  $< 64$ )
 

```

 subfic r31, r6, 32
 srw r3, r3, r6
 slw r0, r2, r31
 or r3, r3, r0
 addic. r31, r6, -32
 sraw r0, r2, r31
 ble $+8
 ori r3, r0, 0
 sraw r2, r2, r6

```
  - Shift-right algebraic,  $n = 3$  (shift amount  $< 32$ )
 

```

 subfic r31, r6, 32
 srw r4, r4, r6
 slw r0, r3, r31
 or r4, r4, r0
 srw r3, r3, r6
 slw r0, r2, r31
 or r3, r3, r0
 sraw r2, r2, r6

```

## Code Optimization Guidelines

The following guidelines can help reduce program execution time in the PPC405. Additional information on PowerPC code optimization can be found in *The PowerPC Compiler Writer's Guide*.

### Conditional Branches

Multi-way branches and compound branches can be implemented in several ways. The implementation choice depends on problem specifics, including the number and distribution of test conditions and the instruction timings and latencies. Usually, the implementation involves a combination of conditional branches and unconditional branches.

Conditional branches require the evaluation of conditional expressions. In evaluating these expressions, performance can be improved by using instructions that update the CR to reflect their results. These results are represented in the CR as boolean variables that can be operated on using the CR-logical instructions. This usually yields better performance than using other instructions to evaluate conditional expressions solely in the GPRs.

The following pseudocode provides a simple example of how the CR register and CR-logical instructions can be used to improve the performance of conditional expressions by eliminating branches. In this example, Var28–Var31 are boolean variables maintained as bits in the CR[CR7] field (CR<sub>28:31</sub>). These variables represent a true condition by using the binary value 0b1 and a false condition by using the binary value 0b0.

```
if (Var28 || Var29 || Var30 || Var 31) branch to target
```

The above pseudocode can be implemented in assembler using branches as follows:

```

bt 28, target
bt 29, target
bt 30, target
bt 31, target

```

The following assembler sequence is functionally equivalent but replaces three of the branches with CR-logical instructions. The processor can usually execute these instructions faster than branches.

```
crr 2, 28, 29
crr 2, 2, 30
crr 2, 2, 31
bt 2, target
```

## Branch Prediction

If the outcome of a conditional branch is likely to contradict the default prediction used by the processor, software can override the default prediction by setting the *y* bit in the branch-instruction BO opcode field (see **Branch Prediction**, [page 406](#) for more information on the *y* bit). Overriding this default prediction is useful in the following situations:

- If an unlikely call to an error handler lies in the fall-through path.
- If program profiling determines that the default branch prediction is likely to be incorrect.
- If a conditional subroutine return is likely to be taken. Subroutine returns are normally programmed using branch to link register instructions which are predicted not taken by default.

## CR Dependencies

If an instruction updates the CR register and the result is used by a conditional branch, two instructions should be placed between the CR-update instruction and conditional branch. This gives the processor sufficient time to resolve the branch without stalling instruction execution due to a possibly incorrect branch prediction. The CR-update instructions that can benefit from this action are:

- Integer-arithmetic, compare, and logical instructions that have the Rc opcode field set.
- The **addic**., **andi**., and **andis**. instructions.
- CR-logical instructions.
- The **mcrf**, **mcrxr**, and **mtcrf** instructions.

## Floating-Point Emulation

The PPC405 is an integer processor and does not support the execution of floating-point instructions in hardware. System software can provide floating-point emulation support using one of two methods.

The preferred method is to supply a call interface to subroutines within a floating-point run-time library. The individual subroutines can emulate the operation of floating-point instructions. This method requires the recompilation of floating-point software in order to add the call interface and link in the library routines.

Alternatively, system software can use the program interrupt. Attempted execution of floating-point instructions on the PPC405 causes a program interrupt to occur due to an illegal instruction. The interrupt handler must be able to decode the illegal instruction and call the appropriate library routines to emulate the floating-point instruction using integer instructions. This method is not preferred due to the overhead associated with executing the interrupt handler. However, this method supports software containing PowerPC floating-point instructions without requiring recompilation. See **Program Interrupt (0x0700)**, [page 547](#), for more information.

## Cache Usage

Code and data can be accessed much faster if it is located in the processor caches instead of external memory. Code and data can be organized to minimize cache misses, reducing the need for external memory accesses.



Any two memory addresses are considered congruent if address bits 19:26 (the cache index) are the same but address bits 0:18 (the cache tag) are different. Address bits 27:31 define the 32-byte cacheline, which is the smallest object that can be brought into the cache. Only two congruent cachelines can be in the cache simultaneously. Accessing a third congruent line causes one of the two lines already in the cache to be removed.

Software can minimize the number of congruent addresses by organizing used addresses such that they are uniformly distributed across address bits 19:26.

## Alignment

Misaligned memory accesses are usually handled by the processor and do not cause an alignment exception. However, the fastest possible memory-access performance is obtained when operands are properly aligned. If an unaligned load or store operand crosses a word boundary, the processor accesses that operand using two memory references.

Branch targets should be aligned on a cache-line boundary if that target is unlikely to be accessed due to a default prediction or a prediction override. This helps minimize the number of unused instructions present in the instruction cache.

## Instruction Performance

The following performance descriptions consider only the “first order” effects of cache misses. The performance penalty associated with a cache miss involves a number of second-order effects. This includes PLB contention between the instruction and data caches and the time associated with performing cache-line fills and flushes. Unless stated otherwise, the number of cycles described applies to systems having zero-wait-state memory access.

## General Rules

The following rules apply to instruction execution in the PPC405:

- Instructions execute in order.
- Assuming cache hits, all instructions execute in one cycle except the following:
  - Divide instructions execute in 35 clock cycles.
  - Branches execute in one to three clock cycles as described in **Branches** below.
  - Multiply-accumulate and multiply instructions execute in one to five cycles as described in **Multiplies** below.
  - Aligned load/store instructions that hit in the data cache execute in one clock cycle. See **Alignment** above for information on the access penalty associated with unaligned load/stores.
- A data cache-control instruction requires two cycles to execute. However, subsequent data-cache accesses stall until the cache-control instruction finishes accessing the data cache. Those accesses do not remain stalled when transfers associated with previous data cache-control instructions continue on the PLB.

## Branches

The performance of a branch instruction depends on how quickly it is resolved. A branch is resolved when all conditions it depends on are known and the branch target is known. Generally, the greater the separation (in instructions) between a branch and the last instruction it depends on, the earlier the branch is resolved. If the branch is resolved early, it can be executed in fewer cycles.

The execution time of branches on the PPC405 can be determined as follows:

- A *known not taken* branch does not have condition dependencies (they are resolved) or

address dependencies (the next instruction is executed). These instructions execute in one clock cycle.

- A *known taken* branch does not have condition dependencies (they are resolved) but can have address dependencies. These instructions execute as follows:
  - When address dependencies are resolved, the instruction executes in one or two cycles depending on where the branch instruction is in the pipeline when the address is resolved. If the address is resolved early (at or before prefetch) it executes in one cycle. If the address is resolved during decode, it executes in two cycles.
  - When address dependencies are not resolved, the instruction executes in two or three cycles. This depends on the separation between the branch and the address-calculation instructions. If the separation is one instruction, the branch executes in two cycles. If there is no separation, the branch executes in three cycles.
- A *predicted not taken* branch has condition dependencies. These instructions execute as follows:
  - If the prediction is correct, the branch executes in one cycle.
  - If the prediction is incorrect, the instruction executes in two or three cycles. This depends on the separation between the branch and conditional instructions. If the separation is one instruction, the branch executes in two cycles. If there is no separation, the branch executes in three cycles.
- A *predicted taken* branch has condition dependencies. These instructions execute as follows:
  - If the prediction is correct, the branch executes in one or two cycles, depending on where the branch instruction is in the pipeline when the prediction occurs. If the instruction is predicted early (at or before prefetch) it executes in one cycle. If the instruction is predicted during decode, it executes in two cycles.
  - If the prediction is incorrect, the instruction executes in two or three cycles. This depends on the separation between the branch and the condition-setting instructions. If the separation is one instruction, the branch executes in two cycles. If there is no separation, the branch executes in three cycles.

## Multiplies

The PPC405 supports word multiplication and halfword multiplication. Multiply-accumulate (MAC) instructions are also supported. All of these instructions use the same multiplication hardware and are pipelined by the processor in the execution unit.

The time required by the processor to multiply two words depends on whether the first operand is larger than the second. The processor reduces the number of cycles required to perform a multiplication by automatically detecting which operand is smaller and internally ordering them appropriately. The operand size is determined by examining the number of bits involved in the sign-extension.

Issue-rate cycles and latency cycles are associated with the pipelining of multiply and MAC instructions, as shown in [Table D-1](#). Issue-rate cycles describe the number of cycles required between operations before the multiplication hardware can accept a new operation. Latency cycles describe the total number of cycles for the multiplication hardware to perform the operation.

Under the conditions described below, a second multiply or MAC instruction can begin execution before the first multiply or MAC instruction completes. When these conditions are met, the issue-rate cycle numbers apply. Otherwise, the latency cycle numbers apply. A multiply or MAC instruction can follow another multiply or MAC and still meet the conditions that support the use of the issue-rate cycle numbers.

Table D-1: Multiply and MAC Instruction Timing

| Operations                                                                                                                                                                                                                                                                                             | Issue-Rate Cycles | Latency Cycles |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------|----------------|
| MAC and Negative MAC                                                                                                                                                                                                                                                                                   | 1                 | 2              |
| Halfword × Halfword (32-bit result)                                                                                                                                                                                                                                                                    | 1                 | 2              |
| Halfword × Word (48-bit result)                                                                                                                                                                                                                                                                        | 2                 | 3              |
| Word × Word (64-bit result)                                                                                                                                                                                                                                                                            | 4                 | 5              |
| <b>Notes:</b><br>For the purposes of this table, words are treated as halfwords if the upper 16 bits of the operand contain a sign extension of the lower 16 bits. For example, if the upper 16 bits of a word operand are zero, the operand is considered a halfword when calculating execution time. |                   |                |

Referring to [Table D-1](#), issue-rate cycle numbers are used in the following cases:

- No operand dependency exists on a previous multiply or MAC instruction in the multiply hardware.
- The result of a MAC instruction is used as the accumulate operand of a subsequent MAC instruction in the multiply hardware. In this case, the processor is capable of forwarding the required result within the time imposed by the issue-rate.

Latency cycle numbers are used in the following cases:

- No multiply or MAC instruction is present in the multiply hardware when the current instruction is executed.
- An operand of a multiply or MAC instruction depends on the result of a previous multiply or MAC instruction in the multiply hardware. An exception to this rule is described in the issue-rate rules described above.

## Scalar Load Instructions

Cacheable load instructions that hit in the data cache usually execute in one cycle.

Cacheable and non-cacheable load instructions that hit in the data fill buffer also execute (usually) in one cycle.

The pipelining of load instructions by the processor can cause loads that hit in the cache or fill buffer to take extra cycles. If a load instruction is followed by an instruction that uses the loaded data, a load-use dependency exists. When the loaded data is available, it is forwarded to the operand register of the dependent instruction. This prevents a processor stall from occurring due to missing operand data. This data forwarding adds an extra latency cycle when updating the appropriate GPR. In this case, the load appears to execute in two cycles.

## Load Misses and Uncacheable Loads

Cacheable load misses and non-cacheable loads incur penalty cycles for accessing memory over the PLB. These penalty cycles depend on the speed of the PLB and when the address acknowledge is returned over the PLB. Assuming the PLB operates at the same frequency as the processor and that the address acknowledge is returned in the same cycle the data-cache unit asserts the PLB request, the number of penalty cycles are as follows:

- Six cycles if operand forwarding is enabled.
- Seven cycles if operand forwarding is not enabled.

Additional cycles are required if the system performance does not match the above assumptions.

The PPC405 can execute instructions following a load miss or non-cacheable load if those subsequent instructions do not have a load-use dependency on the load data. When possible, the instruction using the load data should be separated from the load instruction by as many non-use instructions as possible. This enables the processor to continue executing instructions with minimal delay while the load data is accessed.

## Scalar Store Instructions

Cacheable store instructions that miss in the data cache are queued by the data-cache unit so that they appear to execute in a single cycle (if the store is aligned properly). Non-cacheable store instructions are handled in the same way. Under certain conditions, the data-cache unit can queue up to three store instructions (see **Pipeline Stalls**, page 482 for more information.)

All aligned **stwcx**. instructions execute in two cycles.

## String and Multiple Instructions

The access time for load/store string and load/store multiple instructions depends on the alignment of the data being accessed.

String instructions are decomposed by the processor into multiple word-aligned accesses. The execution time for string instructions is calculated as follows (assuming data-cache hits):

- Access to leading bytes consume one cycle. Unused bytes are discarded if the leading bytes are not aligned on a word boundary.
- Access to intermediate bytes consume one cycle for each word accessed.
- Access to trailing bytes consume one cycle. Unused bytes are discarded if the trailing bytes are not aligned on a word boundary.

Figure D-1 shows an example of a 21-byte string with unaligned leading and trailing bytes. Shaded boxes represent bytes outside the string that are discarded by the processor.

| Address | 0 |   |   |   | 4 |   |   |   | 8 |   |   |    | 12 |    |    |    | 16 |    |    |    | 20 |    |  |  |
|---------|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|--|--|
| Data    |   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |  |  |

Figure D-1: String Access Example

In the above example, access to the string requires six cycles, assuming data-cache hits. This is calculated as follows:

- One cycle is required to access the bytes at addresses 1, 2, and 3. The byte at address 0 is also accessed but discarded.
- Four cycles are required to access the four words at addresses 4, 8, 12, and 16 (one cycle for each word).
- One cycle is required to access the bytes at addresses 20 and 21. The bytes at addresses 22 and 23 are also accessed but discarded.

Load/store multiple instructions are also decomposed by the processor into multiple word-aligned accesses. Unaligned words are assembled (loads) or disassembled (stores) by the processor during the access. The execution time for these instructions is calculated as follows (assuming data-cache hits):

- Access to the leading word consumes one cycle. Unused bytes are discarded if the leading word is not aligned on a word boundary.
- Access to intermediate words consume one cycle for each word accessed.
- Access to the trailing word consumes one cycle. Unused bytes are discarded if the trailing word is not aligned on a word boundary.

Figure D-2 shows an example of a 5-word unaligned operand. Shaded boxes represent bytes outside the operand that are discarded by the processor.

|         |   |   |   |    |    |    |  |  |  |
|---------|---|---|---|----|----|----|--|--|--|
| Address | 0 | 4 | 8 | 12 | 16 | 20 |  |  |  |
| Data    |   | 0 | 1 | 2  | 3  | 4  |  |  |  |

Figure D-2: Multiple-Word Access Example

In the above example, access to the multiple-word operand requires six cycles, assuming data-cache hits. This is calculated as follows:

- One cycle is required to access the first three bytes of word 0. The byte at address 0 is also accessed but discarded.
- Four cycles are required to access the remaining byte of word 0, all bytes in words 1, 2, and 3, and the first three bytes of word 4.
- One cycle is required to access the last byte in word 4. The bytes at addresses 21, 22, and 23 are also accessed but discarded.

## Instruction Cache Misses

Cacheable instruction-fetch misses and non-cacheable instruction-fetches incur penalty cycles for accessing memory over the PLB. These penalty cycles depend on the speed of the PLB and when the address acknowledge is returned over the PLB. The number of penalty cycles are as follows:

- Three cycles if the access is a sequential instruction fetch.
- Four cycles if the access is due to a taken branch recognized by the instruction prefetch buffer.
- Five cycles if the access is due to a taken branch recognized by the instruction decode unit.

The above penalty cycle numbers assume the following:

- The PLB operates at the same frequency as the processor.
- The address acknowledge is returned in the same cycle the data-cache unit asserts the PLB request.
- The target instruction is returned in the cycle following the address acknowledge.

Additional cycles are required if the system performance does not match the above assumptions.

## PowerPC® 6xx/7xx Compatibility

This appendix outlines the programming model differences between the 40x family and the 6xx/7xx family of PowerPC processors. The PowerPC 6xx/7xx family complies with the original PowerPC architecture designed for desktop applications. The PowerPC 40x family complies with the PowerPC embedded-environment architecture designed for embedded applications. The information contained in this appendix is useful to system programmers porting software from one family to another.

The two architectures are compatible at the user instruction-set architecture (UISA) level but differ at the level of the virtual-environment architecture (VEA) and operating-environment architecture (OEA). The PowerPC embedded-environment architecture optimizes the VEA and OEA to meet the unique requirements of embedded applications. These optimizations include changes in memory management, cache management, exceptions, timer resources, and others. Many of these optimizations are reflected by the different special-purpose registers (SPRs) supported by the families.

Porting software between implementations is usually limited to the operating-system kernel and other privileged-mode software. Applications usually require no modification. Software porting can be simplified through the use of structured programming methods that localize program modules requiring modification. For example, if all access to the time base are performed using a single function, only that function needs to be modified when porting software to another PowerPC processor.

More information on the PowerPC architecture can be found in the *PowerPC™ Microprocessor Family: The Programming Environments*. Refer to implementation-specific documentation for more information on initialization and configuration, performance considerations, special-purpose registers, and other software-visible details that can vary from processor to processor.

## Registers

**Table E-1** summarizes the registers supported by the PowerPC 40x family that are not supported by the PowerPC 6xx/7xx family. **Table E-2** summarizes the registers supported by the PowerPC 6xx/7xx family that are not supported by the PowerPC 40x family. Not all registers shown for a particular family are supported by all members within that family.

**Table E-1: 40x Registers Not Supported by 6xx/7xx Processors**

| Name    | Description                         | Purpose                 |
|---------|-------------------------------------|-------------------------|
| SPRG4-7 | SPR general-purpose registers 4-7   | Software defined        |
| USPRG0  | User SPR general-purpose register 0 |                         |
| CCR0    | Core-configuration register         | Processor configuration |

Table E-1: 40x Registers Not Supported by 6xx/7xx Processors

| Name              | Description                             | Purpose                            |
|-------------------|-----------------------------------------|------------------------------------|
| DCCR              | Data-cache cacheability register        | Storage control                    |
| DCWR              | Data-cache write-through register       |                                    |
| ICCR              | Instruction-cache cacheability register |                                    |
| SGR               | Storage Guarded Register                |                                    |
| SLER              | Storage Little-Endian Register          |                                    |
| SU0R              | Storage User-Defined 0 Register         |                                    |
| ZPR               | Zone-Protection Register                |                                    |
| DCRs              | Device control registers                | External device control            |
| DEAR              | Data-error address register             | Exception and interrupt processing |
| ESR               | Exception-syndrome register             |                                    |
| EVPR              | Exception-vector prefix register        |                                    |
| SRR2              | Save/restore register 2                 |                                    |
| SRR3              | Save/restore register 3                 |                                    |
| PIT               | Programmable-Interval Timer             | Timer resources                    |
| TCR               | Timer-Control Register                  |                                    |
| TSR               | Timer-Status Register                   |                                    |
| DAC <sub>n</sub>  | Data address-compare registers          | Debugging                          |
| DBCR <sub>n</sub> | Debug-control registers                 |                                    |
| DBSR              | Debug-status register                   |                                    |
| DVC <sub>n</sub>  | Data value-compare registers            |                                    |
| IAC <sub>n</sub>  | Instruction address-compare registers   |                                    |
| ICDBDR            | Instruction-cache debug-data register   |                                    |

Table E-2: 6xx/7xx Registers Not Supported by 40x Processors

| Name              | Description                             | Purpose                            |
|-------------------|-----------------------------------------|------------------------------------|
| HID <sub>n</sub>  | Hardware implementation registers       | Processor configuration            |
| DBAT <sub>n</sub> | Data BATs                               | Memory management                  |
| IBAT <sub>n</sub> | Instruction BATs                        |                                    |
| SDR1              | Page table base address                 |                                    |
| SR <sub>n</sub>   | Segment registers                       |                                    |
| EAR               | External address register               | External device control            |
| DAR               | Data address register                   | Exception and interrupt processing |
| DSISR             | Data storage interrupt status register  |                                    |
| DEC               | Decrementer                             | Timer resources                    |
| DABR              | Data-address breakpoint register        | Exception and interrupt processing |
| IABR              | Instruction-address breakpoint register |                                    |

Table E-2: 6xx/7xx Registers Not Supported by 40x Processors

| Name               | Description                                   | Purpose                |
|--------------------|-----------------------------------------------|------------------------|
| MMCR <sub>n</sub>  | Monitor control registers                     | Performance monitoring |
| PMC <sub>n</sub>   | Performance counters                          |                        |
| SIA                | Sampled instruction address                   |                        |
| UMMCR <sub>n</sub> | Monitor control registers (user mode)         |                        |
| UPMC <sub>n</sub>  | Performance counters (user mode)              |                        |
| USIA               | Sampled instruction address (user mode)       |                        |
| ICTC               | Instruction cache throttling control register | Cache control          |
| L2CR               | L2 cache control register                     |                        |
| THRM <sub>n</sub>  | Thermal assist unit registers                 | Thermal management     |

## Machine-State Register

Several bits within the machine-state register are supported by either PowerPC 40x processors or PowerPC 6xx/7xx processors, but not both. Others have different meanings depending on the processor family. Table E-3 compares these differences.

Table E-3: Comparison of MSR Bit Definitions

| MSR Bit | PowerPC 40x Family                  | PowerPC 6xx/7xx Family      |
|---------|-------------------------------------|-----------------------------|
| 0:5     | Reserved                            | Reserved                    |
| 6       | AP—Auxiliary Processor Available    |                             |
| 7:11    | Reserved                            |                             |
| 12      | APE—APU Exception Enable            |                             |
| 13      | WE—Wait State Enable                | POW—Power Management Enable |
| 14      | CE—Critical Interrupt Enable        | Reserved                    |
| 15      | Reserved                            | ILE—Interrupt Little Endian |
| 16      | EE—External Interrupt Enable        |                             |
| 17      | PR—Privilege Level                  |                             |
| 18      | FP—Floating-Point Available         |                             |
| 19      | ME—Machine-Check Enable             |                             |
| 20      | FE0—Floating-Point Exception-Mode 0 |                             |
| 21      | DWE—Debug Wait Enable               | SE—Single-Step Trace Enable |
| 22      | DE—Debug Interrupt Enable           | BE—Branch Trace Enable      |
| 23      | FE1—Floating-Point Exception-Mode 1 |                             |
| 24      | Reserved                            |                             |
| 25      | Reserved                            | IP—Exception Prefix         |
| 26      | IR—Instruction Relocate             |                             |
| 27      | DR—Data Relocate                    |                             |
| 28      | Reserved                            |                             |



Table E-3: Comparison of MSR Bit Definitions

| MSR Bit | PowerPC 40x Family | PowerPC 6xx/7xx Family             |
|---------|--------------------|------------------------------------|
| 29      | Reserved           | PM—Performance Monitor Marked Mode |
| 30      |                    | RI—Recoverable Exception           |
| 31      |                    | LE—Little-Endian Mode Enable       |

## Processor-Version Register

The contents of the processor-version register (PVR) are implementation dependent.

## Memory Management

The primary function of memory management is the translation of effective addresses to physical addresses for instruction memory and data memory accesses. The secondary function of memory management is to provide memory-access protection and memory-attribute control. Memory management is handled by the memory-management unit (MMU) in the processor.

## Memory Translation

The PowerPC 6xx/7xx family manages memory translation by dividing the address space into blocks, segments, and pages. The address-space divisions are characterized as follows:

- Blocks specify large, contiguous memory regions (from 128KB to 256MB) with common access protection and memory attributes. Blocks are defined using SPRs called block address-translation (BAT) registers. The BAT registers are used by the MMU to translate a 32-bit effective address within a BAT to a 32-bit physical address.
- Segments are contiguous 256MB memory regions. Segment registers are used by the MMU to translate a 32-bit effective address within a segment to a 52-bit virtual address. 16 segment registers are available and they are accessed using move-to and move-from segment register instructions.
- Pages are contiguous 4KB memory regions. The MMU uses page-translation tables to translate a 52-bit virtual address within a page to a 32-bit physical address. The page-translation tables are created by software and stored in system memory. The processor uses a translation look-aside buffer (TLB) to cache the most frequently used translations. The processor manages many TLB functions in hardware, including page-table walking and TLB entry replacement. TLB instructions are provided for some software management, such as TLB invalidation.

If an effective address is not part of a memory region defined by a BAT, translation of that address to a physical address is handled by the combined segment and page translation mechanism. The effective address is translated first into a virtual address using the segment registers. The resulting virtual address is translated to a physical address using the page tables.

The PowerPC 40x family manages memory translation by dividing the address space into pages. BAT and segment translation are not supported. Page translation in the PowerPC 40x family has the following characteristics:

- Pages are contiguous, variable-sized memory regions. Page sizes can vary from 1KB to 16MB.
- Page-translation tables are created by software and stored in system memory. The most frequently used translations are cached in the TLB. TLB management is the responsibility of software, not hardware.
- The MMU uses the page-translation tables to translate a 40-bit virtual address to a 32-bit physical address. The 40-bit virtual address is the combination of the 32-bit

effective address appended to the 8-bit PID.

**Table E-4** summarizes the memory-translation differences between PowerPC 40x processors and PowerPC 6xx/7xx processors. Gray-shaded cells represent unsupported features.

**Table E-4: Summary of Memory Translation Differences**

| Memory-Translation Feature                    | PowerPC 40x Family                               | PowerPC 6xx/7xx Family                                                                  |
|-----------------------------------------------|--------------------------------------------------|-----------------------------------------------------------------------------------------|
| Block address translation (BAT)               |                                                  | Supported using separate instruction and data BAT registers (SPRs)                      |
| Segment translation                           |                                                  | Supported using 16 segment registers and special instructions to access those registers |
| Page translation                              | Supported                                        | Supported                                                                               |
| Virtual-address width                         | 40 bits (8-bit PID and 32-bit effective address) | 52 bits                                                                                 |
| Page size                                     | 1KB, 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, and 16MB  | 4KB                                                                                     |
| Page table entry                              | Flexible - software defined                      | Defined by PowerPC architecture                                                         |
| Page table organization                       | Flexible - software defined                      | Hashed                                                                                  |
| Page history recording (reference and change) | Software                                         | Hardware                                                                                |
| TLB-entry replacement                         | Software                                         | Hardware                                                                                |
| TLB instructions                              | tlbia<br>tlbre<br>tlbsx[.]<br>tlbsync<br>tlbwe   | tlbia<br>tlbie<br>tlbsync                                                               |
| TLB-miss exceptions                           | Supported                                        |                                                                                         |

## Memory Protection

Both the PowerPC 6xx/7xx and PowerPC 40x processors support no-access, read-only, and read/write memory protection. However, the methods used to specify protection differ in the two processor families:

- PowerPC 6xx/7xx processors:
  - Protection is specified during segment and page translation using a combination of protection keys stored in the segment registers and page-protection bits stored in the page-table entries.
  - Protection is specified during BAT translation using protection bits stored in the BAT registers.
- PowerPC 40x processors:
  - Protection is specified during page translation using page-protection bits stored in the TLB entries.
  - Zone protection can be used to override the access protection specified in a TLB entry. Fields within the zone-protection register (ZPR) define the protection level of a page or set of pages.

## Memory Attributes

Both the PowerPC 6xx/7xx and PowerPC 40x processors support the following memory attributes:

- Write through (W).
- Caching inhibited (I).

- Memory coherence (M). This attribute is not supported by the PPC405 and is ignored.
- Guarded (G).

PowerPC 40x processors also support the following additional memory attributes:

- User-defined (U0).
- Endian (E).

All memory attributes supported by PowerPC 40x processors can be applied in real mode (address translation disabled) using storage-attribute control registers. These registers are not supported by PowerPC 6xx/7xx processors.

## Cache Management

The PowerPC architecture does not define the type, organization, implementation, or existence of internal or external caches. To maximize portability, software that operates on multiple PowerPC implementations should always assume a Harvard cache model is implemented.

**Table E-5** summarizes the PowerPC 40x cache-management instructions not supported by the PowerPC 6xx/7xx family. Implementations within the PowerPC 40x family can vary in the detailed operation of these instructions.

**Table E-5: 40x Cache-Management Instructions**

| Instruction   | 405                                                       | 401 and 403                                                          |
|---------------|-----------------------------------------------------------|----------------------------------------------------------------------|
| <b>dccci</b>  | Invalidates individual data-cache congruence classes.     |                                                                      |
| <b>dcread</b> | Data-cache debug function controlled by CCR0 register.    | Data-cache debug function controlled by CDBCR register.              |
| <b>icbt</b>   | Instruction-cache block touch, executable from user mode. | Instruction-cache block touch, executable from privileged mode only. |
| <b>iccci</b>  | Invalidates the entire instruction cache.                 | Invalidates individual instruction-cache congruence classes.         |
| <b>icread</b> | Function controlled by CCR0 register.                     | Function controlled by CDBCR register.                               |

Some PowerPC processors also support cache locking. Cache locking prevents the replacement of a cacheline regardless of the frequency of its use. Cache locking is supported as follows:

- PowerPC 401 processors—cachelines can be individually locked.
- PowerPC 403 processors—not supported.
- PowerPC 405 processors—not supported.
- PowerPC 6xx/7xx processors—the instruction and data caches can be locked in their entirety.

## Exceptions

The PowerPC 40x family implements several extensions to the exception and interrupt mechanism supported by PowerPC 6xx/7xx processors. The extensions supported by PowerPC 40x processors are:

- A dual-level interrupt structure defining critical and noncritical interrupts. PowerPC 6xx/7xx processors implement a single-level interrupt structure that does not distinguish between critical and noncritical interrupts.
- New save/restore registers (SRR2/SRR3) that support critical interrupts. The PowerPC 40x family uses the SRR0/SRR1 save/restore registers for noncritical interrupts, which are used for all interrupts in the PowerPC 6xx/7xx family.

- Differences in exception-related bits in the machine-state register (MSR). See [Table E-3, page 887](#) for a summary.
- A new interrupt-return instruction (**rfci**) that supports critical interrupts. The PowerPC 40x family uses the **rfi** instruction to return from noncritical interrupts, which is used to return from all interrupts in the PowerPC 6xx/7xx family.
- New special-purpose registers for recording exception information. The PowerPC 40x family defines two registers:
  - The exception-syndrome register (ESR) used to identify the cause of an exception.
  - The data exception-address register (DEAR) used to record the memory-operand effective address of a data-access instruction that causes certain exceptions. The data-address register (DAR) performs a similar function in PowerPC 6xx/7xx processors.
- Greater flexibility in relocating the interrupt-handler table. The exception-vector prefix register (EVPR) supports relocating the interrupt-handler table anywhere in physical-address space, with a base address that falls on a 64KB-aligned boundary. The PowerPC 6xx/7xx family supports two locations for the interrupt-handler table: `0x000n_nnnn` or `0xFFFFn_nnnn`, selected by using the MSR[IP] bit.
- New exceptions and interrupts are defined. Some exceptions and interrupts supported by the PowerPC 6xx/7xx family are not supported by PowerPC 40x processors. [Table E-6](#) summarizes the differences between the exception and interrupt vectors defined by the two families. Gray-shaded cells represent unsupported interrupt vectors. Not all processors within a family support all of the exceptions and interrupts defined by the family.

**Table E-6: Summary of Exception and Interrupt Vector Differences**

| Vector Offset | PowerPC 40x Family          | PowerPC 6xx/7xx Family         |
|---------------|-----------------------------|--------------------------------|
| 0x0100        | Critical-Input              | System Reset                   |
| 0x0900        |                             | Decrementer                    |
| 0x0D00        |                             | Trace                          |
| 0x0F00        |                             | Performance Monitor            |
| 0x0F20        | APU Unavailable             |                                |
| 0x1000        | Programmable-Interval Timer | Instruction-Translation Miss   |
| 0x1010        | Fixed-Interval Timer        |                                |
| 0x1020        | Watchdog Timer              |                                |
| 0x1100        | Data-TLB Miss               | Data-Translation Miss (loads)  |
| 0x1200        | Instruction-TLB Miss        | Data-Translation Miss (stores) |
| 0x1300        |                             | Instruction-Address Breakpoint |
| 0x1400        |                             | System Management              |
| 0x1700        |                             | Thermal Management             |
| 0x2000        | Debug                       |                                |

## Timer Resources

The PowerPC 40x family implements new timer features. These are:

- The programmable-interval timer (PIT) register. This register decrements at the same clock rate as the time base. Its function replaces that of the decrementer in the PowerPC 6xx/7xx family.

- The programmable-interval timer (PIT) interrupt. This interrupt is triggered by a time-out on the PIT registers. Its function replaces that of the decremented interrupt in the PowerPC 6xx/7xx family.
- The fixed-interval timer (FIT) interrupt. This interrupt is triggered by a pre-determined bit transition in the time base. This feature is not supported by the PowerPC 6xx/7xx family.
- The watchdog timer (WDT) interrupt. This critical interrupt is triggered by a pre-determined bit transition in the time base. This feature is not supported by the PowerPC 6xx/7xx family.
- The timer-control register (TCR). This register controls the PowerPC 40x timer resources. It is not supported by the PowerPC 6xx/7xx family.
- The timer-status register (TSR). This register is used by the PowerPC 40x timer resources to report status. It is not supported by the PowerPC 6xx/7xx family.

## Other Differences

### Instructions

PowerPC 40x processors can support implementation-specific instructions that are not supported in PowerPC 6xx/7xx processors. For example, the multiply-accumulate (MAC) instructions are not supported by PowerPC 6xx/7xx processors. Refer to [Table B-32, page 834](#), for a list of implementation dependent PPC405 instructions. This table also shows which PPC405 instructions are not supported by the PowerPC architecture.

### Endian Support

The default memory-access order for all PowerPC processors is big-endian. The PowerPC embedded-environment architecture defines a true little-endian memory-access capability that is implemented using the endian storage attribute (E). The PPC405 supports this capability. The PowerPC architecture supports a little-endian mode that is implemented by PowerPC 6xx/7xx processors. This mode is not supported by the PPC405.

### Debug Resources

Debug resources are implementation dependent. In general, all PowerPC 40x processors support debug events on both instruction addresses and data addresses. Debug events are controlled using the DBCR0 and DBCR1 registers. Debug status is reported by the DBSR register. PowerPC 6xx/7xx processors support debug resources to varying degrees, but the capabilities are often less comprehensive than those supported by PowerPC 40x processors.

### Power Management

The PowerPC 40x family implements power management using the MSR[WE] bit. Setting this bit places the processor in the wait state. Power management is disabled when an interrupt occurs.

The PowerPC 6xx/7xx family similarly implements power management using the MSR[POW] bit. PowerPC 7xx processors support four different power states, programmed using the HID0 register. Power management is disabled when an interrupt occurs.

## PowerPC® Book-E Compatibility

---

This appendix outlines the programming model differences between the PowerPC embedded-environment architecture (40x family of processors) and the PowerPC Book-E architecture. In general, the PowerPC Book-E architecture extends the embedded-system features introduced by the PowerPC embedded-environment architecture. The PowerPC Book-E architecture also introduces 64-bit instructions and addressing, although the scope of this appendix is restricted to 32-bit operations. The information contained in this appendix is useful as a guide to system programmers porting 32-bit software from one family to another.

At the 32-bit user instruction-set architecture (UISA) level, the PowerPC Book-E architecture is compatible with the PowerPC embedded-environment architecture. However, there are differences between the architectures at the virtual-environment architecture (VEA) and operating-environment architecture (OEA) levels. These differences include changes in memory management, cache management, memory synchronization, exceptions, timer resources, and others. Many of the differences are reflected the deletion, modification, and introduction of special-purpose registers.

Porting software between implementations is usually limited to the operating-system kernel and other privileged-mode software. 32-bit applications typically require no modification. Software porting can be simplified through the use of structured programming methods that localize program modules requiring modification. For example, if all access to the time base are performed using a single function, only that function needs to be modified when porting software to another PowerPC processor.

More information on the PowerPC Book-E architecture can be found in the *Book E: Enhanced PowerPC™ Architecture*. Refer to implementation-specific documentation for more information on initialization and configuration, performance considerations, special-purpose registers, and other software-visible details that can vary from processor to processor.

### Registers

**Table F-1** summarizes the registers supported by PowerPC 40x family that are not defined by the PowerPC Book-E architecture. This table indicates whether or not a similar register with a different name and SPR number is defined by the PowerPC Book-E architecture.

Table F-1: Registers Not Defined in PowerPC Book-E Architecture

| Name | Description                             | PowerPC Book-E Architecture Equivalent |
|------|-----------------------------------------|----------------------------------------|
| DCCR | Data-cache cacheability register        | None                                   |
| DCWR | Data-cache write-through register       |                                        |
| ICCR | Instruction-cache cacheability register |                                        |
| SGR  | Storage Guarded Register                |                                        |
| SLER | Storage Little-Endian Register          |                                        |
| SU0R | Storage User-Defined 0 Register         |                                        |
| ZPR  | Zone-Protection Register                |                                        |
| EVPR | Exception-vector prefix register        | IVPR                                   |
| SRR2 | Save/restore register 2                 | CSRR0                                  |
| SRR3 | Save/restore register 3                 | CSRR1                                  |
| PIT  | Programmable-Interval Timer             | DEC                                    |

Table F-2 summarizes the registers supported by the PowerPC 40x processors that have a different SPR number or a different name defined by the PowerPC Book-E architecture.

Table F-2: Renumbered/Renamed Registers in the PowerPC Book-E Architecture

| PowerPC 40x Family |      | PowerPC Book-E Architecture |      |
|--------------------|------|-----------------------------|------|
| Name               | SPRN | Name                        | SPRN |
| DAC1               | 1014 | DAC1                        | 316  |
| DAC2               | 1015 | DAC2                        | 317  |
| DBCR0              | 1010 | DBCR0                       | 308  |
| DBCR1              | 957  | DBCR1                       | 309  |
| DBSR               | 1008 | DBSR                        | 304  |
| DEAR               | 981  | DEAR                        | 61   |
| DVC1               | 950  | DVC1                        | 318  |
| DVC2               | 951  | DVC2                        | 319  |
| ESR                | 980  | ESR                         | 62   |
| IAC1               | 1012 | IAC1                        | 312  |
| IAC2               | 1013 | IAC2                        | 313  |
| IAC3               | 948  | IAC3                        | 314  |
| IAC4               | 949  | IAC4                        | 315  |
| PID                | 945  | PID                         | 48   |
| TCR                | 986  | TCR                         | 340  |
| TSR                | 984  | TSR                         | 336  |
| USPRG0             | 256  | SPRG8                       | 256  |

Table F-3 summarizes the new registers defined by the PowerPC Book-E architecture or present in the PowerPC 440 processor.

Table F-3: New Registers in the PowerPC Book-E Architecture

| Name                                         | Description                                 | Purpose                            |
|----------------------------------------------|---------------------------------------------|------------------------------------|
| MMUCR                                        | Memory-management unit control register     | Memory management                  |
| PIR                                          | Processor ID Register                       | Multiprocessing                    |
| CSRR0                                        | Critical save/restore register 0            | Exception and interrupt processing |
| CSRR1                                        | Critical save/restore register 1            |                                    |
| IVOR0–IVOR15                                 | Interrupt-vector offset registers           |                                    |
| IVPR                                         | Interrupt-vector prefix register            |                                    |
| DEC                                          | Decrementer                                 | Timer resources                    |
| DECAR                                        | Decrementer Auto Reload                     |                                    |
| DNV <sub>n</sub> <sup>1</sup>                | Data-cache normal victim register           | Cache control                      |
| DTV <sub>n</sub> <sup>1</sup>                | Data-cache transient victim register        |                                    |
| DVLIM <sup>1</sup>                           | Data-cache victim limit                     |                                    |
| INV <sub>n</sub> <sup>1</sup>                | Instruction-cache normal victim register    |                                    |
| ITV <sub>n</sub> <sup>1</sup>                | Instruction-cache transient victim register |                                    |
| IVLIM <sup>1</sup>                           | Instruction-cache victim limit              |                                    |
| DBCR2                                        | Debug-control register 2                    | Debugging                          |
| DCDBTRH <sup>1</sup><br>DCDBTRL <sup>1</sup> | Data-cache debug tag registers              |                                    |
| ICDBTRH <sup>1</sup><br>ICDBTRL <sup>1</sup> | Instruction-cache debug tag registers       |                                    |

**Notes:**  
1. Implemented in the 440 processor, but not defined by the PowerPC Book-E architecture.

## Machine-State Register

The PowerPC Book-E architecture redefines some of the bits in the machine-state register (MSR). Table F-4 compares the MSR bit definitions used by PowerPC 40x processors and PowerPC Book-E processors.

Table F-4: Comparison of MSR Bit Definitions

| MSR Bit | PowerPC 40x Family               | PowerPC Book-E Architecture           |
|---------|----------------------------------|---------------------------------------|
| 0:5     | Reserved                         |                                       |
| 6       | AP—Auxiliary Processor Available | Implementation dependent              |
| 7:11    | Reserved                         | Reserved                              |
| 12      | APE—APU Exception Enable         |                                       |
| 13      | WE—Wait State Enable             |                                       |
| 14      | CE—Critical Interrupt Enable     |                                       |
| 15      | Reserved                         | Reserved: ILE—Interrupt Little Endian |
| 16      | EE—External Interrupt Enable     |                                       |
| 17      | PR—Privilege Level               |                                       |
| 18      | FP—Floating-Point Available      |                                       |



Table F-4: Comparison of MSR Bit Definitions

| MSR Bit | PowerPC 40x Family                  | PowerPC Book-E Architecture            |
|---------|-------------------------------------|----------------------------------------|
| 19      | ME—Machine-Check Enable             |                                        |
| 20      | FE0—Floating-Point Exception-Mode 0 |                                        |
| 21      | DWE—Debug Wait Enable               | Implementation dependent               |
| 22      | DE—Debug Interrupt Enable           |                                        |
| 23      | FE1—Floating-Point Exception-Mode 1 |                                        |
| 24      | Reserved                            |                                        |
| 25      | Reserved                            | Reserved: IP—Interrupt Prefix          |
| 26      | IR—Instruction Relocate             | IS—Instruction Address Space           |
| 27      | DR—Data Relocate                    | DS—Data Address Space                  |
| 28:29   | Reserved                            |                                        |
| 30      | Reserved                            | Reserved: RI—Recoverable Interrupt     |
| 31      |                                     | Reserved: LE—Little-Endian Mode Enable |

## Processor-Version Register

The contents of the processor-version register (PVR) are implementation dependent.

## Memory Management

The primary function of memory management is the translation of effective addresses to physical addresses for instruction memory and data memory accesses. The secondary function of memory management is to provide memory-access protection and memory-attribute control. Memory management is handled by the memory-management unit (MMU) in the processor.

## Memory Translation

The PowerPC Book-E architecture extends the page translation capabilities supported by PowerPC 40x processors. These extensions are summarized in Table F-5. Real mode is not supported by PowerPC Book-E implementations. Address translation is always enabled, and one or more TLB entries are initialized by the processor during reset so that instructions can be fetched and data accessed following reset.

The *TLB invalidate all* (**tlbia**) instruction is not supported by PowerPC Book-E processors because translation is always enabled. At least one valid TLB entry must exist—the entry that maps the TLB-miss interrupt handler.

Table F-5: Summary of Memory Translation Extensions

| Memory-Translation Feature | PowerPC 40x Family                                                                                     | 6xx/7xx Family                                                                                                                                                          |
|----------------------------|--------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Real mode                  | Supported                                                                                              | Unsupported                                                                                                                                                             |
| Virtual-address width      | 40 bits: <ul style="list-style-type: none"> <li>8-bit PID</li> <li>32-bit effective address</li> </ul> | 97 bits: <ul style="list-style-type: none"> <li>1-bit instruction or data address-space (from the MSR)</li> <li>32-bit PID</li> <li>64-bit effective address</li> </ul> |
| Page size                  | 1KB to 16MB                                                                                            | 1KB to 1TB (terabyte)                                                                                                                                                   |
| TLB instructions           | tlbia<br>tlbre<br>tlbsx[.]<br>tlbsync<br>tlbwe                                                         | tlbivax<br>tlbre<br>tlbsx[.]<br>tlbsync<br>tlbwe                                                                                                                        |

## Memory Protection

The TLB entries defined by the PowerPC Book-E architecture support the following access controls, which can be independently configured for privileged mode and user mode accesses:

- Execute
- Read
- Write

Software can use any combination of the access controls to manage memory protection. For example, read/write access is specified by enabling both the read and write access controls. No-access is specified by disabling both controls.

PowerPC 40x implementations control memory protection using a combination of fields in the TLB entry and the zone-protection register (ZPR). These controls support many of the same protection characteristics available in PowerPC Book-E processors, but not all of them. For example, write-only protection cannot be specified.

Zone protection is not supported by the PowerPC Book-E architecture.

## Memory Attributes

The PowerPC 40x family and PowerPC Book-E processors support the following memory attributes:

- Write through (W).
- Caching inhibited (I).
- Memory coherence (M). This attribute is not supported by the PPC405 and is ignored.
- Guarded (G).
- Endian (E).
- User-defined. The PowerPC 40x family supports a single user-defined attribute (U0). The PowerPC Book-E architecture supports up to four user-defined attributes (U0, U1, U2, and U3).

All memory attributes supported by PowerPC 40x processors can be used in real mode (address translation disabled) using storage-attribute control registers. These registers are not supported by PowerPC Book-E processors.

## Caches

The PowerPC architecture does not define the type, organization, implementation, or existence of internal or external caches. To maximize portability, software that operates on multiple PowerPC implementations should always assume a Harvard cache model is implemented.

Table F-6 summarizes the cache-management instructions supported by PowerPC 40x processors that are changed in the PowerPC Book-E architecture.

Table F-6: PowerPC 40x Cache-Management Instructions

| Instruction   | PowerPC Book-E Architecture Change                                                                                                             |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>dccci</b>  | This instruction is implementation dependent. On some PowerPC Book-E processors, this instruction invalidates the entire data cache.           |
| <b>dcread</b> | This instruction is implementation dependent. On some PowerPC Book-E processors, the format of data returned by this instruction is different. |
| <b>icbt</b>   | The opcode differs from the opcode recognized by PowerPC 40x processors.                                                                       |
| <b>iccci</b>  | This instruction is implementation dependent. On some PowerPC Book-E processors, this instruction invalidates the entire instruction cache.    |
| <b>icread</b> | This instruction is implementation dependent. On some PowerPC Book-E processors, the format of data returned by this instruction is different. |

Some PowerPC processors also support cache locking. Cache locking prevents the replacement of a cacheline regardless of the frequency of its use. Cache locking is supported as follows:

- PowerPC 401 processors—cachelines can be individually locked.
- PowerPC 403 processors—not supported.
- PowerPC 405 processors—not supported.
- PowerPC 440 processors—cachelines can be individually locked.

## Memory Synchronization

The *memory barrier* (**mbar**) instruction replaces the **eieio** instruction, which uses the same opcode. An MO (memory order) operand can be specified with the **mbar** instruction. This operand is used to specify ordering across a subset of memory-access instructions (for example, order loads but not stores). If the MO operand is zero or not specified, the **mbar** instruction behaves like the **eieio** instruction (orders all memory accesses). This guarantees that existing software that uses **eieio** works properly in PowerPC Book-E implementations.

The *memory synchronize* (**msync**) instruction replaces the **sync** instruction, which uses the same opcode. The **msync** instruction behaves identically to the **sync** instruction. This guarantees that existing software that uses **sync** works properly in PowerPC Book-E implementations.

## Exceptions

Within implementations of the PowerPC Book-E architecture, the effect of invalid instruction forms or other exception-causing events can differ from that of PowerPC 40x processors. In the PowerPC 440 for example, an **stwcx.** to an unaligned memory operand yields a boundedly undefined result. In the PPC405, this operation causes an alignment exception.

The PowerPC Book-E architecture replaces the exception-vector prefix register (EVPR) with the interrupt-vector prefix register (IVPR). The IVPR contains the high-order 16 bits of the exception-vector effective address, which is the same function performed by the EVPR.

The PowerPC Book-E architecture also defines 16 interrupt-vector offset registers (IVOR0–IVOR15) that replace the function of the predefined vector offsets assigned to each exception. Any arbitrary word-aligned vector offset can be loaded into these registers, which are assigned to a specific exception.

When an exception occurs, the processor calculates the interrupt-handler effective address by adding the contents of the IVPR to the contents of the appropriate IVOR $n$ . System software can emulate the operation of the PowerPC 40x interrupt mechanism by preloading the IVOR $n$  registers with the appropriate vector offsets, as shown in [Table F-7](#).

**Table F-7: Exceptions and Associated IVOR $n$  Registers**

| IVOR   | Exception                                 | PowerPC 40x Offset |
|--------|-------------------------------------------|--------------------|
| IVOR0  | Critical Input                            | 0x0100             |
| IVOR1  | Machine Check                             | 0x0200             |
| IVOR2  | Data Storage                              | 0x0300             |
| IVOR3  | Instruction Storage                       | 0x0400             |
| IVOR4  | External                                  | 0x0500             |
| IVOR5  | Alignment                                 | 0x0600             |
| IVOR6  | Program                                   | 0x0700             |
| IVOR7  | FPU Unavailable                           | 0x0800             |
| IVOR8  | System Call                               | 0x0C00             |
| IVOR9  | APU Unavailable                           | 0x0F20             |
| IVOR10 | Decrementer (Programmable-Interval Timer) | 0x1000             |
| IVOR11 | Fixed-Interval Timer                      | 0x1010             |
| IVOR12 | Watchdog Timer                            | 0x1020             |
| IVOR13 | Data TLB Miss                             | 0x1100             |
| IVOR14 | Instruction TLB Miss                      | 0x1200             |
| IVOR15 | Debug                                     | 0x2000             |

Some bits in the exception-syndrome register (ESR) are redefined to support different exception conditions. These changes are shown in [Table F-8](#).

**Table F-8: Comparison of ESR Bit Definitions**

| Bit | PowerPC 40x Function                   | PowerPC Book-E Function       |
|-----|----------------------------------------|-------------------------------|
| 0   | MCI—Instruction Machine Check          | Implementation dependent      |
| 1:3 | Reserved                               |                               |
| 4   | PIL—Program, Illegal Instruction       |                               |
| 5   | PPR—Program, Privileged Instruction    |                               |
| 6   | PTR—Program, Trap Instruction          |                               |
| 7   | PEU—Program, Unimplemented Instruction | FP—Floating-Point Instruction |
| 8   | DST—Data Storage, Store Instruction    | ST—Store                      |

Table F-8: Comparison of ESR Bit Definitions (Continued)

| Bit   | PowerPC 40x Function                              | PowerPC Book-E Function            |
|-------|---------------------------------------------------|------------------------------------|
| 9     | DIZ—Data and Instruction Storage, Zone Protection | Reserved                           |
| 10:11 | Reserved                                          | Implementation dependent           |
| 12    | Program—Floating-Point Instruction                | AP—Auxiliary-Processor Instruction |
| 13    | Program—Auxiliary-Processor Instruction           | PUO—Unimplemented Operation        |
| 14    | Reserved                                          | BO—Byte Ordering                   |
| 15    | Reserved                                          | Reserved                           |
| 16    | Data Storage—U0 Protection                        |                                    |
| 17:23 | Reserved                                          |                                    |
| 24:31 | Reserved                                          | Implementation dependent           |

## Timer Resources

The PowerPC Book-E architecture modifies some aspects of the timer resources, as follows:

- The architecture does not define a *move-from time base* (**mftb**) instruction. Software that reads the time base must use a *move-from SPR* (**mfspr**) instruction with an SPR number corresponding to the appropriate time-base register.
- The programmable-interval timer (PIT) register is replaced by the decremter (DEC). These registers have different SPR addresses.
- A DEC auto-reload mechanism is provided. This mechanism is more flexible than the similar PIT auto-reload mechanism supported by the PowerPC 40x family.
- The programmable-interval timer (PIT) interrupt is replaced by the decremter interrupt.
- The timer-control register (TCR) controls different FIT and watchdog time-out intervals, and it controls the decremter instead of the PIT.
- The timer-status register (TSR) describes decremter status instead of PIT status.

## Other Differences

### Instructions

PowerPC 40x processors and PowerPC Book-E processors can support implementation-specific instructions. For example, the multiply-accumulate (MAC) instructions are considered implementation dependent and are not guaranteed to be supported by other processors. Also, the PowerPC 440 processor supports the implementation-specific *determine left-most zero byte* (**dlmzb**) instruction. Refer to [Table B-32, page 834](#), for a list of implementation dependent PPC405 instructions. This table also shows which PPC405 instructions are not supported by the PowerPC Book-E architecture.

### Debug Resources

Debug resources are implementation dependent. In general, all PowerPC 40x processors and PowerPC Book-E processors support a common set of debug events on both instruction addresses and data addresses. Debug events are controlled using the DBCR<sub>n</sub> registers. Debug status is reported by the DBSR register.

# Index

## A

addition instructions 426 to 428  
addressing  
    *See also* page translation.  
    effective address 380  
    register indirect 416  
    register-indirect immediate-index 414  
    register-indirect index 415  
algebraic-compare instructions 435  
algebraic-shift instructions 440  
alignment  
    *See* operand alignment.  
alignment exception 546  
    partial instruction execution 526  
APU-unavailable exception 551  
atomic memory access 463, 484

## B

big endian 385  
boundary-scan description  
    language 595  
boundedly undefined 391  
branch instructions 403  
    *See also* conditional branch.  
    AA opcode field 408, 409, 410  
    BD opcode field 409, 410  
    branch to CTR 406, 411  
    branch to LR 405, 411  
    branch-conditional absolute 405, 410  
    branch-conditional relative 405, 409  
    branch-unconditional absolute 404, 410  
    branch-unconditional relative 404, 408  
    LI opcode field 408, 410  
    LK opcode field 408, 409, 410, 411  
    target address calculation 408  
branch prediction 406 to 408  
    default prediction 407  
    link register stack 407  
    overriding default prediction 407  
    simplified mnemonics 863  
    y bit 407  
branch taken (BT)  
    *See* debug events.  
byte, definition 383  
byte-reverse instructions  
    *See* load instructions.  
    *See* store instructions.

## C

cache  
    access example 476 to 477

congruence class 474  
debug control 497  
debug instructions 504 to 505  
dirty 475  
flush 480, 502 to 503  
hit 476, 477, 479  
line 474  
losing coherency 499 to 501  
LRU 475  
miss 476, 477, 480  
physical index 475  
physical tag 475  
self-modifying code 503  
software enforced coherency 501 to 503  
virtual index 475, 478  
cache block  
    *See* cache, line.  
cache-control instructions 492 to 495  
    DAC debug events 588  
    effect of access protection 519 to 521  
chip reset  
    *See* reset.  
clear register instructions 865  
compare instructions 434, 864  
complement register  
    instruction 870  
condition register 397  
    CR mask (CRM) 459  
    CR0 397  
    CR1 398  
    CR-logical instructions 412, 864  
    effect of Rc opcode field 397  
    equal (EQ) 398  
    greater than (GT) 398  
    integer instruction update 425  
    less than (LT) 398  
    move instructions 459  
    negative (LT) 398  
    positive (GT) 398  
    summary overflow (SO) 398  
    zero (EQ) 398  
conditional branch  
    BI opcode field 404  
    BO opcode field 403, 404  
    simplified mnemonics 857 to 863  
    specifying conditions 403  
    specifying CR bits 404  
congruence class  
    *See* cache, congruence class.  
context synchronization  
    *See* synchronization, context.  
core-configuration register 495  
    programming guidelines 497  
count leading-zeros  
    instructions 434  
count register 400  
    branching to 406, 411

## CR

*See* condition register.  
critical exception 528  
critical-input exception 539  
CTR  
    *See* count register.

## D

DACn  
    *See* data address-compare registers.  
data address-compare (DAC)  
    *See* debug events.  
data address-compare registers 579  
data cache  
    *See also* cache.  
    control instructions 493 to 495  
    fill buffer 480  
    hint instructions 483  
    line buffer 479  
    load without allocate 481, 496  
    load word as line 481, 496  
    operation 479 to 480  
    pipeline stall 482  
    PLB priority 482, 496, 497  
    store without allocate 481, 496  
data exception-address register 538  
data relocate  
    *See* virtual mode.  
data TLB-miss exception 517, 555  
data value-compare (DVC)  
    *See* debug events.  
data value-compare registers 579  
data-cache cacheability register 490  
data-cache write-through  
    register 489  
data-storage exception 517, 542  
    partial instruction execution 526  
    U0 exception 496  
DBCRn  
    *See* debug-control registers.  
DBSR  
    *See* debug-status register.  
DCR  
    *See* device control register.  
DCU  
    *See* data cache.  
DEAR  
    *See* data exception-address register.  
debug  
    cache 504 to 505  
debug events  
    branch taken (BT) 582  
    cache-control instructions 588  
    DAC address-range match 587  
    DAC exact-address match 586

DAC exact-match granularity 586  
 DAC inclusive/exclusive ranges 588  
 data address-compare (DAC) 585  
 data value-compare (DVC) 589  
 DVC compare modes 590  
 DVC read/write events 591  
 exception taken (EDE) 582  
 IAC address-range match 584  
 IAC exact-address match 583  
 IAC inclusive/exclusive ranges 584  
 IAC range toggling 585  
 imprecise (IDE) 592  
 instruction address-compare (IAC) 583  
 instruction complete (IC) 581  
 resources used by 580  
 trap instruction (TDE) 582  
 unconditional (UDE) 583  
 debug exception 557, 580  
   disabled (pending) 592  
   trap instruction 413  
 debug modes  
   debug-wait mode 573, 580  
   external-debug mode 572, 580  
   internal-debug mode 572, 580  
   real-time trace mode 573, 580  
 debug-control registers 574 to 577  
 debug-status register 577 to 578  
 debug-wait mode  
   See debug modes.  
 defined instruction class 391  
 device control register 470  
   move instructions 472  
 dirty  
   See cache, dirty.  
 divide instructions 431  
 DTLB  
   See TLB, data shadow TLB.  
 DVCn  
   See data value-compare registers.  
 dynamic branch prediction 406

## E

effective address  
   See addressing, effective address.  
 effective page number 509  
 ESR  
   See exception-syndrome register.  
 EVPR  
   See exception-vector prefix register.  
 exception  
   See also interrupt.  
   alignment 546  
   APU unavailable 551  
   asynchronous 526  
   critical input 539  
   data storage 542  
   data TLB miss 555  
   debug 557  
   definition of 525  
   external 545  
   fixed-interval timer 553  
   FPU unavailable 549

identifying cause of 536 to 538  
 instruction storage 544  
 instruction TLB miss 556  
 machine check 540  
 partial instruction execution 526  
 persistent 532  
 program 547  
 programmable-interval timer 552  
 simultaneous 531  
 synchronous 526  
 system call 550  
 watchdog timer 554  
 exception taken (EDE)  
   See debug events.  
 exceptions  
   listing 527  
 exception-syndrome register 536 to 538  
   data TLB-miss exception 555  
   data-storage exception 543  
   instruction-storage exception 544  
   machine-check exception 540  
   program exception 548  
 exception-vector prefix register 536  
 execution model  
   See also synchronization.  
   sequential 377  
   speculative execution 377  
   weakly consistent 377  
 execution synchronization  
   See synchronization, execution.  
 extended arithmetic  
   addition 426  
   subtraction 428  
 extended mnemonics 857  
 external exception 545  
 external-debug mode  
   See debug modes.  
 extract instructions 865

## F

FIT exception 553  
 fixed-interval timer 553, 569  
   See also FIT exception.  
   disabling 569  
   enabling 569  
   FIT period 569  
 fixed-point exception register 399  
   carry (CA) 399  
   integer instruction update 425  
   overflow (OV) 399  
   summary overflow (SO) 399  
   transfer-byte count (TBC) 399, 424  
 floating-point emulation 458, 547  
 flow-control instructions 403  
 FPU-unavailable exception 549

## G

G storage attribute  
   See storage attribute, guarded.

general-purpose register 396  
 GPR  
   See general-purpose register.  
 guarded storage 544

## H

halfword, definition 383  
 Harvard cache model 473

## I

I storage attribute  
   See storage attribute, caching inhibited.  
 IACn  
   See instruction address-compare registers.  
 ICU  
   See instruction cache.  
 illegal instructions 392, 547  
 imprecise (IDE)  
   See debug events.  
 initialization requirements 599  
 insert instructions 865  
 instruction address-compare (IAC)  
   See debug events.  
 instruction address-compare registers 578  
 instruction cache  
   See also cache.  
   cacheable prefetch 496  
   control instructions 492  
   fetch without allocate 497  
   fill buffer 478  
   hint instruction 478  
   line buffer 477  
   non-cacheable prefetch 496  
   non-cacheable request size 497  
   operation 477 to 478  
   PLB priority 496, 497  
   self-modifying code 503  
   synonym 478  
 instruction complete (IC)  
   See debug events.  
 instruction forms 606  
 instruction relocate  
   See virtual mode.  
 instruction TLB-miss  
   exception 517, 556  
 instruction-cache cacheability  
   register 490  
 instruction-cache debug-data  
   register 504  
 instruction-storage exception 517, 544  
 internal-debug mode  
   See debug modes.  
 interrupt  
   See also exception.



definition of 525  
 imprecise 526  
 masking 532  
 precise 526  
 priority 531  
**interrupt handler** 525  
   base address 536  
   returning from 530 to 531  
   transferring control to 528 to 530  
**invalid instruction form** 391  
**ITLB**  
   See TLB, instruction shadow TLB.

## J

**JTAG connector** 593  
**JTAG debug port** 593

## L

**link register** 399  
   branch update 408, 409, 410, 411  
   branching to 405, 411  
   LK opcode field 408, 409, 410, 411  
   stack 407  
**little endian** 385  
   *See also* storage attribute, endian  
   byte-reverse instructions 388  
   data access 388  
   instruction fetch 387  
   operand alignment 389  
   PPC405 support 386 to 388  
**load address instruction** 870  
**load immediate instruction** 870  
**load instructions** 417  
   byte reverse 421  
   load and reserve 462  
   load byte and zero 417  
   load halfword algebraic 418  
   load halfword and zero 417  
   load multiple word 422, 526  
   load string 423, 526  
   load word and zero 418  
   partially executed 527  
**load multiple instructions**  
   *See* load instructions.  
**load word and reserve** 462  
**logical address**  
   *See* addressing, effective address.  
**logical instructions** 431 to 433  
**logical-comparison**  
   instructions 435  
**logical-shift instructions** 439  
**LR**  
   *See* link register.  
**LRU**  
   *See* cache, LRU.

## M

**M storage attribute**

*See* storage attribute, memory coherency.  
**MAC instructions** 441  
   cross halfword to word 442 to 444  
   high halfword to word 444 to 446  
   low halfword to word 446 to 449  
   negative cross halfword to word 449 to 451  
   negative high halfword to word 451 to 453  
   negative low halfword to word 453 to 455  
**machine-check exception** 540  
**machine-state register** 467  
   after an interrupt 533  
   APU-unavailable 551  
   critical-interrupt enable 539, 554  
   data relocate 508, 555  
   debug-interrupt enable 557  
   external-interrupt enable 545, 552, 553  
   FPU-unavailable 549  
   instruction relocate 508, 556  
   instructions 471  
   machine-check enable 540  
   reset state 598  
   wait-state enable 472  
**masking interrupts** 532  
**memory coherency** 484  
**memory management** 381  
**memory synchronization**  
   *See* synchronization, storage.  
**memory-control instructions** 463  
**modulo arithmetic** 441  
**most-recent reset** 597  
**move register instruction** 870  
**move to CR instruction** 871  
**MSR**  
   *See* machine-state register.

**multiply instructions**  
   cross halfword to word 455  
   high halfword to word 456  
   low halfword to word 457  
   word to word 430

## N

**negation instructions** 429  
**negative MAC instructions**  
   *See* MAC instructions.  
**noncritical exception** 528  
**no-operation instruction** 870

## O

**OEA**  
   *See* PowerPC.  
**operand alignment**  
   alignment exception 390, 546  
   definition 389  
   performance effects 389  
**optional instructions** 392

## P

**page translation**  
   page number 509  
   page-translation table 510 to 511  
   process ID 509  
**paging**  
   *See also* TLB.  
   and cache synonyms 479  
   executable pages 513, 518  
   no-access-allowed pages 518, 542, 544  
   non-executable pages 518, 544  
   page locking 510  
   page replacement 510  
   page size 513, 514  
   process protection 518  
   read-only pages 518, 542  
   recording accesses 523  
   recording changes 523  
   table walking 510  
   writable pages 513, 518  
**persistent exceptions** 532  
**physical memory** 381  
**physical-page number** 513  
**PID**  
   *See* process ID register.  
**pipeline stall** 482  
**PIT**  
   *See* programmable-interval timer.  
**PIT exception** 552  
**PLB-request priority** 497  
**PowerPC**  
   architecture components 359 to 360  
   Book-E architecture 365  
   embedded-environment  
     architecture 362 to 364  
   features not in architecture 361  
   latitude within the architecture 361  
   OEA 360, 364  
   UISA 360  
   VEA 360, 363  
**PPC405** 370 to 375  
   caches 373, 474 to 477  
   central-processing unit 371  
   debug resources 374  
   exception-handling logic 372  
   external interfaces 374  
   memory system 473 to 474  
   memory-management unit 372, 507  
   timers 373  
**preferred instruction form** 391  
**privileged instructions** 470, 547  
**privileged mode** 379  
**privileged registers** 465  
**problem state**  
   *See* user mode.  
**process ID** 509, 515  
**process ID register** 510  
**process tag** 513, 515  
**processor reset**  
   *See* reset.  
**processor version register** 469



processor-control instructions 458  
 program exception 547  
   system trap 413  
 programmable-interval timer 552,  
   568  
   *See also* PIT exception.  
   auto-reload mode 568  
   disabling 568  
   enabling 568  
   PIT register 563  
 PVR  
   *See* processor version register.

## R

Rc opcode field  
   *See* record bit.  
 real mode 383, 507  
   storage attribute control 488 to 492  
 real-time trace mode  
   *See* debug modes.  
 record bit 397, 426  
 registers  
   privileged registers 465  
   supported by PPC405 368  
   user registers 395  
 reservation 462  
 reserved instructions 392  
 reset 597  
   due to debug control 574  
   due to watchdog time-out 566  
   first instruction executed 599  
   processor state 597 to 599  
 return from interrupt 530 to 531  
 right rotation 435  
 rotate instructions 435, 865  
   AND mask instructions 436  
   mask generation 436  
   mask insert instructions 437  
 RPN  
   *See* physical-page number.

## S

saturating arithmetic 441  
 save/restore registers  
   SRR0 534  
   SRR1 534  
   SRR2 535  
   SRR3 535  
 sequential execution  
   *See* execution model.  
 shadow TLB  
   *See* TLB.  
 shift instructions 439, 865  
 sign-extension instructions 433  
 simplified mnemonics 857  
 single stepping  
   branches 582  
   exceptions 582  
   sequential 581

special-purpose register  
   CCR0 495  
   CTR 400  
   DACn 579  
   DBCR0 574  
   DBCR1 575  
   DCCR 490  
   DCWR 489  
   DEAR 538  
   DVCn 579  
   ESR 536 to 538  
   EVPR 536  
   IACn 578  
   ICCR 490  
   ICDBDR 504  
   LR 399  
   move instructions 460, 471  
   PID 510  
   PIT 563  
   privileged mode 467  
   PVR 469  
   SGR 491  
   SLER 491  
   SPRGn 401, 468  
   SRR0 534  
   SRR1 534  
   SRR2 535  
   SRR3 535  
   SUOR 491  
   TCR 564  
   TSR 565  
   user mode 396  
   USPRG0 400  
   XER 399  
   ZPR 519  
 speculative execution  
   *See* execution model.  
 split-field notation 607  
 SPR  
   *See* special-purpose register.  
 SPR general-purpose register  
   privileged mode 468  
   user mode 401  
 SPRGn  
   *See* SPR general-purpose register.  
 SRRn  
   *See* save/restore registers.  
 static branch prediction 406  
 storage attribute 487 to 488  
   caching inhibited 487, 514  
   endian 387, 488, 514  
   guarded 488, 514  
   in TLB entry 514  
   memory coherency 487, 514  
   real mode control 488 to 492  
   U0 exception 496, 542  
   user defined 488, 514  
   write through 487, 514  
 storage guarded register 491  
 storage little-endian register 491  
 storage synchronization  
   *See* synchronization, storage.  
 storage user-defined 0 register 491  
 store instructions 420  
   byte reverse 421

  partially executed 527  
   store byte 420  
   store conditional 462  
   store halfword 420  
   store multiple word 422, 527  
   store string 423, 527  
   store word 421  
 store multiple instructions  
   *See* store instructions.  
 store word conditional 462  
 string instructions  
   *See* load instructions.  
   *See* store instructions.  
 subtraction instructions 428 to 429  
 supervisor state  
   *See* privileged mode.  
 synchronization  
   context 378, 461  
   effect of instructions 461  
   execution 378, 461  
   semaphore 462  
   storage 379, 461, 484  
 synchronization instructions 460  
   eieio and sync implementation 461  
 synonym  
   *See* instruction cache, synonym.  
 system linkage instructions 470  
 system reset  
   *See* reset.  
 system-call exception 412, 550  
 system-call instruction 412, 550  
 system-trap instruction 413, 547  
   *See also* debug events.  
   TO opcode field 413

## T

tag  
   cache 475  
   TLB 513  
 TBH  
   *See* time base register.  
 TBL  
   *See* time base register.  
 TCR  
   *See* timer-control register.  
 TID  
   *See* process tag.  
 time base register 560 to 561  
   reading 561  
   user mode 401  
   writing 561  
 time-of-day computation 562  
 timer events 565  
 timer-control register 564  
   FIT-interrupt enable 553  
   PIT-interrupt enable 552  
   watchdog-interrupt enable 554  
 timer-status register 565  
 TLB 511 to 517  
   *See also* paging.  
   access 515 to 516

access failure 516 to 517  
 data shadow TLB 512  
 hit 515  
 instruction shadow TLB 511  
 maintaining shadow TLBs 523  
 miss 516  
 TLB-miss exceptions 517  
 unified TLB 511

TLB entry 512 to 514  
 access control 513, 518 to 519  
 executable 513  
 page size 514  
 physical page number 513  
 physical-page identification 513  
 storage attributes 514  
 TLBHI 513  
 TLBLO 513  
 valid 513  
 virtual-page identification 513  
 writable 513  
 zone selection 513

TLB-management instructions 521 to 522

trap instruction  
 causing debug event 582

trigger event 580

TSR  
 See timer-status register.

## U

U0 storage attribute  
 See storage attribute, user defined.

UISA  
 See PowerPC.

unconditional (UDE)  
 See debug events.

user mode 380  
 user registers 395  
 user-SPR general-purpose register 400

USPRG0  
 See user-SPR general-purpose register.

UTLB  
 See TLB, unified TLB.

## V

VEA  
 See PowerPC.

virtual memory 381  
 virtual mode 383, 507  
 virtual page number 509

## W

W storage attribute  
 See storage attribute, write through.  
 wait state 472  
 watchdog timer 566 to 568

disabling 568  
 enable next watchdog 566  
 enabling 566  
 interrupt status 567  
 reset control 566  
 state machine 567  
 using 567  
 watchdog period 566  
 watchdog-timer exception 554  
 weakly consistent  
 See execution model.  
 word, definition 383

## X

XER  
 See fixed-point exception register.

## Y

y bit  
 See branch prediction.

## Z

zone protection 518 to 519  
 data-storage exception 542  
 instruction-storage exception 544  
 TLB entry 513  
 zone-protection register 519  
 ZPR  
 See zone-protection register.



# **Volume 2(b): PPC405 Processor Block Manual**

***Virtex-II Pro™ Platform FPGA  
Developer's Kit***

March 2002 Release





The Xilinx logo shown above is a registered trademark of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

"Xilinx" and the Xilinx logo are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, XC5210 are registered Trademarks of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Bencher, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Bencher, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, Nano-Blaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, Rocket I/O, SelectI/O, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM      IBM Logo      PowerPC      PowerPC Logo      Blue Logic      CoreConnect      CodePack

All other trademarks are the property of their respective owners.

Xilinx does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 2002 Xilinx, Inc. All Rights Reserved.

## About This Book

---

This document serves as a technical reference describing the hardware interface to the PowerPC™ PPC405x3 processor block. It contains information on input/output signals, timing relationships between signals, and the mechanisms software can use to control the interface operation. The document is intended for use by FPGA and system hardware designers and by system programmers who need to understand how certain operations affect hardware external to the processor.

### Document Organization

- , provides an overview of the PowerPC embedded-environment architecture and the features supported by the PPC405x3.
- **Chapter 2, Input/Output Interfaces**, describes the interface signals into and out of the PPC405x3 processor block. Where appropriate, timing diagrams are provided to assist in understanding the functional relationship between multiple signals.
- **Chapter 3, PowerPC® 405 OCM Controller**, describes the features, interface signals, timing specifications, and programming model for the PPC405x3 on-chip memory (OCM) controller. The OCM controller serves as a dedicated interface between the block RAMs in the FPGA and OCM signals available on the embedded PPC405x3 core.
- **Appendix A, RISCWatch and RISCTrace Interfaces**, describes the interface requirements between the PPC405x3 processor block and the RISCWatch and RISCTrace tools.
- **Appendix B, Signal Summary**, lists all PPC405x3 interface signals in alphabetical order.

### Document Conventions

#### General Conventions

**Table 3-1** lists the general notational conventions used throughout this document.

**Table 3-1: General Notational Conventions**

| Convention       | Definition                                          |
|------------------|-----------------------------------------------------|
| <b>mnemonic</b>  | Instruction mnemonics are shown in lower-case bold. |
| <i>variable</i>  | Variable items are shown in italic.                 |
| <u>ActiveLow</u> | An overbar indicates an active-low signal.          |
| <i>n</i>         | A decimal number.                                   |

Table 3-1: General Notational Conventions (Continued)

| Convention                 | Definition                                                                                                                |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------|
| $0xn$                      | A hexadecimal number.                                                                                                     |
| $0bn$                      | A binary number.                                                                                                          |
| $OBJECT_b$                 | A single bit in any object (a register, an instruction, an address, or a field) is shown as a subscripted number or name. |
| $OBJECT_{b:b}$             | A range of bits in any object (a register, an instruction, an address, or a field).                                       |
| $OBJECT_{b,b,\dots}$       | A list of bits in any object (a register, an instruction, an address, or a field).                                        |
| REGISTER[FIELD]            | Fields within any register are shown in square brackets.                                                                  |
| REGISTER[FIELD, FIELD ...] | A list of fields in any register.                                                                                         |
| REGISTER[FIELD:FIELD]      | A range of fields in any register.                                                                                        |

## Registers

Table 3-2 lists the PPC405x3 registers used in this document and their descriptive names.

Table 3-2: PPC405x3 Registers

| Register | Descriptive Name              |
|----------|-------------------------------|
| CCR0     | Core-configuration register 0 |
| DBCR $n$ | Debug-control register $n$    |
| DBSR     | Debug-status register         |
| ESR      | Exception-syndrome register   |
| MSR      | Machine-state register        |
| PIT      | Programmable-interval timer   |
| TBL      | Time-base lower               |
| TBU      | Time-base upper               |
| TCR      | Timer-control register        |
| TSR      | Timer-status register         |

## Terms

### *active*

As applied to signals, this term indicates a signal is in a state that causes an action to occur in the receiving device, or indicates an action occurred in the sending device. An *active-high* signal drives a logic 1 when active. An *active-low* signal drives a logic 0 when active.

|                          |                                                                                                                                                                                                                                                    |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>assert</i>            | As applied to signals, this term indicates a signal is driven to its active state.                                                                                                                                                                 |
| <i>atomic access</i>     | A memory access that attempts to read from and write to the same address uninterrupted by other accesses to that address. The term refers to the fact that such transactions are indivisible.                                                      |
| <i>big endian</i>        | A memory byte ordering where the address of an item corresponds to the most-significant byte.                                                                                                                                                      |
| <i>Book-E</i>            | An version of the PowerPC architecture designed specifically for embedded applications.                                                                                                                                                            |
| <i>cache block</i>       | Synonym for <i>cache line</i> .                                                                                                                                                                                                                    |
| <i>cache line</i>        | A portion of a cache array that contains a copy of contiguous system-memory addresses. Cache lines are 32-bytes long and aligned on a 32-byte address.                                                                                             |
| <i>cache set</i>         | Synonym for <i>congruence class</i> .                                                                                                                                                                                                              |
| <i>clear</i>             | To write a bit value of 0.                                                                                                                                                                                                                         |
| <i>clock</i>             | Unless otherwise specified, this term refers to the PPC405x3 processor clock.                                                                                                                                                                      |
| <i>congruence class</i>  | A collection of cache lines with the same index.                                                                                                                                                                                                   |
| <i>cycle</i>             | The time between two successive rising edges of the associated clock.                                                                                                                                                                              |
| <i>dead cycle</i>        | A cycle in which no useful activity occurs on the associated interface.                                                                                                                                                                            |
| <i>deassert</i>          | As applied to signals, this term indicates a signal is driven to its inactive state.                                                                                                                                                               |
| <i>dirty</i>             | An indication that cache information is more recent than the copy in memory.                                                                                                                                                                       |
| <i>doubleword</i>        | Eight bytes, or 64 bits.                                                                                                                                                                                                                           |
| <i>effective address</i> | The untranslated memory address as seen by a program.                                                                                                                                                                                              |
| <i>exception</i>         | An abnormal event or condition that requires the processor's attention. They can be caused by instruction execution or an external device. The processor records the occurrence of an exception and they often cause an <i>interrupt</i> to occur. |
| <i>fill buffer</i>       | A buffer that receives and sends data and instructions between the processor and PLB. It is used when cache misses occur and when access to non-cacheable memory occurs.                                                                           |
| <i>flush</i>             | A cache operation that involves writing back a modified entry to memory, followed by an invalidation of the entry.                                                                                                                                 |
| <i>GB</i>                | Gigabyte, or one-billion bytes.                                                                                                                                                                                                                    |
| <i>halfword</i>          | Two bytes, or 16 bits.                                                                                                                                                                                                                             |
| <i>hit</i>               | An indication that requested information exists in the accessed cache array, the associated fill buffer, or on the corresponding OCM interface.                                                                                                    |



|                         |                                                                                                                                                                                                                                                                             |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>inactive</i>         | As applied to signals, this term indicates a signal is in a state that does not cause an action to occur, nor does it indicate an action occurred. An <i>active-high</i> signal drives a logic 0 when inactive. An <i>active-low</i> signal drives a logic 1 when inactive. |
| <i>interrupt</i>        | The process of stopping the currently executing program so that an exception can be handled.                                                                                                                                                                                |
| <i>invalidate</i>       | A cache or TLB operation that causes an entry to be marked as invalid. An invalid entry can be subsequently replaced.                                                                                                                                                       |
| <b>KB</b>               | Kilobyte, or one-thousand bytes.                                                                                                                                                                                                                                            |
| <i>line buffer</i>      | A buffer located in the cache array that can temporarily hold the contents of an entire cache line. It is loaded with the contents of a cache line when a cache hit occurs.                                                                                                 |
| <i>line fill</i>        | A transfer of the contents of the instruction or data line buffer into the appropriate cache.                                                                                                                                                                               |
| <i>line transfer</i>    | A transfer of an aligned, sequentially addressed 4-word or 8-word quantity (instructions or data) across the PLB interface. The transfer can be from the PLB slave (read) or to the PLB slave (write).                                                                      |
| <i>little endian</i>    | A memory byte ordering where the address of an item corresponds to the least-significant byte.                                                                                                                                                                              |
| <i>logical address</i>  | Synonym for <i>effective address</i> .                                                                                                                                                                                                                                      |
| <b>MB</b>               | Megabyte, or one-million bytes.                                                                                                                                                                                                                                             |
| <i>memory</i>           | Collectively, cache memory and system memory.                                                                                                                                                                                                                               |
| <i>miss</i>             | An indication that requested information does not exist in the accessed cache array, the associated fill buffer, or on the corresponding OCM interface.                                                                                                                     |
| <b>OEA</b>              | The PowerPC operating-environment architecture, which defines the memory-management model, supervisor-level registers and instructions, synchronization requirements, the exception model, and the time-base resources as seen by supervisor programs.                      |
| <i>on chip</i>          | In system-on-chip implementations, this indicates on the same FPGA chip as the processor core, but external to the processor core.                                                                                                                                          |
| <i>pending</i>          | As applied to interrupts, this indicates that an exception occurred, but the interrupt is disabled. The interrupt occurs when it is later enabled.                                                                                                                          |
| <i>physical address</i> | The address used to access physically-implemented memory. This address can be translated from the effective address. When address translation is not used, this address is equal to the effective address.                                                                  |
| <b>PLB</b>              | Processor local bus.                                                                                                                                                                                                                                                        |
| <i>privileged mode</i>  | The operating mode typically used by system software. Privileged operations are allowed and software can access all registers and memory.                                                                                                                                   |

|                         |                                                                                                                                                                                                                             |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>problem state</i>    | Synonym for <i>user mode</i> .                                                                                                                                                                                              |
| <i>process</i>          | A program (or portion of a program) and any data required for the program to run.                                                                                                                                           |
| <i>real address</i>     | Synonym for <i>physical address</i> .                                                                                                                                                                                       |
| <i>scalar</i>           | Individual data objects and instructions. Scalars are of arbitrary size.                                                                                                                                                    |
| <i>set</i>              | To write a bit value of 1.                                                                                                                                                                                                  |
| <i>sleep</i>            | A state in which the PPC405x3 processor clock is prevented from toggling. The execution state of the PPC405x3 does not change when in the sleep state.                                                                      |
| <i>sticky</i>           | A bit that can be set by software, but cleared only by the processor. Alternatively, a bit that can be cleared by software, but set only by the processor.                                                                  |
| <i>string</i>           | A sequence of consecutive bytes.                                                                                                                                                                                            |
| <i>supervisor state</i> | Synonym for <i>privileged mode</i> .                                                                                                                                                                                        |
| <i>system memory</i>    | Physical memory installed in a computer system external to the processor core, such RAM, ROM, and flash.                                                                                                                    |
| <i>tag</i>              | As applied to caches, a set of address bits used to uniquely identify a specific cache line within a congruence class. As applied to TLBs, a set of address bits used to uniquely identify a specific entry within the TLB. |
| <i>UIISA</i>            | The PowerPC user instruction-set architecture, which defines the base user-level instruction set, registers, data types, the memory model, the programming model, and the exception model as seen by user programs.         |
| <i>user mode</i>        | The operating mode typically used by application software. Privileged operations are not allowed in user mode, and software can access a restricted set of registers and memory.                                            |
| <i>VEA</i>              | The PowerPC virtual-environment architecture, which defines a multi-access memory model, the cache model, cache-control instructions, and the time-base resources as seen by user programs.                                 |
| <i>virtual address</i>  | An intermediate address used to translate an effective address into a physical address. It consists of a process ID and the effective address. It is only used when address translation is enabled.                         |
| <i>wake up</i>          | The transition of the PPC405x3 out of the sleep state. The PPC405x3 processor clock begins toggling and the execution state of the PPC405x3 advances from that of the sleep state.                                          |
| <i>word</i>             | Four bytes, or 32 bits.                                                                                                                                                                                                     |

## Additional Reading

The following documents contain additional information of potential interest to readers of this manual:

- XILINX *PPC405 User Manual*

- XILINX *Virtex-II Pro Platform FPGA Handbook*

# Introduction to the PowerPC® 405 Processor

---

The PPC405 is a 32-bit implementation of the *PowerPC™ embedded-environment architecture* that is derived from the PowerPC architecture. Specifically, the PPC405 is an embedded PowerPC 405D5 processor core (PPC405D5). The term *processor block* is used throughout this document to refer to the combination of PPC405D5 core, on-chip memory logic (OCM), and the gasket logic and interface.

The PowerPC architecture provides a software model that ensures compatibility between implementations of the PowerPC family of microprocessors. The PowerPC architecture defines parameters that guarantee compatible processor implementations at the application-program level, allowing broad flexibility in the development of derivative PowerPC implementations that meet specific market requirements.

This chapter provides an overview of the PowerPC architecture and an introduction to the features of the PPC405 core.

## PowerPC Architecture

The PowerPC architecture is a 64-bit architecture with a 32-bit subset. The various features of the PowerPC architecture are defined at three levels. This layering provides flexibility by allowing degrees of software compatibility across a wide range of implementations. For example, an implementation such as an embedded controller can support the user instruction set, but not the memory management, exception, and cache models where it might be impractical to do so.

The three levels of the PowerPC architecture are defined in [Table 1-1](#).

Table 1-1: Three Levels of PowerPC Architecture

| User Instruction-Set Architecture (UISA)                                                                                                                                                                                                                                                                                                                                                                                                               | Virtual Environment Architecture (VEA)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | Operating Environment Architecture (OEA)                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>Defines the architecture level to which user-level (sometimes referred to as problem state) software should conform</li> <li>Defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions, exception model as seen by user programs, memory model, and the programming model</li> </ul> <p><b>Note:</b> All PowerPC implementations adhere to the UISA.</p> | <ul style="list-style-type: none"> <li>Defines additional user-level functionality that falls outside typical user-level software requirements</li> <li>Describes the memory model for an environment in which multiple devices can access memory</li> <li>Defines aspects of the cache model and cache-control instructions</li> <li>Defines the time-base resources from a user-level perspective</li> </ul> <p><b>Note:</b> Implementations that conform to the VEA level are guaranteed to conform to the UISA level.</p> | <ul style="list-style-type: none"> <li>Defines supervisor-level resources typically required by an operating system</li> <li>Defines the memory-management model, supervisor-level registers, synchronization requirements, and the exception model</li> <li>Defines the time-base resources from a supervisor-level perspective</li> </ul> <p><b>Note:</b> Implementations that conform to the OEA level are guaranteed to conform to the UISA and VEA levels.</p> |

The PowerPC architecture requires that all PowerPC implementations adhere to the UISA, offering compatibility among all PowerPC application programs. However, different versions of the VEA and OEA are permitted.

Embedded applications written for the PPC405 are compatible with other PowerPC implementations. Privileged software generally is not compatible. The migration of privileged software from the PowerPC architecture to the PPC405 is in many cases straightforward because of the simplifications made by the PowerPC embedded-environment architecture. Refer to *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*) for more information on programming the PPC405.

## PowerPC Embedded-Environment Architecture

The PPC405 is an implementation of the PowerPC embedded-environment architecture. This architecture is optimized for embedded controllers and is a forerunner to the PowerPC Book-E architecture. The PowerPC embedded-environment architecture provides an alternative definition for certain features specified by the PowerPC VEA and OEA. Implementations that adhere to the PowerPC embedded-environment architecture also adhere to the PowerPC UISA. PowerPC embedded-environment processors are 32-bit only implementations and thus do not include the special 64-bit extensions to the PowerPC UISA. Also, floating-point support can be provided either in hardware or software by PowerPC embedded-environment processors.

The following are features of the PowerPC embedded-environment architecture:

- Memory management optimized for embedded software environments.
- Cache-management instructions for optimizing performance and memory control in complex applications that are graphically and numerically intensive.
- Storage attributes for controlling memory-system behavior.
- Special-purpose registers for controlling the use of debug resources, timer resources, interrupts, real-mode storage attributes, memory-management facilities, and other architected processor resources.
- A device-control-register address space for managing on-chip peripherals such as memory controllers.

- A dual-level interrupt structure and interrupt-control instructions.
- Multiple timer resources.
- Debug resources that enable hardware-debug and software-debug functions such as instruction breakpoints, data breakpoints, and program single-stepping.

## Virtual Environment

The virtual environment defines architectural features that enable application programs to create or modify code, to manage storage coherency, and to optimize memory-access performance. It defines the cache and memory models, the timekeeping resources from a user perspective, and resources that are accessible in user mode but are primarily used by system-library routines. The following summarizes the virtual-environment features of the PowerPC embedded-environment architecture:

- Storage model:
  - Storage-control instructions as defined in the PowerPC virtual-environment architecture. These instructions are used to manage instruction caches and data caches, and for synchronizing and ordering instruction execution.
  - Storage attributes for controlling memory-system behavior. These are: write-through, cacheability, memory coherence (optional), guarded, and endian.
  - Operand-placement requirements and their effect on performance.
- The time-base function as defined by the PowerPC virtual-environment architecture, for user-mode read access to the 64-bit time base.

## Operating Environment

The operating environment describes features of the architecture that enable operating systems to allocate and manage storage, to handle errors encountered by application programs, to support I/O devices, and to provide operating-system services. It specifies the resources and mechanisms that require privileged access, including the memory-protection and address-translation mechanisms, the exception-handling model, and privileged timer resources. [Table 1-2](#) summarizes the operating-environment features of the PowerPC embedded-environment architecture.

**Table 1-2: OEA Features of the PowerPC Embedded-Environment Architecture**

| Operating Environment | Features                                                                                                                                                                                                                                                                                                                                                                                             |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Register model        | <ul style="list-style-type: none"> <li>• Privileged special-purpose registers (SPRs) and instructions for accessing those registers</li> <li>• Device control registers (DCRs) and instructions for accessing those registers</li> </ul>                                                                                                                                                             |
| Storage model         | <ul style="list-style-type: none"> <li>• Privileged cache-management instructions</li> <li>• Storage-attribute controls</li> <li>• Address translation and memory protection</li> <li>• Privileged TLB-management instructions</li> </ul>                                                                                                                                                            |
| Exception model       | <ul style="list-style-type: none"> <li>• Dual-level interrupt structure supporting various exception types</li> <li>• Specification of interrupt priorities and masking</li> <li>• Privileged SPRs for controlling and handling exceptions</li> <li>• Interrupt-control instructions</li> <li>• Specification of how partially executed instructions are handled when an interrupt occurs</li> </ul> |

Table 1-2: OEA Features of the PowerPC Embedded-Environment Architecture (Continued)

| Operating Environment                 | Features                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Debug model                           | <ul style="list-style-type: none"> <li>Privileged SPRs for controlling debug modes and debug events</li> <li>Specification for seven types of debug events</li> <li>Specification for allowing a debug event to cause a reset</li> <li>The ability of the debug mechanism to freeze the timer resources</li> </ul>                                                                                                                                                     |
| Time-keeping model                    | <ul style="list-style-type: none"> <li>64-bit time base</li> <li>32-bit decremter (the programmable-interval timer)</li> <li>Three timer-event interrupts: <ul style="list-style-type: none"> <li>Programmable-interval timer (PIT)</li> <li>Fixed-interval timer (FIT)</li> <li>Watchdog timer (WDT)</li> </ul> </li> <li>Privileged SPRs for controlling the timer resources</li> <li>The ability to freeze the timer resources using the debug mechanism</li> </ul> |
| Synchronization requirements          | <ul style="list-style-type: none"> <li>Requirements for special registers and the TLB</li> <li>Requirements for instruction fetch and for data access</li> <li>Specifications for context synchronization and execution synchronization</li> </ul>                                                                                                                                                                                                                     |
| Reset and initialization requirements | <ul style="list-style-type: none"> <li>Specification for two internal mechanisms that can cause a reset: <ul style="list-style-type: none"> <li>Debug-control register (DBCR)</li> <li>Timer-control register (TCR)</li> </ul> </li> <li>Contents of processor resources after a reset</li> <li>The software-initialization requirements, including an initialization code example</li> </ul>                                                                          |

## PPC405 Software Features

The PPC405 processor core is an implementation of the PowerPC embedded-environment architecture. The processor provides fixed-point embedded applications with high performance at low power consumption. It is compatible with the PowerPC UISA. Much of the PPC405 VEA and OEA support is also available in implementations of the PowerPC Book-E architecture. Key software features of the PPC405 include:

- A fixed-point execution unit fully compliant with the PowerPC UISA:
  - 32-bit architecture, containing thirty-two 32-bit general purpose registers (GPRs).
- PowerPC embedded-environment architecture extensions providing additional support for embedded-systems applications:
  - True little-endian operation
  - Flexible memory management
  - Multiply-accumulate instructions for computationally intensive applications
  - Enhanced debug capabilities
  - 64-bit time base
  - 3 timers: programmable interval timer (PIT), fixed interval timer (FIT), and watchdog timer (all are synchronous with the time base)
- Performance-enhancing features, including:
  - Static branch prediction
  - Five-stage pipeline with single-cycle execution of most instructions, including loads and stores
  - Multiply-accumulate instructions

- Hardware multiply/divide for faster integer arithmetic (4-cycle multiply, 35-cycle divide)
- Enhanced string and multiple-word handling
- Support for unaligned loads and unaligned stores to cache arrays, main memory, and on-chip memory (OCM)
- Minimized interrupt latency
- Integrated instruction-cache:
  - 16 KB, 2-way set associative
  - Eight words (32 bytes) per cache line
  - Fetch line buffer
  - Instruction-fetch hits are supplied from the fetch line buffer
  - Programmable prefetch of next-sequential line into the fetch line buffer
  - Programmable prefetch of non-cacheable instructions: full line (eight words) or half line (four words)
  - Non-blocking during fetch line fills
- Integrated data-cache:
  - 16 KB, 2-way set associative
  - Eight words (32 bytes) per cache line
  - Read and write line buffers
  - Load and store hits are supplied from/to the line buffers
  - Write-back and write-through support
  - Programmable load and store cache line allocation
  - Operand forwarding during cache line fills
  - Non-blocking during cache line fills and flushes
- Support for on-chip memory (OCM) that can provide memory-access performance identical to a cache hit
- Flexible memory management:
  - Translation of the 4 GB logical-address space into the physical-address space
  - Independent control over instruction translation and protection, and data translation and protection
  - Page-level access control using the translation mechanism
  - Software control over the page-replacement strategy
  - Write-through, cacheability, user-defined 0, guarded, and endian (WIU0GE) storage-attribute control for each virtual-memory region
  - WIU0GE storage-attribute control for thirty-two 128 MB regions in real mode
  - Additional protection control using zones
- Enhanced debug support with logical operators:
  - Four instruction-address compares
  - Two data-address compares
  - Two data-value compares
  - JTAG instruction for writing into the instruction cache
  - Forward and backward instruction tracing
- Advanced power management support

The following sections describe the software resources available in the PPC405. Refer to the *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*) for more information on using these resources.



## Privilege Modes

Software running on the PPC405 can do so in one of two privilege modes: privileged and user.

### Privileged Mode

*Privileged mode* allows programs to access all registers and execute all instructions supported by the processor. Normally, the operating system and low-level device drivers operate in this mode.

### User Mode

*User mode* restricts access to some registers and instructions. Normally, application programs operate in this mode.

## Address Translation Modes

The PPC405 also supports two modes of address translation: real and virtual.

### Real Mode

In *real mode*, programs address physical memory directly.

### Virtual Mode

In *virtual mode*, programs address virtual memory and virtual-memory addresses are translated by the processor into physical-memory addresses. This allows programs to access much larger address spaces than might be implemented in the system.

## Addressing Modes

Whether the PPC405 is running in real mode or virtual mode, data addressing is supported by the load and store instructions using one of the following addressing modes:

- Register-indirect with immediate index—A base address is stored in a register, and a displacement from the base address is specified as an immediate value in the instruction.
- Register-indirect with index—A base address is stored in a register, and a displacement from the base address is stored in a second register.
- Register indirect—The data address is stored in a register.

Instructions that use the two indexed forms of addressing also allow for automatic updates to the base-address register. With these instruction forms, the new data address is calculated, used in the load or store data access, and stored in the base-address register.

With sequential-instruction execution, the next-instruction address is calculated by adding four bytes to the current-instruction address. In the case of branch instructions, however, the next-instruction address is determined using one of four branch-addressing modes:

- Branch to relative—The next-instruction address is at a location relative to the current-instruction address.
- Branch to absolute—The next-instruction address is at an absolute location in memory.
- Branch to link register—The next-instruction address is stored in the link register.
- Branch to count register—The next-instruction address is stored in the count register.

## Data Types

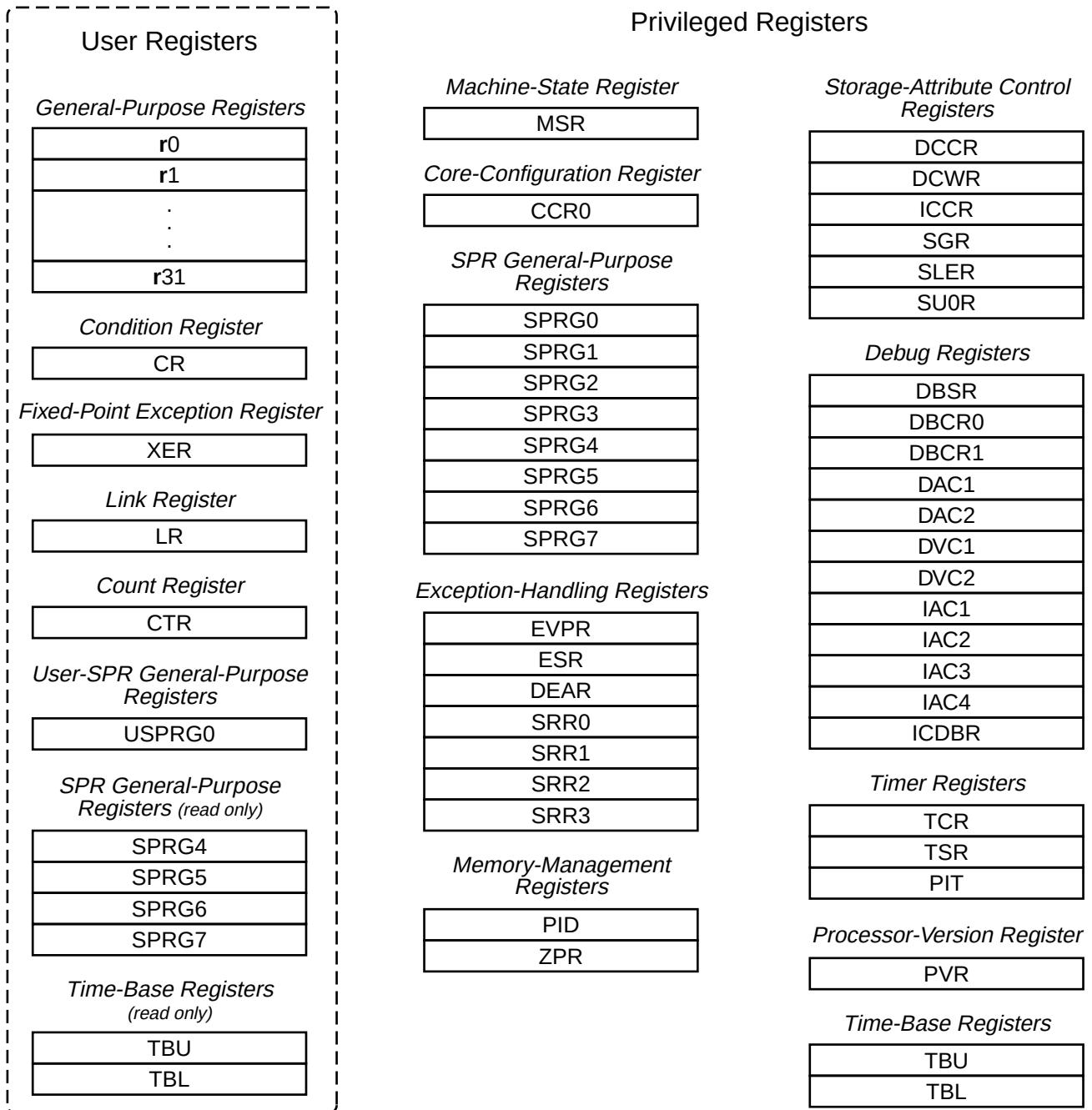
PPC405 instructions support byte, halfword, and word operands. Multiple-word operands are supported by the load/store multiple instructions and byte strings are supported by

the load/store string instructions. Integer data are either signed or unsigned, and signed data is represented using two's-complement format.

The address of a multi-byte operand is determined using the lowest memory address occupied by that operand. For example, if the four bytes in a word operand occupy addresses 4, 5, 6, and 7, the word address is 4. The PPC405 supports both big-endian (an operand's *most significant* byte is at the lowest memory address) and little-endian (an operand's *least significant* byte is at the lowest memory address) addressing.

## Register Set Summary

**Figure 1-1** shows the registers contained in the PPC405. Descriptions of the registers are in the following sections.



UG018\_36\_102401

Figure 1-1: PPC405 Registers

## General-Purpose Registers

The processor contains thirty-two 32-bit *general-purpose registers* (GPRs), identified as r0 through r31. The contents of the GPRs are read from memory using load instructions and written to memory using store instructions. Computational instructions often read operands from the GPRs and write their results in GPRs. Other instructions move data between the GPRs and other registers. GPRs can be accessed by all software.

## Special-Purpose Registers

The processor contains a number of 32-bit *special-purpose registers* (SPRs). SPRs provide access to additional processor resources, such as the count register, the link register, debug resources, timers, interrupt registers, and others. Most SPRs are accessed only by privileged software, but a few, such as the count register and link register, are accessed by all software.

## Machine-State Register

The 32-bit *machine-state register* (MSR) contains fields that control the operating state of the processor. This register can be accessed only by privileged software.

## Condition Register

The 32-bit *condition register* (CR) contains eight 4-bit fields, CR0–CR7. The values in the CR fields can be used to control conditional branching. Arithmetic instructions can set CR0 and compare instructions can set any CR field. Additional instructions are provided to perform logical operations and tests on CR fields and bits within the fields. The CR can be accessed by all software.

## Device Control Registers

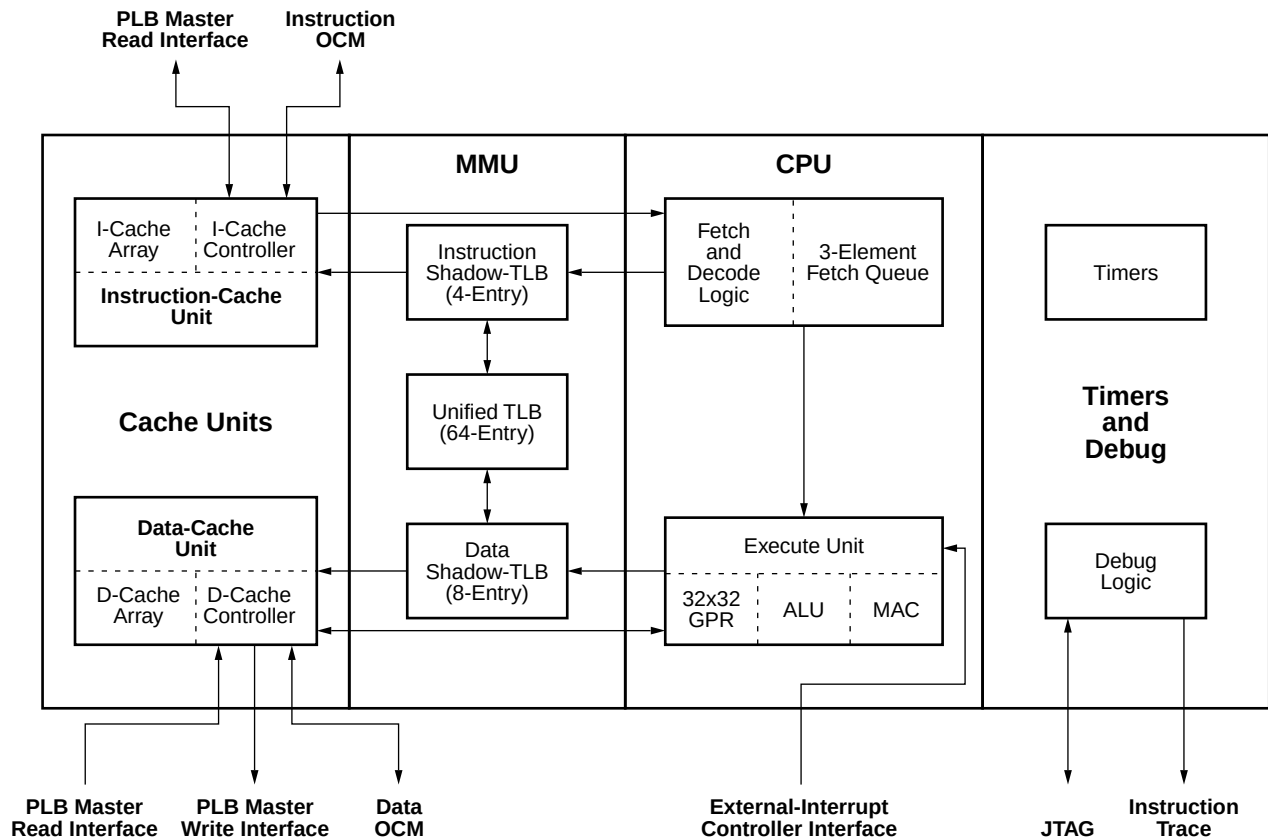
The 32-bit *device control registers* (not shown) are used to configure, control, and report status for various external devices that are not part of the PPC405 processor. The OCM controllers are examples of devices that contain DCRs. Although the DCRs are not part of the PPC405 implementation, they are accessed using the **mtdcr** and **mfdcr** instructions. The DCRs can be accessed only by privileged software.

# PPC405 Hardware Organization

As shown in [Figure 1-2](#), the PPC405 processor contains the following elements:

- A 5-stage pipeline consisting of fetch, decode, execute, write-back, and load write-back stages
- A virtual-memory-management unit that supports multiple page sizes and a variety of storage-protection attributes and access-control options
- Separate instruction-cache and data-cache units
- Debug support, including a JTAG interface
- Three programmable timers

The following sections provide an overview of each element. Refer to the *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*) for more information on how software interacts with these elements.



UG018\_35\_102401

Figure 1-2: PPC405 Organization

## Central-Processing Unit

The PPC405 central-processing unit (CPU) implements a 5-stage instruction pipeline consisting of fetch, decode, execute, write-back, and load write-back stages.

The fetch and decode logic sends a steady flow of instructions to the execute unit. All instructions are decoded before they are forwarded to the execute unit. Instructions are queued in the fetch queue if execution stalls. The fetch queue consists of three elements: two prefetch buffers and a decode buffer. If the prefetch buffers are empty instructions flow directly to the decode buffer.

Up to two branches are processed simultaneously by the fetch and decode logic. If a branch cannot be resolved prior to execution, the fetch and decode logic predicts how that branch is resolved, causing the processor to speculatively fetch instructions from the predicted path. Branches with negative-address displacements are predicted as taken, as are branches that do not test the condition register or count register. The default prediction can be overridden by software at assembly or compile time.

The PPC405 has a single-issue execute unit containing the general-purpose register file (GPR), arithmetic-logic unit (ALU), and the multiply-accumulate unit (MAC). The GPRs consist of thirty-two 32-bit registers that are accessed by the execute unit using three read ports and two write ports. During the decode stage, data is read out of the GPRs for use by the execute unit. During the write-back stage, results are written to the GPR. The use of five read/write ports on the GPRs allows the processor to execute load/store operations in parallel with ALU and MAC operations.

The execute unit supports all 32-bit PowerPC UISA integer instructions in hardware, and is compliant with the PowerPC embedded-environment architecture specification. Floating-point operations are not supported.

The MAC unit supports implementation-specific multiply-accumulate instructions and multiply-halfword instructions. MAC instructions operate on either signed or unsigned 16-bit operands, and they store their results in a 32-bit GPR. These instructions can produce results using either modulo arithmetic or saturating arithmetic. All MAC instructions have a single cycle throughput.

## Exception Handling Logic

Exceptions are divided into two classes: critical and noncritical. The PPC405 CPU services exceptions caused by error conditions, the internal timers, debug events, and the external interrupt controller (EIC) interface. Across the two classes, a total of 19 possible exceptions are supported, including the two provided by the EIC interface.

Each exception class has its own pair of save/restore registers. SRR0 and SRR1 are used for noncritical interrupts, and SRR2 and SRR3 are used for critical interrupts. The exception-return address and the machine state are written to these registers when an exception occurs, and they are automatically restored when an interrupt handler exits using the return-from-interrupt (**rfi**) or return-from critical-interrupt (**rfci**) instruction. Use of separate save/restore registers allows the PPC405 to handle critical interrupts independently of noncritical interrupts.

## Memory Management Unit

The PPC405 supports 4 GB of flat (non-segmented) address space. The memory-management unit (MMU) provides address translation, protection functions, and storage-attribute control for this address space. The MMU supports demand-paged virtual memory using multiple page sizes of 1 KB, 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB and 16 MB. Multiple page sizes can improve memory efficiency and minimize the number of TLB misses. When supported by system software, the MMU provides the following functions:

- Translation of the 4 GB logical-address space into a physical-address space.
- Independent enabling of instruction translation and protection from that of data translation and protection.
- Page-level access control using the translation mechanism.
- Software control over the page-replacement strategy.
- Additional protection control using zones.
- Storage attributes for cache policy and speculative memory-access control.

The translation look-aside buffer (TLB) is used to control memory translation and protection. Each one of its 64 entries specifies a page translation. It is fully associative, and can simultaneously hold translations for any combination of page sizes. To prevent TLB contention between data and instruction accesses, a 4-entry instruction and an 8-entry data shadow-TLB are maintained by the processor transparently to software.

Software manages the initialization and replacement of TLB entries. The PPC405 includes instructions for managing TLB entries by software running in privileged mode. This capability gives significant control to system software over the implementation of a page replacement strategy. For example, software can reduce the potential for TLB thrashing or delays associated with TLB-entry replacement by reserving a subset of TLB entries for globally accessible pages or critical pages.

Storage attributes are provided to control access of memory regions. When memory translation is enabled, storage attributes are maintained on a page basis and read from the TLB when a memory access occurs. When memory translation is disabled, storage attributes are maintained in storage-attribute control registers. A zone-protection register

(ZPR) is provided to allow system software to override the TLB access controls without requiring the manipulation of individual TLB entries. For example, the ZPR can provide a simple method for denying read access to certain application programs.

## Instruction and Data Caches

The PPC405 accesses memory through the instruction-cache unit (ICU) and data-cache unit (DCU). Each cache unit includes a PLB-master interface, cache arrays, and a cache controller. Hits into the instruction cache and data cache appear to the CPU as single-cycle memory accesses. Cache misses are handled as requests over the PLB bus to another PLB device, such as an external-memory controller.

The PPC405 implements separate instruction-cache and data-cache arrays. Each is 16 KB in size, is two-way set-associative, and operates using 8-word (32 byte) cache lines. The caches are non-blocking, allowing the PPC405 to overlap instruction execution with reads over the PLB (when cache misses occur).

The cache controllers replace cache lines according to a least-recently used (LRU) replacement policy. When a cache line fill occurs, the most-recently accessed line in the cache set is retained and the other line is replaced. The cache controller updates the LRU during a cache line fill.

The ICU supplies up to two instructions every cycle to the fetch and decode unit. The ICU can also forward instructions to the fetch and decode unit during a cache line fill, minimizing execution stalls caused by instruction-cache misses. When the ICU is accessed, four instructions are read from the appropriate cache line and placed temporarily in a line buffer. Subsequent ICU accesses check this line buffer for the requested instruction prior to accessing the cache array. This allows the ICU cache array to be accessed as little as once every four instructions, significantly reducing ICU power consumption.

The DCU can independently process load/store operations and cache-control instructions. The DCU can also dynamically reprioritize PLB requests to reduce the length of an execution stall. For example, if the DCU is busy with a low-priority request and a subsequent storage operation requested by the CPU is stalled, the DCU automatically increases the priority of the current (low-priority) request. The current request is thus finished sooner, allowing the DCU to process the stalled request sooner. The DCU can forward data to the execute unit during a cache line fill, further minimizing execution stalls caused by data-cache misses.

Additional features allow programmers to tailor data-cache performance to a specific application. The DCU can function in write-back or write-through mode, as determined by the storage-control attributes. Loads and stores that do not allocate cache lines can also be specified. Inhibiting certain cache line fills can reduce potential pipeline stalls and unwanted external-bus traffic.

## Timer Resources

The PPC405 contains a 64-bit time base and three timers. The time base is incremented synchronously using the CPU clock or an external clock source. The three timers are incremented synchronously with the time base. The three timers supported by the PPC405 are:

- Programmable Interval Timer
- Fixed Interval Timer
- Watchdog Timer

### Programmable Interval Timer

The *programmable interval timer* (PIT) is a 32-bit register that is decremented at the time-base increment frequency. The PIT register is loaded with a delay value. When the PIT count reaches 0, a PIT interrupt occurs. Optionally, the PIT can be programmed to automatically reload the last delay value and begin decrementing again.



## Fixed Interval Timer

The *fixed interval timer* (FIT) causes an interrupt when a selected bit in the time-base register changes from 0 to 1. Programmers can select one of four predefined bits in the time-base for triggering a FIT interrupt.

## Watchdog Timer

The *watchdog timer* causes a hardware reset when a selected bit in the time-base register changes from 0 to 1. Programmers can select one of four predefined bits in the time-base for triggering a reset, and the type of reset can be defined by the programmer.

## Debug

The PPC405 debug resources include special debug modes that support the various types of debugging used during hardware and software development. These are:

- *Internal-debug mode* for use by ROM monitors and software debuggers
- *External-debug mode* for use by JTAG debuggers
- *Debug-wait mode*, which allows the servicing of interrupts while the processor appears to be stopped
- *Real-time trace mode*, which supports event triggering for real-time tracing

Debug events are supported that allow developers to manage the debug process. Debug modes and debug events are controlled using debug registers in the processor. The debug registers are accessed either through software running on the processor or through the JTAG port. The JTAG port can also be used for board tests.

The debug modes, events, controls, and interfaces provide a powerful combination of debug resources for hardware and software development tools.

## PPC405 Interfaces

The PPC405 provides the following set of interfaces that support the attachment of cores and user logic:

- Processor local bus interface
- Device control register interface
- Clock and power management interface
- JTAG port interface
- On-chip interrupt controller interface
- On-chip memory controller interface

### Processor Local Bus

The *processor local bus* (PLB) interface provides a 32-bit address and three 64-bit data buses attached to the instruction-cache and data-cache units. Two of the 64-bit buses are attached to the data-cache unit, one supporting read operations and the other supporting write operations. The third 64-bit bus is attached to the instruction-cache unit to support instruction fetching.

### Device Control Register

The *device control register* (DCR) bus interface supports the attachment of on-chip registers for device control. Software can access these registers using the **mfdcr** and **mtdcr** instructions.

### Clock and Power Management

The *clock and power-management interface* supports several methods of clock distribution and power management.



## JTAG Port

The *JTAG port interface* supports the attachment of external debug tools. Using the JTAG test-access port, a debug tool can single-step the processor and examine internal-processor state to facilitate software debugging. This capability complies with the IEEE 1149.1 specification for vendor-specific extensions, and is therefore compatible with standard JTAG hardware for boundary-scan system testing.

## On-Chip Interrupt Controller

The *on-chip interrupt controller interface* is an external interrupt controller that combines asynchronous interrupt inputs from on-chip and off-chip sources and presents them to the core using a pair of interrupt signals (critical and noncritical). Asynchronous interrupt sources can include external signals, the JTAG and debug units, and any other on-chip peripherals.

## On-Chip Memory Controller

An *on-chip memory (OCM) interface* supports the attachment of additional memory to the instruction and data caches that can be accessed at performance levels matching the cache arrays.

# PPC405 Performance

The PPC405 executes instructions at sustained speeds approaching one cycle per instruction. **Table 1-3** lists the typical execution speed (in processor cycles) of the instruction classes supported by the PPC405.

Instructions that access memory (loads and stores) consider only the “first order” effects of cache misses. The performance penalty associated with a cache miss involves a number of second-order effects. This includes PLB contention between the instruction and data caches and the time associated with performing cache-line fills and flushes. Unless stated otherwise, the number of cycles described applies to systems having zero-wait-state memory access.

**Table 1-3: PPC405 Cycles per Instruction**

| Instruction Class                                       | Execution Cycles    |
|---------------------------------------------------------|---------------------|
| Arithmetic                                              | 1                   |
| Trap                                                    | 2                   |
| Logical                                                 | 1                   |
| Shift and Rotate                                        | 1                   |
| Multiply (32-bit, 48-bit, 64-bit results, respectively) | 1, 2, 4             |
| Multiply Accumulate                                     | 1                   |
| Divide                                                  | 35                  |
| Load                                                    | 1                   |
| Load Multiple and Load String (cache hit)               | 1 per data transfer |
| Store                                                   | 1                   |
| Store Multiple and Store String (cache hit or miss)     | 1 per data transfer |
| Move to/from device-control register                    | 3                   |
| Move to/from special-purpose register                   | 1                   |

Table 1-3: PPC405 Cycles per Instruction (*Continued*)

| Instruction Class          | Execution Cycles |
|----------------------------|------------------|
| Branch known taken         | 1 or 2           |
| Branch known not taken     | 1                |
| Predicted taken branch     | 1 or 2           |
| Predicted not-taken branch | 1                |
| Mispredicted branch        | 2 or 3           |



# Input/Output Interfaces

---

The processor block (PPC405, OCM controllers, and gasket logic) provides input/output (I/O) signals that are grouped functionally into the following interfaces:

- **Clock and Power Management Interface, page 933**
- **CPU Control Interface, page 937**
- **Reset Interface, page 939**
- **Instruction-Side Processor Local Bus Interface, page 943**
- **Data-Side Processor Local Bus Interface, page 965**
- **Device-Control Register Interface, page 994**
- **External Interrupt Controller Interface, page 1004**
- **JTAG Interface, page 1006**
- **Debug Interface, page 1011**
- **Trace Interface, page 1014**

Each section within this chapter provides the following information:

- An overview summarizing the purpose of the interface.
- An I/O symbol providing a quick view of the signal names and the direction of information flow with respect to the processor block.
- A signal table that summarizes the function of each signal. The I/O column in these tables specifies the direction of information flow with respect to the processor block.
- Detailed descriptions for each signal.

Detailed timing diagrams (where appropriate) that more clearly describe the operation of the interface. The diagrams typically illustrate best-case performance when the core is attached to the FPGA processor local bus (PLB) core, or to custom bus interface unit (BIU) designs.

The instruction-side and data-side OCM controller interfaces are described separately in **Chapter 3, PowerPC® 405 OCM Controller**.

**Appendix B, Signal Summary**, alphabetically lists the signals described in this chapter. The I/O designation and a description summary are included for each signal.

## Signal Naming Conventions

The following convention is used for signal names throughout this document:

PREFIX1PREFIX2SIGNALNAME1[SIGNALNAME1][NEG][(m:n)]

The components of a signal name are as follows:

- PREFIX1 is an uppercase prefix identifying the source of the signal. This prefix specifies either a unit (for example, CPU) or a type of interface (for example, DCR). If PREFIX1 specifies the processor block, the signal is considered an output signal. Otherwise, it is an input signal.
- PREFIX2 is an uppercase prefix identifying the destination of the signal. This prefix specifies either a unit (for example, CPU) or a type of interface (for example, DCR). If PREFIX2 specifies the processor block, the signal is considered an input signal. Otherwise, it is an output signal.
- SIGNALNAME1 is an uppercase name identifying the primary function of the signal.
- SIGNALNAME1 is an uppercase name identifying the primary function of the signal.
- [NEG] is an optional notation that indicates a signal is active low. If this notation is not use, the signal is active high.
- [m:n] is an optional notation that indicates a bussed signal. “m” designates the most-significant bit of the bus and “n” designates the least-significant bit of the bus.

**Table 2-1** defines the prefixes used in the signal names. The last column in the table identifies whether the functional unit resides *inside* the processor block or *outside* the processor block.

**Table 2-1: Signal Name Prefix Definitions**

| Prefix1 or Prefix2 | Definition                                              | Location               |
|--------------------|---------------------------------------------------------|------------------------|
| CPM                | Clock and power management                              | Outside                |
| C405               | Processor block                                         | Inside                 |
| DBG                | Debug unit                                              | Inside                 |
| DCR                | Device control register                                 | Outside                |
| DSOCM              | Data-side on-chip memory (DSOCM)                        | Outside <sup>(1)</sup> |
| EIC                | External interrupt controller                           | Outside                |
| ISOCM              | Instruction-side on-chip memory (ISOCM)                 | Outside <sup>(1)</sup> |
| JTG                | JTAG                                                    | Inside                 |
| PLB                | Processor local bus                                     | Inside                 |
| RST                | Reset                                                   | Inside                 |
| TIE                | TIE (signal tied statically to GND or V <sub>DD</sub> ) | Outside                |
| TRC                | Trace                                                   | Inside                 |
| XXX                | Unspecified FPGA unit                                   | Outside                |

**Notes:**

1. OCM controllers are located in the Processor Block.

## Clock and Power Management Interface

The clock and power management (CPM) interface enables power-sensitive applications to control the processor clock using external logic. The OCM controllers are clocked separately from the processor. Two types of processor clock control are possible:

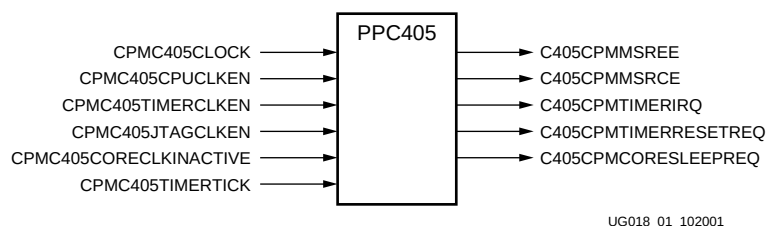
- *Global local enables* control a clock zone within the processor. These signals are used to disable the clock splitters within a zone so that the clock signal is prevented from propagating to the latches within the zone. The PPC405 is divided into three clock zones: core, timer, and JTAG. Control over a zone is exercised as follows:
  - The core clock zone contains most of the logic comprising the PPC405. It does not contain logic that belongs to the timer or JTAG zones, or other logic within the processor block. The core zone is controlled by the CPMC405CPUCLKEN signal.
  - The timer clock zone contains the PPC405 timer logic. It does not contain logic that belongs to the core or JTAG zones, or other logic within the processor block. This zone is separated from the core zone so that timer events can be used to “wake up” the core logic if a power management application has put it to sleep. The timer zone is controlled by the CPMC405TIMERCLKEN signal.
  - The JTAG clock zone contains the PPC405 JTAG logic. It does not contain logic that belongs to the core or timer zones, or other logic within the processor block. The JTAG zone is controlled by the CPMC405JTAGCLKEN signal. Although an enable is provided for this zone, the JTAG standard does not allow local gating of the JTAG clock. This enables basic JTAG functions to be maintained when the rest of the chip (including the CPM FPGA macro) is not running.
- *Global gating* controls the toggling of the PPC405 clock, CPMC405CLOCK. Instead of using the global-local enables to prevent the clock signal from propagating through a zone, CPM logic can stop the PPC405 clock input from toggling. If this method of power management is employed, the clock signal should be held active (logic 1). The CPMC405CLOCK is used by the core and timer zones, but not the JTAG zone.

CPM logic should be designed to wake the PPC405 from sleep mode when any of the following occurs:

- A timer interrupt or timer reset is asserted by the PPC405.
- A chip-reset or system-reset request is asserted (this request comes from a source other than the PPC405).
- An external interrupt or critical interrupt input is asserted and the corresponding interrupt is enabled by the appropriate machine-state register (MSR) bit.
- The DBG405DEBUGHALT chip-input signal (if provided) is asserted. Assertion of this signal indicates that an external debug tool wants to control the PPC405 processor. See [page 1012](#) for more information.

### CPM Interface I/O Signal Summary

Figure 2-1 shows the block symbol for the CPM interface. The signals are summarized in Table 2-2.



UG018\_01\_102001

Figure 2-1: CPM Interface Block Symbol

Table 2-2: CPM Interface I/O Signals

| Signal                 | I/O Type | If Unused  | Function                                                                                  |
|------------------------|----------|------------|-------------------------------------------------------------------------------------------|
| CPMC405CLOCK           | I        | Required   | PPC405 clock input (for all non-JTAG logic, including timers).                            |
| CPMC405CPUCLKEN        | I        | 1          | Enables the core clock zone.                                                              |
| CPMC405TIMERCLKEN      | I        | 1          | Enables the timer clock zone.                                                             |
| CPMC405JTAGCLKEN       | I        | 1          | Enables the JTAG clock zone.                                                              |
| CPMC405CORECLKINACTIVE | I        | 0          | Indicates the CPM logic disabled the clocks to the core.                                  |
| CPMC405TIMERTICK       | I        | 1          | Increments or decrements the PPC405 timers every time it is active with the CPMC405CLOCK. |
| C405CPMMSREE           | O        | No Connect | Indicates the value of MSR[EE].                                                           |
| C405CPMMSRCE           | O        | No Connect | Indicates the value of MSR[CE].                                                           |
| C405CPMTIMERIRQ        | O        | No Connect | Indicates a timer-interrupt request occurred.                                             |
| C405CPMTIMERRESETREQ   | O        | No Connect | Indicates a watchdog-timer reset request occurred.                                        |
| C405CPMCORESLEEPREQ    | O        | No Connect | Indicates the core is requesting to be put into sleep mode.                               |

## CPM Interface I/O Signal Descriptions

The following sections describe the operation of the CPM interface I/O signals.

### CPMC405CLOCK (input)

This signal is the source clock for all PPC405 logic (including timers). It is not the source clock for the JTAG logic. External logic can implement a power management mode that stops toggling of this signal. If such a method is employed, the clock signal should be held active (logic 1).

### CPMC405CPUCLKEN (input)

Enables the core clock zone when asserted and disables the zone when deasserted. If logic is not implemented to control this signal, it must be held active (tied to 1).

### CPMC405TIMERCLKEN (input)

Enables the timer clock zone when asserted and disables the zone when deasserted. If logic is not implemented to control this signal, it must be held active (tied to 1).

### CPMC405JTAGCLKEN (input)

Enables the JTAG clock zone when asserted and disables the zone when deasserted. CPM logic should not control this signal. The JTAG standard requires that it be held active (tied to 1).

### CPMC405CORECLKINACTIVE (input)

This signal is a status indicator that is latched by an internal PPC405 register (JDSR). An external debug tool (such as RISCWatch) can read this register and determine that the PPC405 is in sleep mode. This signal should be asserted by the CPM when it places the PPC405 in sleep mode using either of the following methods:

- Deasserting CPMC405CPUCLKEN to disable the core clock zone.
- Stopping CPMC405CLOCK from toggling by holding it active (logic 1).

### CPMC405TIMERTICK (input)

This signal is used to control the update frequency of the PPC405 time base and PIT (the FIT and WDT are timer events triggered by the time base). The time base is incremented and the PIT is decremented every cycle that CPMC405TIMERTICK and CPMC405CLOCK are both active. CPMC405TIMERTICK should be synchronous with CPMC405CLOCK for the timers to operate predictably. The timers are updated at the PPC405 clock frequency if CPMC405TIMERTICK is held active.

### C405CPMMSREE (output)

This signal indicates the state of the MSR[EE] (external-interrupt enable) bit. When asserted, external interrupts are enabled (MSR[EE]=1). When deasserted, external interrupts are disabled (MSR[EE]=0). The CPM can use this signal to wake the processor from sleep mode when an external noncritical interrupt occurs.

When the processor wakes up, it deasserts the C405CPMMSREE, C405CPMMSRCE, and C405CPMTIMERIRQ signals one processor clock cycle before it deasserts the C405CPMCORESLEEPREQ signal. For this reason, the CPM should latch the C405CPMMSREE, C405CPMMSRCE, and C405CPMTIMERIRQ signals before using them to control the processor clocks.

### C405CPMMSRCE (output)

This signal indicates the state of the MSR[CE] (critical-interrupt enable) bit. When asserted, critical interrupts are enabled (MSR[CE]=1). When deasserted, critical interrupts are disabled (MSR[CE]=0). The CPM can use this signal to wake the processor from sleep mode when an external critical interrupt occurs.

When the processor wakes up, it deasserts the C405CPMMSREE, C405CPMMSRCE, and C405CPMTIMERIRQ signals one processor clock cycle before it deasserts the C405CPMCORESLEEPREQ signal. For this reason, the CPM should latch the C405CPMMSREE, C405CPMMSRCE, and C405CPMTIMERIRQ signals before using them to control the processor clocks.

### C405CPMTIMERIRQ (output)

When asserted, this signal indicates a timer exception occurred within the PPC405 and an interrupt request is pending to handle the exception. When deasserted, no timer-interrupt request is pending. This signal is the logical OR of interrupt requests from the programmable-interval timer (PIT), the fixed-interval timer (FIT), and the watchdog timer (WDT). The CPM can use this signal to wake the processor from sleep mode when an internal timer exception occurs.

When the processor wakes up, it deasserts the C405CPMMSREE, C405CPMMSRCE, and C405CPMTIMERIRQ signals one processor clock cycle before it deasserts the C405CPMCORESLEEPREQ signal. For this reason, the CPM should latch the C405CPMMSREE, C405CPMMSRCE, and C405CPMTIMERIRQ signals before using them to control the processor clocks.

### C405CPMTIMERRESETREQ (output)

When asserted, this signal indicates a watchdog time-out occurred and a reset request is pending. When deasserted, no reset request is pending. This signal is the logical OR of the core, chip, and system reset modes that are programmed using the watchdog timer mechanism. The CPM can use this signal to wake the processor from sleep mode when a watchdog time-out occurs.

### C405CPMCORESLEEPREQ (output)

When asserted, this signal indicates the PPC405 has requested to be put into sleep mode. When deasserted, no request exists. This signal is asserted after software enables the wait state by setting the MSR[WE] (wait-state enable) bit to 1. The processor completes



execution of all prior instructions and memory accesses before asserting this signal. The CPM can use this signal to place the processor in sleep mode at the request of software.

When the processor wakes up at a later time, it deasserts the C405CPMMSREE, C405CPMMSRCE, and C405CPMTIMERIRQ signals one processor clock cycle before it deasserts the C405CPMCORESLEEPREQ signal. For this reason, the CPM should latch the C405CPMMSREE, C405CPMMSRCE, and C405CPMTIMERIRQ signals before using them to control the processor clocks.

## CPU Control Interface

The CPU control interface is used primarily to provide CPU setup information to the PPC405. It is also used to report the detection of a machine check condition within the PPC405.

### CPU Control Interface I/O Signal Summary

Figure 2-2 shows the block symbol for the CPU control interface. The signals are summarized in Table 2-3.

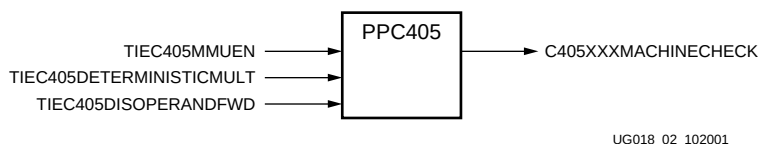


Figure 2-2: CPU Control Interface Block Symbol

Table 2-3: CPU Control Interface I/O Signals

| Signal                   | I/O Type | If Unused  | Function                                                                                                        |
|--------------------------|----------|------------|-----------------------------------------------------------------------------------------------------------------|
| TIEC405MMUEN             | I        | Required   | Enables the memory-management unit (MMU)                                                                        |
| TIEC405DETERMINISTICMULT | I        | Required   | Specifies whether all multiply operations complete in a fixed number of cycles or have an early-out capability. |
| TIEC405DISOPERANDFWD     | I        | Required   | Disables operand forwarding for load instructions.                                                              |
| C405XXXMACHINECHECK      | O        | No Connect | Indicates a machine-check error has been detected by the PPC405.                                                |

### CPU Control Interface I/O Signal Descriptions

The following sections describe the operation of the CPU control-interface I/O signals.

#### TIEC405MMUEN (input)

When held active (tied to logic 1), this signal enables the PPC405 memory-management unit (MMU). When held inactive (tied to logic 0), this signal disables the MMU. The MMU is used for virtual to address translation and for memory protection. Its operation is described in the *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*). Disabling the MMU can improve the performance (clock frequency) of the PPC405.

#### TIEC405DETERMINISTICMULT (input)

When held active (tied to logic 1), this signal disables the hardware multiplier *early-out* capability. All multiply instructions have a 4-cycle reissue rate and a 5-cycle latency rate. When held inactive (tied to logic 0), this signal enables the hardware multiplier early-out capability. If early out is enabled, multiply instructions are executed in the number of cycles specified in Table 2-4. The performance of multiply instructions is described in the *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*).

Table 2-4: Multiply and MAC Instruction Timing

| Operations                                                                                                                                                                                                                                                                                             | Issue-Rate Cycles | Latency Cycles |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------|----------------|
| MAC and Negative MAC                                                                                                                                                                                                                                                                                   | 1                 | 2              |
| Halfword $\times$ Halfword (32-bit result)                                                                                                                                                                                                                                                             | 1                 | 2              |
| Halfword $\times$ Word (48-bit result)                                                                                                                                                                                                                                                                 | 2                 | 3              |
| Word $\times$ Word (64-bit result)                                                                                                                                                                                                                                                                     | 4                 | 5              |
| <b>Notes:</b><br>For the purposes of this table, words are treated as halfwords if the upper 16 bits of the operand contain a sign extension of the lower 16 bits. For example, if the upper 16 bits of a word operand are zero, the operand is considered a halfword when calculating execution time. |                   |                |

### TIEC405DISOPERANDFWD (input)

When held active (tied to logic 1), this signal disables operand forwarding. When held inactive (tied to logic 0), this signal enables operand forwarding. The processor uses operand forwarding to send load-instruction data from the data cache to the execution units as soon as it is available. Operand forwarding often saves a clock cycle when instructions following the load require the loaded data. Disabling operand forwarding can improve the performance (clock frequency) of the PPC405.

### C405XXXMACHINECHECK (output)

When asserted, this signal indicates the PPC405 detected an instruction machine-check error. When deasserted, no error exists. This signal is asserted when the processor attempts to execute an instruction that was transferred to the PPC405 with the PLBC405ICUERR signal asserted. This signal remains asserted until software clears the instruction machine-check bit in the exception-syndrome register (ESR[MCI]).

## Reset Interface

A reset causes the processor block to perform a hardware initialization. It always occurs when the processor block is powered-up and can occur at any time during normal operation. If it occurs during normal operation, instruction execution is immediately halted and all processor state is lost.

The processor block recognizes three types of reset:

- A *processor reset* affects the processor block only, including PPC405 execution units, cache units, and the on-chip memory controller (OCM). External devices (on-chip and off-chip) are not affected. This type of reset is also referred to as a *core reset*.
- A *chip reset* affects the processor block and all other devices or peripherals located on the same chip as the processor.
- A *system reset* affects the processor chip and all other devices or peripherals external to the processor chip that are connected to the same system-reset network. The scope of a system reset depends on the system implementation. Power-on reset (POR) is a form of system reset.

Input signals are provided to the processor block for each reset type. The signals are used to reset the processor block and to record the reset type in the debug-status register (DBSR[MRR]). The processor block can produce reset-request output signals for each reset type. External reset logic can process these output signals and generate the appropriate reset input signals to the processor block. Reset activity does not occur when the processor block requests the reset. Reset activity only occurs when external logic asserts the appropriate reset input signal.

## Reset Requirements

FPGA logic (external to the processor block) is required to generate the reset input signals to the processor block. The reset input signals can be based on the reset-request output signals from the processor block, system-specific reset-request logic, or some combination of the two. Reset input signals must meet the following minimum requirements:

- The reset input signals must be synchronized with the PPC405 clock.
- The reset input signals must be asserted for at least eight clock cycles.
- Only the combinations of signals shown in [Table 2-5](#) are used to cause a reset.

POR (power-on reset) is handled by logic within the processor block. This logic asserts the RSTC405RESETCORE, RSTC405RESETCCHIP, RSTC405RESETSYS, and JTGC405TRSTNEG signals for at least eight clock cycles. FPGA designers cannot modify the processor block power-on reset mechanism.

The reset logic is not required to support all three types of reset. However, distinguishing resets by type can make it easier to isolate errors during system debug. For example, a system could reset the core to recover from an external error that affects software operation. Following the chip reset, a debugger could be used to locate the external error source which is preserved because neither a chip or system reset occurred.

[Table 2-5](#) shows the valid combinations of reset signals and their effect on the DBSR[MRR] field following reset.

Table 2-5: Valid Reset Signal Combinations and Effect on DBSR(MRR)

| Reset Input Signal                                                             | Reset Type            |          |          |          |                       |
|--------------------------------------------------------------------------------|-----------------------|----------|----------|----------|-----------------------|
|                                                                                | None                  | Core     | Chip     | System   | Power-On <sup>1</sup> |
| RSTC405RESETCORE                                                               | Deassert              | Assert   | Assert   | Assert   | Assert                |
| RSTC405RESETCHIP                                                               | Deassert              | Deassert | Assert   | Assert   | Assert                |
| RSTC405RESETSYS                                                                | Deassert              | Deassert | Deassert | Assert   | Assert                |
| JTGC405TRSTNEG                                                                 | Deassert              | Deassert | Deassert | Deassert | Assert                |
| Value of DBSR[MRR]<br>following reset                                          | Previous<br>DBSR[MRR] | 0b01     | 0b10     | 0b11     | 0b11                  |
| <b>Notes:</b><br>1. Handled automatically by logic within the processor block. |                       |          |          |          |                       |

## Reset Interface I/O Signal Summary

Figure 2-3 shows the block symbol for the reset interface. The signals are summarized in Table 2-6.

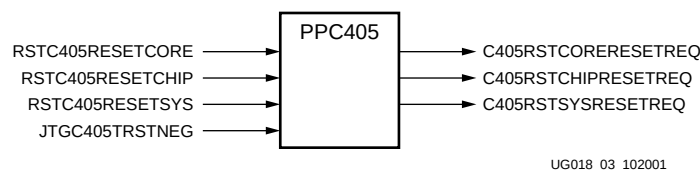


Figure 2-3: Reset Interface Block Symbol

Table 2-6: Reset Interface I/O Signals

| Signal              | I/O Type | If Unused | Function                                                                                                                             |
|---------------------|----------|-----------|--------------------------------------------------------------------------------------------------------------------------------------|
| C405RSTCORERESETREQ | O        | Required  | Indicates a core-reset request occurred.                                                                                             |
| C405RSTCHIPRESETREQ | O        | Required  | Indicates a chip-reset request occurred.                                                                                             |
| C405RSTSYSRESETREQ  | O        | Required  | Indicates a system-reset request occurred.                                                                                           |
| RSTC405RESETCORE    | I        | Required  | Resets the processor block, including the PPC405 core logic, data cache, instruction cache, and the on-chip memory controller (OCM). |
| RSTC405RESETCHIP    | I        | Required  | Indicates a chip-reset occurred.                                                                                                     |
| RSTC405RESETSYS     | I        | Required  | Indicates a system-reset occurred. Resets the logic in the PPC405 JTAG unit.                                                         |
| JTGC405TRSTNEG      | I        | Required  | Performs a JTAG test reset (TRST).                                                                                                   |

## Reset Interface I/O Signal Descriptions

The following sections describe the operation of the reset interface I/O signals.

### C405RSTCORERESETREQ (output)

When asserted, this signal indicates the processor block is requesting a core reset. If this signal is asserted, it remains active until two clock cycles after external logic asserts the

RSTC405RESETCORE input to the processor block. When deasserted, no core-reset request exists. The processor asserts this signal when one of the following occurs:

- A JTAG debugger sets the reset field in the debug-control register 0 (DBCR0[RST]) to 0b01.
- Software sets the reset field in the debug-control register 0 (DBCR0[RST]) to 0b01.
- The timer-control register watchdog-reset control field (TCR[WRC]) is set to 0b01 and a watchdog time-out causes the watchdog-event state machine to enter the reset state.

### C405RSTCHIPRESETREQ (output)

When asserted, this signal indicates the processor block is requesting a chip reset. If this signal is asserted, it remains active until two clock cycles after external logic asserts the RSTC405RESETCHIP input to the processor block. When deasserted, no chip-reset request exists. The processor asserts this signal when one of the following occurs:

- A JTAG debugger sets the reset field in the debug-control register 0 (DBCR0[RST]) to 0b10.
- Software sets the reset field in the debug-control register 0 (DBCR0[RST]) to 0b10.
- The timer-control register watchdog-reset control field (TCR[WRC]) is set to 0b10 and a watchdog time-out causes the watchdog-event state machine to enter the reset state.

### C405RSTSYSRESETREQ (output)

When asserted, this signal indicates the processor block is requesting a system reset. If this signal is asserted, it remains active until two clock cycles after external logic asserts the RSTC405RESETSYS input to the processor block. When deasserted, no system-reset request exists. The processor asserts this signal when one of the following occurs:

- A JTAG debugger sets the reset field in the debug-control register 0 (DBCR0[RST]) to 0b11.
- Software sets the reset field in the debug-control register 0 (DBCR0[RST]) to 0b11.
- The timer-control register watchdog-reset control field (TCR[WRC]) is set to 0b11 and a watchdog time-out causes the watchdog-event state machine to enter the reset state.

### RSTC405RESETCORE (input)

External logic asserts this signal to reset the processor block (core). This includes the PPC405 core logic, data cache, instruction cache, and the on-chip memory controller (OCM). The PPC405 also uses this signal to record a core reset type in the DBSR[MRR] field. This signal should be asserted for at least eight clock cycles to guarantee that the processor block initiates its reset sequence. No reset occurs and none is recorded in DBSR[MRR] when this signal is deasserted.

Table 2-5, page 940 shows the valid combinations of the RSTC405RESETCORE, RSTC405RESETCHIP, and RSTC405RESETSYS signals and their effect on the DBSR[MRR] field following reset.

### RSTC405RESETCHIP (input)

External logic asserts this signal to reset the chip. A chip reset involves the FPGA logic, on-chip peripherals, and the processor block (the PPC405 core logic, data cache, instruction cache, and the OCM). The signal does not reset logic in the processor block. The PPC405 uses this signal only to record a chip reset type in the DBSR[MRR] field. The RSTC405RESETCORE signal must be asserted with this signal to cause a core reset. Both signals must be asserted for at least eight clock cycles to guarantee that the processor block recognizes the reset type and initiates the core-reset sequence. The PPC405 does not record a chip reset type in DBSR[MRR] when this signal is deasserted.

Table 2-5, page 940 shows the valid combinations of the RSTC405RESETCORE, RSTC405RESETCCHIP, and RSTC405RESETSYS signals and their effect on the DBSR[MRR] field following reset.

### RSTC405RESETSYS (input)

External logic asserts this signal to reset the system. A system reset involves logic external to the FPGA, the FPGA logic, on-chip peripherals, and the processor block (the PPC405 core logic, data cache, instruction cache, and the OCM). This signal resets the logic in the PPC405 JTAG unit, but it does not reset any other processor block logic. The PPC405 uses this signal to record a system reset type in the DBSR[MRR] field. The RSTC405RESETCORE signal must be asserted with this signal to cause a core reset. The RSTC405RESETCORE, RSTC405RESETCCHIP, and RSTC405RESETSYS signals must be asserted for at least eight clock cycles to guarantee that the processor block recognizes the reset type and initiates the core-reset sequence. The PPC405 does not record a system reset type in DBSR[MRR] when this signal is deasserted.

This signal must be asserted during a power-on reset to properly initialize the JTAG unit.

Table 2-5, page 940 shows the valid combinations of the RSTC405RESETCORE, RSTC405RESETCCHIP, and RSTC405RESETSYS signals and their effect on the DBSR[MRR] field following reset.

### JTGC405TRSTNEG (input)

This input is the JTAG test reset ( $\overline{\text{TRST}}$ ) signal. It can be connected to the chip-level  $\overline{\text{TRST}}$  signal. Although optional in IEEE Standard 1149.1, this signal is automatically used by the processor block during power-on reset to properly reset all processor block logic, including the JTAG and debug logic. When deasserted, no JTAG test reset exists.

This is a negative active signal.

## Instruction-Side Processor Local Bus Interface

The instruction-side processor local bus (ISPLB) interface enables the PPC405 instruction cache unit (ICU) to fetch (read) instructions from any memory device connected to the processor local bus (PLB). The ICU cannot write to memory. This interface has a dedicated 30-bit address bus output and a dedicated 64-bit read-data bus input. The interface is designed to attach as a master to a 64-bit PLB, but it also supports attachment as a master to a 32-bit PLB. The interface is capable of one transfer (64 or 32 bits) every PLB cycle.

At the chip level, the ISPLB can be combined with the data-side read-data bus (also a PLB master) to create a shared read-data bus. This is done if a single PLB arbiter services both PLB masters and the PLB arbiter implementation only returns data to one PLB master at a time.

Refer to the *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*) for more information on the operation of the PPC405 ICU.

### Instruction-Side PLB Operation

Fetch requests are produced by the ICU and communicated over the PLB interface. Fetch requests occur when an access misses the instruction cache or the memory location that is accessed is non-cacheable. A fetch request contains the following information:

- A fetch request is indicated by C405PLBICUREQUEST (page 947).
- The target address of the instruction to be fetched is specified by the address bus, C405PLBICUABUS[0:29] (page 947). Bits 30:31 of the 32-bit instruction-fetch address are always zero and must be tied to zero at the PLB arbiter. The ICU always requests an aligned doubleword of data, so the byte enables are not used.
- The transfer size is specified as four words (quadword) or eight words (cache line) using C405PLBICUSIZE[2:3] (page 947). The remaining bits of the transfer size (0:1) must be tied to zero at the PLB arbiter.
- The cacheability storage attribute is indicated by C405PLBICUCACHEABLE (page 948). Cacheable transfers are always performed with an eight-word transfer size.
- The user-defined storage attribute is indicated by C405PLBICUU0ATTR (page 948).
- The request priority is indicated by C405PLBICUPRIORITY[0:1] (page 949). The PLB arbiter uses this information to prioritize simultaneous requests from multiple PLB masters.

The processor can abort a PLB fetch request using C405PLBICUABORT (page 949). This can occur when a branch instruction is executed or when an interrupt occurs.

Fetches instructions are returned to the ICU by a PLB slave device over the PLB interface. A fetch response contains the following information:

- The fetch-request address is acknowledged by the PLB slave using PLBC405ICUADDRACK (page 949).
- Instructions sent from the PLB slave to the ICU during a line transfer are indicated as valid using PLBC405ICURDDACK (page 950).
- The PLB-slave bus width, or size (32-bit or 64-bit), is specified by PLBC405ICUFSIZE1 (page 950). The PLB slave is responsible for packing data bytes from non-word devices so that the information sent to the ICU is presented appropriately, as determined by the transfer size.
- The instructions returned to the ICU by the PLB slave are sent using four-word or eight-word line transfers, as specified by the transfer size in the fetch request. These instructions are returned over the ICU read-data bus, PLBC405ICURDDBUS[0:63] (page 951). Line transfers operate as follows:
  - A four-word line transfer returns the quadword aligned on the address specified by C405PLBICUABUS[0:27]. This quadword contains the target instruction



- requested by the ICU. The quadword is returned using two doubleword or four word transfer operations, depending on the PLB slave bus width (64-bit or 32-bit, respectively).
- An eight-word line transfer returns the eight-word cache line aligned on the address specified by C405PLBICUABUS[0:26]. This cache line contains the target instruction requested by the ICU. The cache line is returned using four doubleword or eight word transfer operations, depending on the PLB slave bus width (64-bit or 32-bit, respectively).
- The words returned during a line transfer can be sent from the PLB slave to the ICU in any order (target-word-first, sequential, other). This transfer order is specified by PLBC405ICURDWDADDR[1:3] (page 951).

## Interaction with the ICU Fill Buffer

As mentioned above, the PLB slave can transfer instructions to the ICU in any order (target-word-first, sequential, other). When instructions are received by the ICU from the PLB slave, they are placed in the ICU fill buffer. When the ICU receives the target instruction, it forwards it immediately from the fill buffer to the instruction-fetch unit so that pipeline stalls due to instruction-fetch delays are minimized. This operation is referred to as a *bypass*. The remaining instructions are received from the PLB slave and placed in the fill buffer. Subsequent instruction fetches read from the fill buffer if the instruction is already present in the buffer. For the best possible software performance, the PLB slave should be designed to return the target word first.

Non-cacheable instructions are transferred using a four-word or eight-word line-transfer size. Software controls this transfer size using the *non-cacheable request-size* bit in the core-configuration register (CCR0[NCRS]). This enables non-cacheable transfers to take advantage of the PLB line-transfer protocol to minimize PLB-arbitration delays and bus delays associated with multiple, single-word transfers. The transferred instructions are placed in the ICU fill buffer, but not in the instruction cache. Subsequent instruction fetches from the same non-cacheable line are read from the fill buffer instead of requiring a separate arbitration and transfer sequence across the PLB. Instructions in the fill buffer are fetched with the same performance as a cache hit. The non-cacheable line remains in the fill buffer until the fill buffer is needed by another line transfer.

Cacheable instructions are always transferred using an eight-word line-transfer size. The transferred instructions are placed in the ICU fill buffer as they are received from the PLB slave. Subsequent instruction fetches from the same cacheable line are read from the fill buffer during the time the line is transferred from the PLB slave. When the fill buffer is full, its contents are transferred to the instruction cache. Software can prevent this transfer by setting the *fetch without allocate* bit in the core-configuration register (CCR0[FWOA]). In this case, the cacheable line remains in the fill buffer until the fill buffer is needed by another line transfer. An exception is that the contents of the fill buffer are always transferred if the line was fetched because an *icbt* instruction was executed.

## Prefetch and Address Pipelining

A *prefetch* is a request for the eight-word cache line that sequentially follows the current eight-word fetch request. Prefetched instructions are fetched before it is known that they are needed by the sequential execution of software.

The ICU can overlap a single prefetch request with the prior fetch request. This process, known as *address pipelining*, enables a second address to be presented to a PLB slave while the slave is returning data associated with the first address. Address pipelining can occur if a prefetch request is produced before all instructions from the previous fetch request are transferred by the slave. This capability maximizes PLB-transfer throughput by reducing dead cycles between instruction transfers associated with the two requests. The ICU can pipeline the prefetch with any combination of sequential, branch, and interrupt fetch requests. A prefetch request is communicated over the PLB two or more cycles after the prior fetch request is acknowledged by the PLB slave.

Address pipelining of prefetch requests never occurs under any one of the following conditions:

- The PLB slave does not support address pipelining.
- The prefetch address falls outside the 1 KB physical page holding the current fetch address. This limitation avoids potential problems due to protection violations or storage-attribute mismatches.
- Non-cacheable transfers are programmed to use a four-word line-transfer size (CCR0[NCRS]=0).
- For non-cacheable transfers, prefetching is disabled (CCR0[PFNC]=0).
- For cacheable transfers, prefetching is disabled (CCR0[PFC]=0).

Address pipelining of non-cacheable prefetch requests can occur if all of the following conditions are met:

- Address pipelining is supported by the PLB slave.
- The ICU is not already involved in an address-pipelined PLB transfer.
- A branch or interrupt does not modify the sequential execution of the current (first) instruction-fetch request.
- Non-cacheable prefetching is enabled (CCR0[PFNC]=1).
- A non-cacheable instruction-prefetch is requested, and the instruction is not in the fill buffer or being returned over the ISOCM interface.
- The prefetch address does not fall outside the current 1 KB physical page.

Address pipelining of cacheable prefetch requests can occur if all of the following conditions are met:

- Address pipelining is supported by the PLB slave.
- The ICU is not already involved in an address-pipelined PLB transfer.
- A branch or interrupt does not modify the sequential execution of the current (first) instruction-fetch request.
- Cacheable prefetching is enabled (CCR0[PFC]=1).
- A cacheable instruction-prefetch is requested, and the instruction is not in the instruction cache, the fill buffer, or being returned over the ISOCM interface.
- The prefetch address does not fall outside the current 1 KB physical page.

## Guarded Storage

Accesses to guarded storage are not indicated by the ISPLB interface. This is because the PowerPC Architecture allows instruction prefetching when:

- The processor is in real mode (instruction address translation is disabled).
- The fetched instruction is located in the same physical page (1 KB) as an instruction that is required by the sequential execution model, or
- The fetched instruction is located in the next physical page (1 KB) as an instruction that is required by the sequential execution model.

Memory should be organized such that real-mode instruction prefetching from the same or next 1 KB page does not affect sensitive addresses, such as memory-mapped I/O devices.

If the processor is in virtual mode, an attempt to prefetch from guarded storage causes an instruction-storage interrupt. In this case, the prefetch never appears on the ISPLB.

## Instruction-Side PLB I/O Signal Table

Figure 2-4 shows the block symbol for the instruction-side PLB interface. The signals are summarized in Table 2-7.

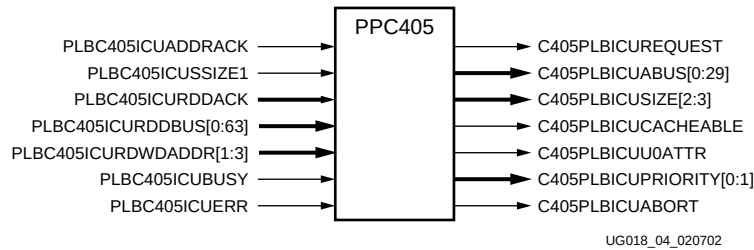


Figure 2-4: Instruction-Side PLB Interface Block Symbol

Table 2-7: Instruction-Side PLB Interface Signal Summary

| Signal                  | I/O Type | If Unused                 | Function                                                                                                                |
|-------------------------|----------|---------------------------|-------------------------------------------------------------------------------------------------------------------------|
| C405PLBICUREQUEST       | O        | No Connect                | Indicates the ICU is making an instruction-fetch request.                                                               |
| C405PLBICUABUS[0:29]    | O        | No Connect                | Specifies the memory address of the instruction-fetch request. Bits 30:31 of the 32-bit address are assumed to be zero. |
| C405PLBICUFSIZE[2:3]    | O        | No Connect                | Specifies a four word or eight word line-transfer size.                                                                 |
| C405PLBICUCACHEABLE     | O        | No Connect                | Indicates the value of the cacheability storage attribute for the target address.                                       |
| C405PLBICUU0ATTR        | O        | No Connect                | Indicates the value of the user-defined storage attribute for the target address.                                       |
| C405PLBICUPRIORITY[0:1] | O        | No Connect                | Indicates the priority of the ICU fetch request.                                                                        |
| C405PLBICUABORT         | O        | No Connect                | Indicates the ICU is aborting an unacknowledged fetch request.                                                          |
| PLBC405ICUADDRACK       | I        | 0                         | Indicates a PLB slave acknowledges the current ICU fetch request.                                                       |
| PLBC405ICUFSIZE1        | I        | 0                         | Specifies the bus width (size) of the PLB slave that accepted the request.                                              |
| PLBC405ICURDDACK        | I        | 0                         | Indicates the ICU read-data bus contains valid instructions for transfer to the ICU.                                    |
| PLBC405ICURDDBUS[0:63]  | I        | 0x0000_0000<br>_0000_0000 | The ICU read-data bus used to transfer instructions from the PLB slave to the ICU.                                      |
| PLBC405ICURDWDADDR[1:3] | I        | 0b000                     | Indicates which word or doubleword of a four-word or eight-word line transfer is present on the ICU read-data bus.      |
| PLBC405ICUBUSY          | I        | 0                         | Indicates the PLB slave is busy performing an operation requested by the ICU.                                           |
| PLBC405ICUERR           | I        | 0                         | Indicates an error was detected by the PLB slave during the transfer of instructions to the ICU.                        |

## Instruction-Side PLB Interface I/O Signal Descriptions

The following sections describe the operation of the instruction-side PLB interface I/O signals.

Throughout these descriptions and unless otherwise noted, the term *clock* refers to the PLB clock signal, PLBCLK (see [page 1018](#) for information on this clock signal). The term *cycle*

refers to a PLB cycle. To simplify the signal descriptions, it is assumed that PLBCLK and the PPC405 clock (CPMC405CLOCK) operate at the same frequency.

### C405PLBICUREQUEST (output)

When asserted, this signal indicates the ICU is requesting instructions from a PLB slave device. The PLB slave asserts PLBC405ICUADDRACK to acknowledge the request. The request can be acknowledged in the same cycle it is presented by the ICU. The request is deasserted in the cycle after it is acknowledged by the PLB slave. When deasserted, no unacknowledged instruction-fetch request exists.

The following output signals contain information for the PLB slave device and are valid when the request is asserted. The PLB slave must latch these signals by the end of the same cycle it acknowledges the request:

- C405PLBICUABUS[0:31] contains the word address of the instruction-fetch request.
- C405PLBICUSIZE[2:3] indicates the instruction-fetch line-transfer size.
- C405PLBICUCACHEABLE indicates whether the instruction-fetch address is cacheable.
- C405PLBICUU0ATTR indicates the value of the user-defined storage attribute for the instruction-fetch address.

C405PLBICUPRIORITY[0:1] is also valid when the request is asserted. This signal indicates the priority of the instruction-fetch request. It is used by the PLB arbiter to prioritize simultaneous requests from multiple PLB masters.

The ICU supports two outstanding fetch requests over the PLB. The ICU can make a second fetch request (a prefetch) after the current request is acknowledged. The ICU deasserts C405PLBICUREQUEST for at least one cycle after the current request is acknowledged and before the subsequent request is asserted.

If the PLB slave supports address pipelining, it must respond to the two fetch requests in the order they are presented by the ICU. All instructions associated with the first request must be returned before any instruction associated with the second request is returned. The ICU cannot present a third fetch request until the first request is completed by the PLB slave. This third request can be presented two cycles after the last read acknowledge (PLBC405ICURDDACK) is sent from the PLB slave to the ICU, completing the first request.

The ICU can abort a fetch request if it no longer requires the requested instruction. The ICU removes a request by asserting C405PLBICUABORT while the request is asserted. In the next cycle the request is deasserted and remains deasserted for at least one cycle.

### C405PLBICUABUS[0:29] (output)

This bus specifies the memory address of the instruction-fetch request. Bits 30:31 of the 32-bit address are assumed to be zero so that all fetch requests are aligned on a word boundary. The fetch address is valid during the time the fetch request signal (C405PLBICUREQUEST) is asserted. It remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405ICUADDRACK to acknowledge the request).

C405PLBICUSIZE[2:3] indicates the instruction-fetch line-transfer size. The PLB slave uses memory-address bits [0:27] to specify an aligned four-word address for a four-word transfer size. Memory-address bits [0:26] are used to specify an aligned eight-word address for an eight-word transfer size.

### C405PLBICUSIZE[2:3] (output)

These signals are used to specify the line-transfer size of the instruction-fetch request. A four-word transfer size is specified when C405PLBICUSIZE[2:3]=0b01. An eight-word transfer size is specified when C405PLBICUSIZE[2:3]=0b10. The transfer size is valid during the cycles the fetch-request signal (C405PLBICUREQUEST) is asserted. It remains

valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405ICUADDRACK to acknowledge the request).

A four-word line transfer returns the quadword aligned on the address specified by C405PLBICUABUS[0:27]. This quadword contains the target instruction requested by the ICU. The quadword is returned using two doubleword or four word transfer operations, depending on the PLB slave bus width (64-bit or 32-bit, respectively).

An eight-word line transfer returns the eight-word cache line aligned on the address specified by C405PLBICUABUS[0:26]. This cache line contains the target instruction requested by the ICU. The cache line is returned using four doubleword or eight word transfer operations, depending on the PLB slave bus width (64-bit or 32-bit, respectively).

The words returned during a line transfer can be sent from the PLB slave to the ICU in any order (target-word-first, sequential, other). This transfer order is specified by PLBC405ICURDWDADDR[1:3].

### C405PLBICUCACHEABLE (output)

This signal indicates whether the requested instructions are cacheable. It reflects the value of the cacheability storage attribute for the target address. The requested instructions are non-cacheable when the signal is deasserted (0). They are cacheable when the signal is asserted (1). This signal is valid during the time the fetch-request signal (C405PLBICUREQUEST) is asserted. It remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405ICUADDRACK to acknowledge the request).

Non-cacheable instructions are transferred using a four-word or eight-word line-transfer size. Software controls this transfer size using the *non-cacheable request-size* bit in the core-configuration register (CCR0[NCRS]). This enables non-cacheable transfers to take advantage of the PLB line-transfer protocol to minimize PLB-arbitration delays and bus delays associated with multiple, single-word transfers. The transferred instructions are placed in the ICU fill buffer, but not in the instruction cache. Subsequent instruction fetches from the same non-cacheable line are read from the fill buffer instead of requiring a separate arbitration and transfer sequence across the PLB. Instructions in the fill buffer are fetched with the same performance as a cache hit. The non-cacheable line remains in the fill buffer until the fill buffer is needed by another line transfer.

Cacheable instructions are always transferred using an eight-word line-transfer size. The transferred instructions are placed in the ICU fill buffer as they are received from the PLB slave. Subsequent instruction fetches from the same cacheable line are read from the fill buffer during the time the line is transferred from the PLB slave. When the fill buffer is full, its contents are transferred to the instruction cache. Software can prevent this transfer by setting the *fetch without allocate* bit in the core-configuration register (CCR0[FWOA]). In this case, the cacheable line remains in the fill buffer until the fill buffer is needed by another line transfer. An exception is that the contents of the fill buffer are always transferred if the line was fetched because an **icbt** instruction was executed.

### C405PLBICUU0ATTR (output)

This signal reflects the value of the user-defined (U0) storage attribute for the target address. The requested instructions are not in memory locations characterized by this attribute when the signal is deasserted (0). They are in memory locations characterized by this attribute when the signal is asserted (1). This signal is valid during the time the fetch-request signal (C405PLBICUREQUEST) is asserted. It remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405ICUADDRACK to acknowledge the request).

The system designer can use this signal to assign special behavior to certain memory addresses. Its use is optional.



## C405PLBICUABORT (output)

When asserted, this signal indicates the ICU is aborting the current fetch request. It is used by the ICU to abort a request that has not been acknowledged, or is in the process of being acknowledged by the PLB slave. The fetch request continues normally if this signal is not asserted. This signal is only valid during the time the fetch-request signal (C405PLBICUREQUEST) is asserted. It must be ignored by the PLB slave if the fetch-request signal is not asserted. In the cycle after the abort signal is asserted, the fetch-request signal is deasserted and remains deasserted for at least one cycle.

If the abort signal is asserted in the same cycle that the fetch request is acknowledged by the PLB slave (PLBC405ICUADDRACK is asserted), the PLB slave is responsible for ensuring that the transfer does not proceed further. The PLB slave cannot assert the ICU read-data bus acknowledgement signal (PLBC405ICURDDACK) for an aborted request.

The ICU can abort an address-pipelined fetch request while the PLB slave is responding to a previous fetch request. The PLB slave is responsible for completing the previous fetch request and aborting the new (pipelined) request.

## C405PLBICUPRIORITY[0:1] (output)

These signals are used to specify the priority of the instruction-fetch request. [Table 2-8](#) shows the encoding of the 2-bit PLB-request priority signal. The priority is valid during the cycles the fetch-request signal (C405PLBICUREQUEST) is asserted. It remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405ICUADDRACK to acknowledge the request).

**Table 2-8: PLB-Request Priority Encoding**

| Bit 0 | Bit 1 | Definition                            |
|-------|-------|---------------------------------------|
| 0     | 0     | Lowest PLB-request priority.          |
| 0     | 1     | Next-to-lowest PLB-request priority.  |
| 1     | 0     | Next-to-highest PLB-request priority. |
| 1     | 1     | Highest PLB-request priority.         |

Software establishes the instruction-fetch request priority by writing the appropriate value into the *ICU PLB-priority bits 0:1* of the core-configuration register (CCR0[IPP]). After a reset, the priority is set to the highest level (CCR0[IPP]=0b11).

## PLBC405ICUADDRACK (input)

When asserted, this signal indicates the PLB slave acknowledges the ICU fetch request (indicated by the ICU assertion of C405PLBICUREQUEST). When deasserted, no such acknowledgement exists. A fetch request can be acknowledged by the PLB slave in the same cycle the request is asserted by the ICU. The PLB slave must latch the following fetch-request information in the same cycle it asserts the fetch acknowledgement:

- C405PLBICUABUS[0:29], which contains the word address of the instruction-fetch request.
- C405PLBICUSIZE[2:3], which indicates the instruction-fetch line-transfer size.
- C405PLBICUCACHEABLE, which indicates whether the instruction-fetch address is cacheable.
- C405PLBICUU0ATTR, which indicates the value of the user-defined storage attribute for the instruction-fetch address (use of this signal is optional).

During the acknowledgement cycle, the PLB slave must return its bus width indicator (32 bits or 64 bits) using the PLBC405ICUFSIZE1 signal.

The acknowledgement signal remains asserted for one cycle. In the next cycle, both the fetch request and acknowledgement are deasserted. Instructions can be returned to the ICU from the PLB slave beginning in the cycle following the acknowledgement. The PLB slave must abort an ICU fetch request (return no instructions) if the ICU asserts C405PLBICUABORT in the same cycle the PLB slave acknowledges the request.

The ICU supports two outstanding fetch requests over the PLB. The ICU can make a second fetch request after the current request is acknowledged. The ICU deasserts C405PLBICUREQUEST for at least one cycle after the current request is acknowledged and before the subsequent request is asserted.

If the PLB slave supports address pipelining, it must respond to the two fetch requests in the order they are presented by the ICU. All instructions associated with the first request must be returned before any instruction associated with the second request is returned. The ICU cannot present a third fetch request until the first request is completed by the PLB slave. This third request can be presented two cycles after the last read acknowledge (PLBC405ICURDDACK) is sent from the PLB slave to the ICU, completing the first request.

### PLBC405ICUFSIZE1 (input)

This signal indicates the bus width (size) of the PLB slave device that acknowledged the ICU fetch request. A 32-bit PLB slave responded when the signal is deasserted (0). A 64-bit PLB slave responded when the signal is asserted (1). This signal is valid during the cycle the acknowledge signal (PLBC405ICUADDRACK) is asserted.

The size signal is used by the ICU to determine how instructions are read from the 64-bit PLB interface during a transfer cycle (a transfer occurs when the PLB slave asserts PLBC405ICURDDACK). The ICU uses the size signal as follows:

- When a 32-bit PLB slave responds, an aligned word is sent from the slave to the ICU during each transfer cycle. The 32-bit PLB slave bus should be connected to both the high and low 32 bits of the 64-bit ICU read-data bus (see [Figure 2-5, page 951](#)). This type of connection duplicates the word returned by the slave across the 64-bit bus. The ICU reads either the low 32 bits or the high 32 bits of the 64-bit interface, depending on the order of the transfer (PLBC405ICURDWDADDR[1:3]).
- When a 64-bit PLB slave responds, an aligned doubleword is sent from the slave to the ICU during each transfer cycle. Both words are read from the 64-bit interface by the ICU in this cycle.

[Table 2-10, page 952](#) shows the location of instructions on the ICU read-data bus as a function of PLB-slave size, line-transfer size, and transfer order.

### PLBC405ICURDDACK (input)

When asserted, this signal indicates the ICU read-data bus contains valid instructions sent by the PLB slave to the ICU (read data is acknowledged). The ICU latches the data from the bus at the end of the cycle this signal is asserted. The contents of the ICU read-data bus are not valid when this signal is deasserted.

Read-data acknowledgement is asserted for one cycle per transfer. There is no limit to the number of cycles between two transfers. The number of transfers (and the number of read-data acknowledgements) depends on the following:

- The PLB slave size (bus width) specified by PLBC405ICUFSIZE1.
- The line-transfer size specified by C405PLBICUFSIZE[2:3].
- The cacheability of the fetched instructions specified by C405PLBICUCACHEABLE.
- The value of the *non-cacheable request-size* bit (CCR0[NCRS]).

[Table 2-9](#) summarizes the effect these parameters have on the number of transfers.

Table 2-9: Number of Transfers Required for Instruction-Fetch Requests

| PLB-Slave Size | Line-Transfer Size | Instruction Cacheability | CCR0[NCRS] | Number of Transfers |
|----------------|--------------------|--------------------------|------------|---------------------|
| 32-Bit         | Four Words         | Non-Cacheable            | 0          | 4                   |
|                | Eight Words        |                          | 1          | 8                   |
|                | Eight Words        | Cacheable                | —          | 8                   |
| 64-Bit         | Four Words         | Non-Cacheable            | 0          | 2                   |
|                | Eight Words        |                          | 1          | 4                   |
|                | Eight Words        | Cacheable                | —          | 4                   |

### PLBC405ICURDDBUS[0:63] (input)

This read-data bus contains the instructions transferred from a PLB slave to the ICU. The contents of the bus are valid when the read-data acknowledgement signal (PLBC405ICURDDACK) is asserted. This acknowledgment is asserted for one cycle per transfer. There is no limit to the number of cycles between two transfers. The bus contents are not valid when the read-data acknowledgement signal is deasserted.

The PLB slave returns either a single instruction (an aligned word) or two instructions (an aligned doubleword) per transfer. The number of instructions sent per transfer depends on the PLB slave size (bus width), as follows:

- When a 32-bit PLB slave responds, an aligned word is sent from the slave to the ICU during each transfer cycle. The 32-bit PLB slave bus should be connected to both the high and low 32 bits of the 64-bit read-data bus, as shown in [Figure 2-5](#) below. This type of connection duplicates the word returned by the slave across the 64-bit bus. The ICU reads either the low 32 bits or the high 32 bits of the 64-bit interface, depending on the value of PLBC405ICURDWDADDR[1:3].
- When a 64-bit PLB slave responds, an aligned doubleword is sent from the slave to the ICU during each transfer cycle. Both words are read from the 64-bit interface by the ICU in this cycle.

[Table 2-10](#) shows the location of instructions on the ICU read-data bus as a function of PLB-slave size, line-transfer size, and transfer order.

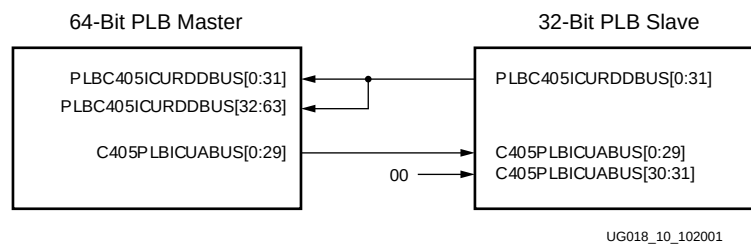


Figure 2-5: Attachment of ISPLB Between 32-Bit Slave and 64-Bit Master

### PLBC405ICURDWDADDR[1:3] (input)

These signals are used to specify the transfer order. They identify which word or doubleword of a line transfer is present on the ICU read-data bus when the PLB slave returns instructions to the ICU. The words returned during a line transfer can be sent from the PLB slave to the ICU in any order (target-word-first, sequential, other). The transfer-order signals are valid when the read-data acknowledgement signal (PLBC405ICURDDACK) is asserted. This acknowledgment is asserted for one cycle per



transfer. There is no limit to the number of cycles between two transfers. The transfer-order signals are not valid when the read-data acknowledgement signal is deasserted.

Table 2-10 shows the location of instructions on the ICU read-data bus as a function of PLB-slave size, line-transfer size, and transfer order. In this table, the *Transfer Order* column contains the possible values of PLBC405ICURDWDADDR[1:3]. For 64-bit PLB slaves, PLBC405ICURDWDADDR[3] should always be 0 during a transfer. In this case, the transfer order is invalid if this signal asserted. The entries for a 32-bit PLB slave assume the connection to a 64-bit master shown in Figure 2-5, page 951.

Table 2-10: Contents of ICU Read-Data Bus During Line Transfer

| PLB-Slave Size | Line-Transfer Size | Transfer Order <sup>1</sup> | ICU Read-Data Bus [0:31] <sup>2</sup> | ICU Read-Data Bus [32:63] <sup>2</sup> |
|----------------|--------------------|-----------------------------|---------------------------------------|----------------------------------------|
| 32-Bit         | Four Words         | x00                         | Instruction 0                         | <i>Instruction 0</i>                   |
|                |                    | x01                         | <i>Instruction 1</i>                  | Instruction 1                          |
|                |                    | x10                         | Instruction 2                         | <i>Instruction 2</i>                   |
|                |                    | x11                         | <i>Instruction 3</i>                  | Instruction 3                          |
|                | Eight Words        | 000                         | Instruction 0                         | <i>Instruction 0</i>                   |
|                |                    | 001                         | <i>Instruction 1</i>                  | Instruction 1                          |
|                |                    | 010                         | Instruction 2                         | <i>Instruction 2</i>                   |
|                |                    | 011                         | <i>Instruction 3</i>                  | Instruction 3                          |
|                |                    | 100                         | Instruction 4                         | <i>Instruction 4</i>                   |
|                |                    | 101                         | <i>Instruction 5</i>                  | Instruction 5                          |
|                |                    | 110                         | Instruction 6                         | <i>Instruction 6</i>                   |
|                |                    | 111                         | <i>Instruction 7</i>                  | Instruction 7                          |
| 64-Bit         | Four Words         | x00                         | Instruction 0                         | Instruction 1                          |
|                |                    | x10                         | Instruction 2                         | Instruction 3                          |
|                |                    | xx1                         | Invalid                               |                                        |
|                | Eight Words        | 000                         | Instruction 0                         | Instruction 1                          |
|                |                    | 010                         | Instruction 2                         | Instruction 3                          |
|                |                    | 100                         | Instruction 4                         | Instruction 5                          |
|                |                    | 110                         | Instruction 6                         | Instruction 7                          |
|                |                    | xx1                         | Invalid                               |                                        |

**Notes:**

- “x” indicates a don’t-care value in PLBC405ICURDWDADDR[1:3].
- An instruction shown in *italics* is ignored by the ICU during the transfer.

### PLBC405ICUBUSY (input)

When asserted, this signal indicates the PLB slave acknowledged and is responding to (is busy with) an ICU fetch request. When deasserted, the PLB slave is not responding to an ICU fetch request.

This signal should be asserted in the cycle after an ICU fetch request is acknowledged by the PLB slave and remain asserted until the request is completed by the PLB slave. It should be deasserted in the cycle after the last read-data acknowledgement signal is asserted by the PLB slave, completing the transfer. If multiple fetch requests are initiated

and overlap, the busy signal should be asserted in the cycle after the first request is acknowledged and remain asserted until the cycle after the final read-data acknowledgement is completed for the last request.

Following reset, the processor block prevents the ICU from fetching instructions until the busy signal is deasserted for the first time. This is useful in situations where the processor block is reset by a core reset, but PLB devices are not reset. Waiting for the busy signal to be deasserted prevents fetch requests following reset from interfering with PLB activity that was initiated before reset.

### PLBC405ICUERR (input)

When asserted, this signal indicates the PLB slave detected an error when attempting to access or transfer the instructions requested by the ICU. This signal should be asserted with the read-data acknowledgement signal that corresponds to the erroneous transfer. The error signal should be asserted for only one cycle. When deasserted, no error is detected.

If a cacheable instruction is transferred with an error indication, it is loaded into the ICU fill buffer. However, the cache line held in the fill buffer is not transferred to the instruction cache.

The PLB slave must not terminate instruction transfers when an error is detected. The processor block is responsible for responding to any error detected by the PLB slave. A machine-check exception occurs if the PPC405 attempts to *execute* an instruction that was transferred to the ICU with an error indication. If an instruction is transferred with an error indication but is never executed, no machine-check exception occurs.

The PLB slave should latch error information in DCRs so that software diagnostic routines can attempt to report and recover from the error. A bus-error address register (BEAR) should be implemented for storing the address of the access that caused the error. A bus-error syndrome register (BESR) should be implemented for storing information about cause of the error.

## Instruction-Side PLB Interface Timing Diagrams

The following timing diagrams show typical transfers that can occur on the ISPLB interface between the ICU and a bus-interface unit (BIU). These timing diagrams represent the optimal timing relationships supported by the processor block. The BIU can be implemented using the FPGA processor local bus (PLB) or using customized hardware. Not all BIU implementations support these optimal timing relationships.

The ICU only performs reads (fetches) when accessing instructions across the ISPLB interface.

### ISPLB Timing Diagram Assumptions

The following assumptions and simplifications were made in producing the optimal timing relationships shown in the timing diagrams:

- Fetch requests are acknowledged by the BIU in the same cycle they are presented by the ICU. This represents the earliest cycle a BIU can acknowledge a fetch request.
- The first read-data acknowledgement for a line transfer is asserted in the cycle immediately following the fetch-request acknowledgement. This represents the earliest cycle a BIU can begin transferring instructions to the ICU in response to a fetch request. However, the earliest the FPGA PLB begins transferring instructions is *two cycles* after the fetch request is acknowledged.
- Subsequent read-data acknowledgements for a line transfer are asserted in the cycle immediately following the prior read-data acknowledgement. This represents the fastest rate at which a BIU can transfer instructions to the ICU (there is no limit to the number of cycles between two transfers).
- All line transfers assume the target instruction (word) is returned first. Subsequent instructions in the line are returned sequentially by address, wrapping as necessary to the lower addresses in the same line.
- The rate at which the ICU makes instruction-fetch requests to the BIU is not limited by the rate instructions are executed.
- An ICU fetch request to the BIU occurs two cycles after a miss is determined by the ICU.
- The ICU latches instructions into the fill buffer in the cycle after the instructions are received from the BIU on the PLB.
- The transfer of instructions from the fill buffer to the instruction cache takes three cycles. This transfer takes place after all instructions are read into the fill buffer from the BIU.
- The BIU size (bus width) is 64 bits, so PLBC405ICUFSIZE1 is not shown.
- No instruction-access errors occur, so PLBC405ICUERR is not shown.
- The abort signal, C405PLBICUABORT is shown only in the last example.
- The storage attribute signals are not shown.
- The ICU activity is shown only as an aide in describing the examples. The occurrence and duration of this activity is not observable on the ISPLB.

The following abbreviations appear in the timing diagrams:

Table 2-11: Key to ISPLB Timing Diagram Abbreviations

| Abbreviation <sup>1</sup> | Description                                                                                                               | Where Used                                                         |                                                                  |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------|------------------------------------------------------------------|
| rl#                       | Fetch-request identifier                                                                                                  | Request<br>Request acknowledge<br>Read-data acknowledge            | (C405PLBICUREQUEST)<br>(PLBC405ICUADDRACK)<br>(PLBC405ICURDDACK) |
| adr#                      | Fetch-request address                                                                                                     | Request address                                                    | (C405PLBICUABUS[0:29])                                           |
| d# <sub>#</sub>           | Doublewords (two instructions) transferred as a result of a fetch request                                                 | ICU read-data bus                                                  | (PLBC405ICURDDBUS[0:63])                                         |
| miss#                     | The ICU detects a cache miss that causes a fetch request on the PLB                                                       | ICU                                                                |                                                                  |
| fill#                     | The ICU is busy performing a fill operation                                                                               | ICU                                                                |                                                                  |
| byp#                      | The ICU forwards instructions to the PPC405 instruction-fetch unit from the fill buffer as they become available (bypass) | ICU                                                                |                                                                  |
| prefetch#                 | The ICU speculatively prefetches instructions from the BIU                                                                | ICU                                                                |                                                                  |
| Subscripts                | Used to identify the instruction words returned by a transfer                                                             | Read-data acknowledge<br>ICU read-data bus<br>ICU forward (bypass) | (PLBC405ICURDDACK)<br>(PLBC405ICURDDBUS[0:63])                   |
| #                         | Used to identify the order doublewords are sent to the ICU                                                                | Transfer order                                                     | (PLBC405ICURDWDADDR[1:3])                                        |

**Notes:**

1. “#” indicates a number.

## ISPLB Non-Pipelined Cacheable Sequential Fetch (Case 1)

The timing diagram in **Figure 2-6** shows two consecutive eight-word line fetches that are not address pipelined. The example assumes instructions are fetched sequentially from the beginning of the first line through the end of the second line.

The first line read (r11) is requested by the ICU in cycle 3 in response to a cache miss (represented by the miss1 transaction in cycles 1 and 2). Instructions are sent from the BIU to the ICU fill buffer in cycles 4 through 7. Instructions in the fill buffer are bypassed to the instruction fetch unit to prevent a processor stall during sequential execution (represented by the byp1 transaction in cycles 5 through 8). After all instructions are received, they are transferred by the ICU from the fill buffer to the instruction cache. This is represented by the fill1 transaction in cycles 9 through 11.

After the last instruction in the line is fetched, a sequential fetch from the next cache line causes a miss in cycle 13 (miss2). The second line read (r12) is requested by the ICU in cycle 15 in response to the cache miss. Instructions are sent from the BIU to the ICU fill buffer in cycles 16 through 19. Instructions in the fill buffer are bypassed to the instruction fetch unit to prevent a processor stall during sequential execution (represented by the byp2 transaction in cycles 17 through 20). After all instructions are received, they are transferred by the ICU from the fill buffer to the instruction cache (not shown).

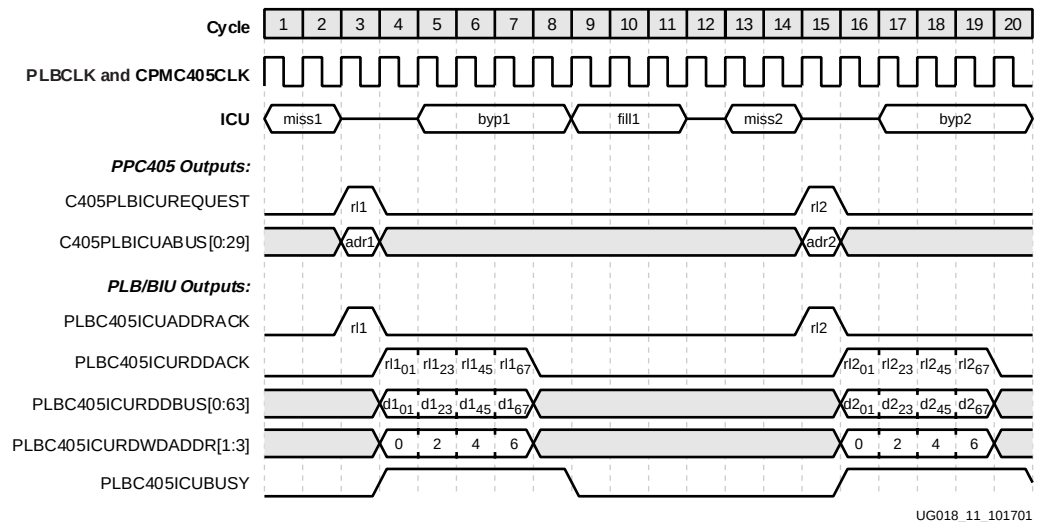


Figure 2-6: ISPLB Non-Pipelined Cacheable Sequential Fetch (Case 1)

## ISPLB Non-Pipelined Cacheable Sequential Fetch (Case 2)

The timing diagram in **Figure 2-7** shows two consecutive eight-word line fetches that are not address pipelined. The example assumes instructions are fetched sequentially from the end of the first line through the end of the second line. It provides an illustration of a transfer where the target instruction returned first by the BIU is not located at the start of the cache line.

The first line read (r1) is requested by the ICU in cycle 3 in response to a cache miss (represented by the miss1 transaction in cycles 1 and 2). Instructions are sent from the BIU to the ICU fill buffer in cycles 4 through 7. The target instruction is bypassed to the instruction fetch unit in cycle 5 (byp1). After all instructions are received, they are transferred by the ICU from the fill buffer to the instruction cache. This is represented by the fill1 transaction in cycles 8 through 10.

After the target instruction is bypassed, a sequential fetch from the next cache line causes a miss in cycle 6 (miss2). The second line read (r12) is requested by the ICU in cycle 8 in response to the cache miss. After the first line is read from the BIU, instructions for the second line are sent from the BIU to the ICU fill buffer. This occurs in cycles 9 through 12. Instructions in the fill buffer are bypassed to the instruction fetch unit to prevent a processor stall during sequential execution (represented by the byp2 transaction in cycles 11 through 13). After all instructions are received, they are transferred by the ICU from the fill buffer to the instruction cache (represented by the fill2 transaction in cycles 14 through 16).

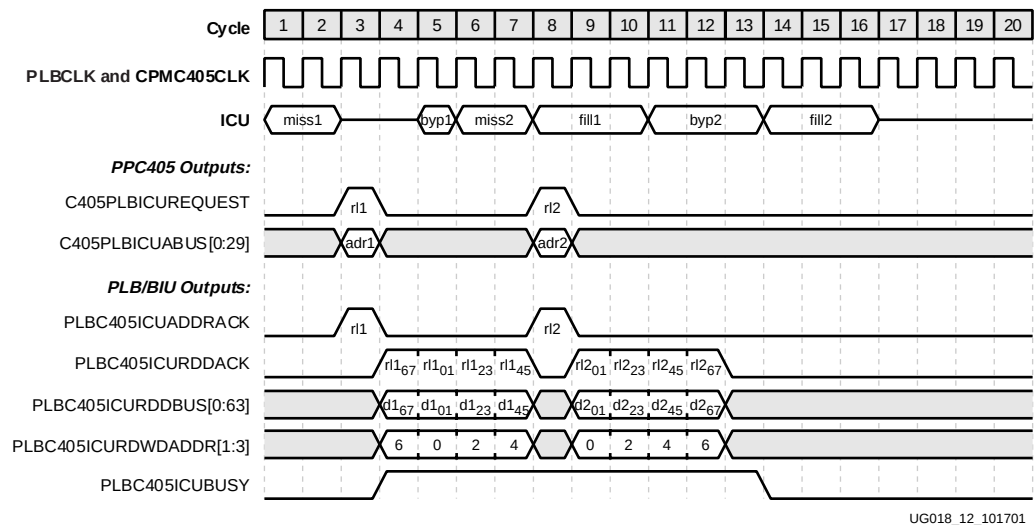


Figure 2-7: ISPLB Non-Pipelined Cacheable Sequential Fetch (Case 2)

UG018\_12\_101701

## ISPLB Pipelined Cacheable Sequential Fetch (Case 1)

The timing diagram in **Figure 2-8** shows two consecutive eight-word line fetches that are address pipelined. The example assumes instructions are fetched sequentially from the beginning of the first line through the end of the second line. It shows the fastest speed at which the ICU can request and receive instructions over the PLB.

The first line read (r1) is requested by the ICU in cycle 3 in response to a cache miss (represented by the miss1 transaction in cycles 1 and 2). Instructions are sent from the BIU to the ICU fill buffer in cycles 4 through 7. Instructions in the fill buffer are bypassed to the instruction fetch unit to prevent a processor stall during sequential execution (represented by the byp1 transaction in cycles 5 through 8). After all instructions are received, they are transferred by the ICU from the fill buffer to the instruction cache. This is represented by the fill1 transaction in cycles 9 through 11.

After the first miss is detected, the ICU performs a prefetch in anticipation of requiring instructions from the next cache line (represented by the prefetch2 transaction in cycles 3 and 4). The second line read (r2) is requested by the ICU in cycle 5 in response to the prefetch. After the first line is read from the BIU, instructions for the second line are sent from the BIU to the ICU fill buffer. This occurs in cycles 8 through 11. After all instructions are received, they are transferred by the ICU from the fill buffer to the instruction cache (represented by the fill2 transaction in cycles 13 through 15). Instructions from this second line are not bypassed because the fill buffer is transferred to the cache before the instructions are required.

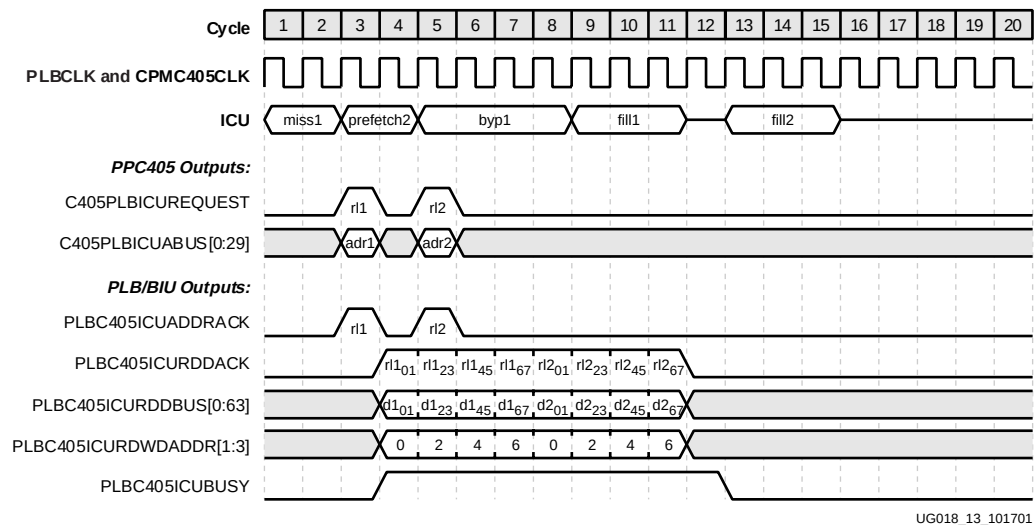


Figure 2-8: ISPLB Pipelined Cacheable Sequential Fetch (Case 1)

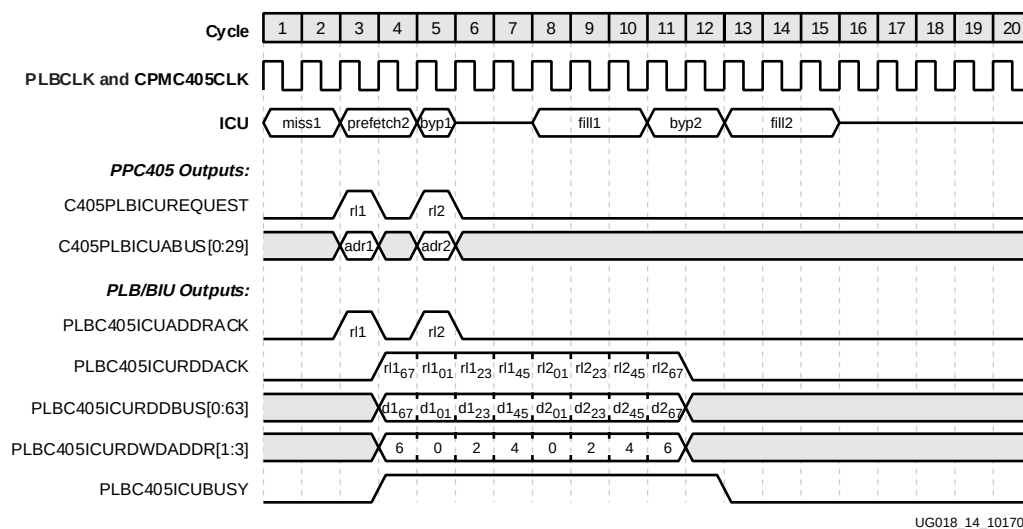
UG018\_13\_101701

## ISPLB Pipelined Cacheable Sequential Fetch (Case 2)

The timing diagram in **Figure 2-9** shows two consecutive eight-word line fetches that are address pipelined. The example assumes instructions are fetched sequentially from the end of the first line through the end of the second line. As with the previous example, it shows the fastest speed at which the ICU can request and receive instructions over the PLB. It also illustrates a transfer where the target instruction returned first by the BIU is not located at the start of the cache line.

The first line read (r11) is requested by the ICU in cycle 3 in response to a cache miss (represented by the miss1 transaction in cycles 1 and 2). Instructions are sent from the BIU to the ICU fill buffer in cycles 4 through 7. The target instruction is bypassed to the instruction fetch unit in cycle 5 (byp1). After all instructions are received, they are transferred by the ICU from the fill buffer to the instruction cache. This is represented by the fill1 transaction in cycles 8 through 10.

After the first miss is detected, the ICU performs a prefetch in anticipation of requiring instructions from the next cache line (represented by the prefetch2 transaction in cycles 3 and 4). The second line read (r12) is requested by the ICU in cycle 5 in response to the prefetch. After the first line is read from the BIU, instructions for the second line are sent from the BIU to the ICU fill buffer. This occurs in cycles 8 through 11. Instructions in the fill buffer are bypassed to the instruction fetch unit to prevent a processor stall during sequential execution (represented by the byp2 transaction in cycles 11 through 12). After all instructions are received, they are transferred by the ICU from the fill buffer to the instruction cache (represented by the fill2 transaction in cycles 13 through 15).



UG018\_14\_101701

Figure 2-9: ISPLB Pipelined Cacheable Sequential Fetch (Case 2)

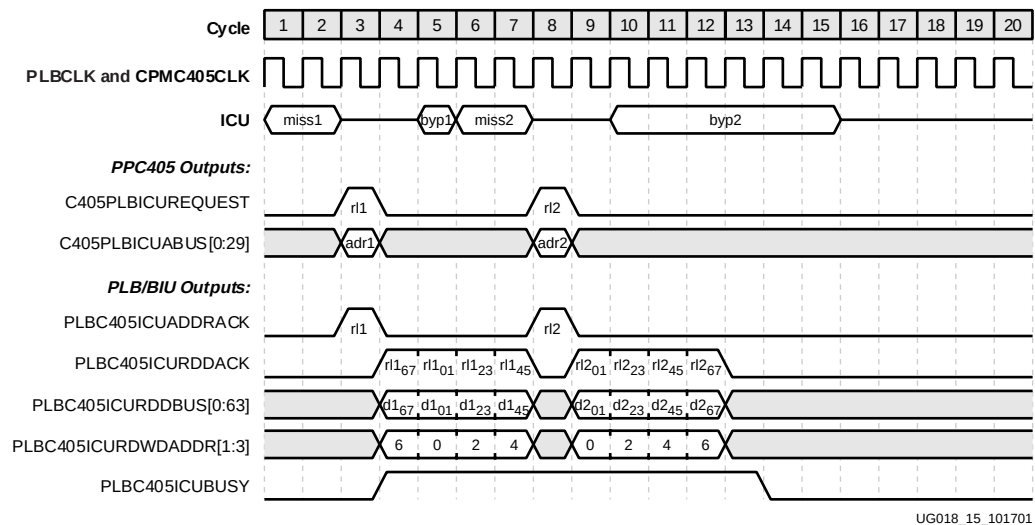


## ISPLB Non-Pipelined Non-Cacheable Sequential Fetch

The timing diagram in **Figure 2-10** shows two consecutive eight-word line fetches that are not address pipelined. The example assumes the instructions are not cacheable. It also assumes the instructions are fetched sequentially from the end of the first line through the end of the second line. It provides an illustration of how all instructions in a line must be transferred even though some of the instructions are discarded.

The first line read (r1) is requested by the ICU in cycle 3 in response to a cache miss (represented by the miss1 transaction in cycles 1 and 2). Instructions are sent from the BIU to the ICU fill buffer in cycles 4 through 7. The target instruction is bypassed to the instruction fetch unit in cycle 5 (byp1). Because the instructions are executing sequentially, the target instruction is the only instruction in the line that is executed. The line is not cacheable, so instructions are not transferred from the fill buffer to the instruction cache.

After the target instruction is bypassed, a sequential fetch from the next cache line causes a miss in cycle 6 (miss2). The second line read (r12) is requested by the ICU in cycle 8 in response to the cache miss. After the first line is read from the BIU, instructions for the second line are sent from the BIU to the ICU fill buffer. This occurs in cycles 9 through 12. These instructions overwrite the instructions from the previous line. After loading into the fill buffer, instructions from the second line are bypassed to the instruction fetch unit to prevent a processor stall during sequential execution (represented by the byp2 transaction in cycles 10 through 15). The line is not cacheable, so instructions are not transferred from the fill buffer to the instruction cache.



UG018\_15\_101701

Figure 2-10: ISPLB Non-Pipelined Non-Cacheable Sequential Fetch

## ISPLB Pipelined Non-Cacheable Sequential Fetch

The timing diagram in [Figure 2-11](#) shows two consecutive eight-word line fetches that are address pipelined. The example assumes the instructions are not cacheable. It also assumes the instructions are fetched sequentially from the end of the first line through the end of the second line. As with the previous example, it provides an illustration of how all instructions in a line must be transferred even though some of the instructions are discarded.

The first line read (r1) is requested by the ICU in cycle 3 in response to a cache miss (represented by the miss1 transaction in cycles 1 and 2). Instructions are sent from the BIU to the ICU fill buffer in cycles 4 through 7. The target instruction is bypassed to the instruction fetch unit in cycle 5 (byp1). Because the instructions are executing sequentially, the target instruction is the only instruction in the line that is executed. The line is not cacheable, so instructions are not transferred from the fill buffer to the instruction cache.

After the first miss is detected, the ICU performs a prefetch in anticipation of requiring instructions from the next cache line (represented by the prefetch2 transaction in cycles 3 and 4). The second line read (r12) is requested by the ICU in cycle 5 in response to the prefetch. After the first line is read from the BIU, instructions for the second line are sent from the BIU to the ICU fill buffer. This occurs in cycles 8 through 11. These instructions overwrite the instructions from the previous line. After loading into the fill buffer, instructions from the second line are bypassed to the instruction fetch unit to prevent a processor stall during sequential execution (represented by the byp2 transaction in cycles 9 through 14). The line is not cacheable, so instructions are not transferred from the fill buffer to the instruction cache.

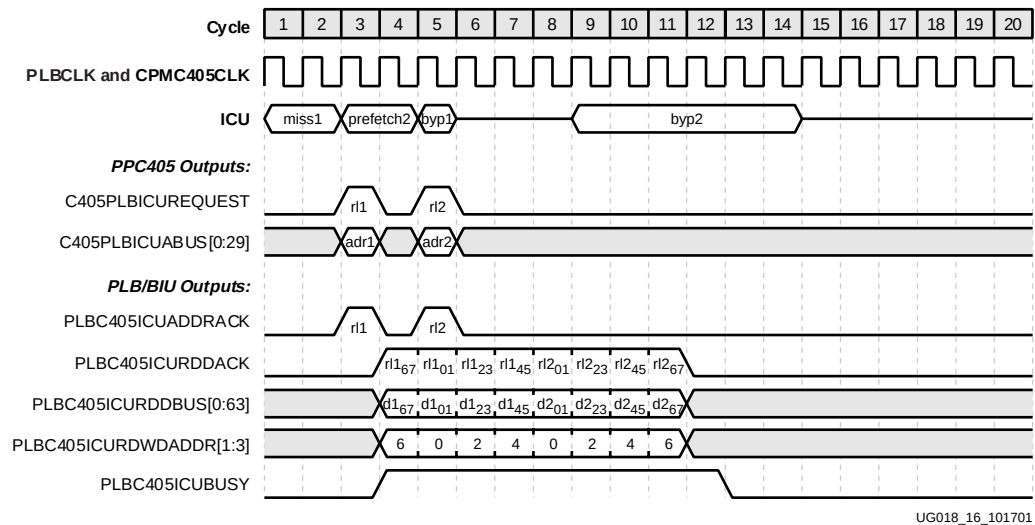
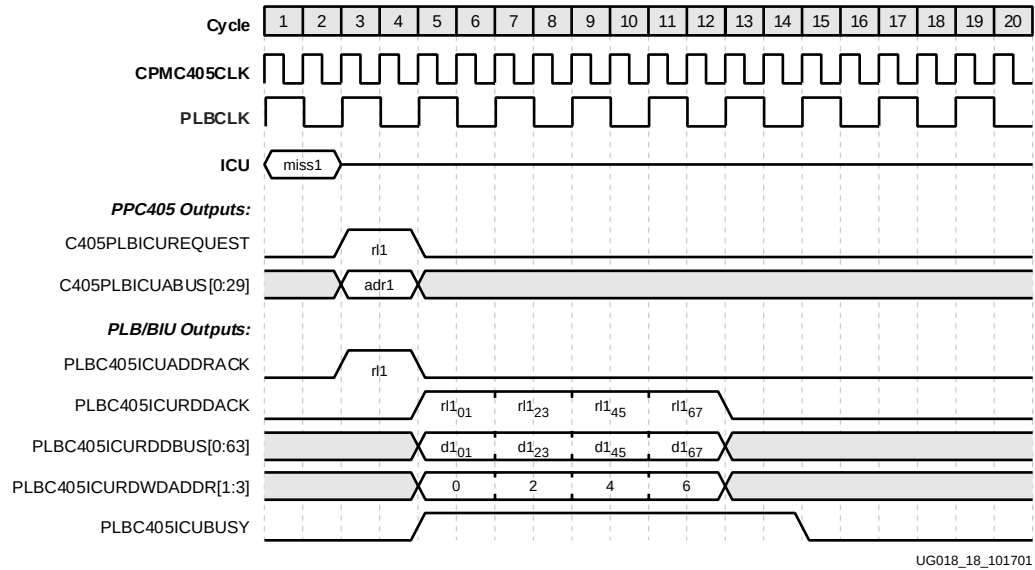


Figure 2-11: ISPLB Pipelined Non-Cacheable Sequential Fetch

## ISPLB 2:1 Core-to-PLB Line Fetch

The timing diagram in **Figure 2-12** shows an eight-word line fetch in a system with a PLB clock that runs at one half the frequency of the PPC405 clock.

The line read (r1) is requested by the ICU in PLB cycle 2, which corresponds to PPC405 cycle 3. The BIU responds in the same cycle. Instructions are sent from the BIU to the ICU fill buffer in PLB cycles 3 through 6 (PPC405 cycles 5 through 12). After all instructions associated with this line are read, the line is transferred by the ICU from the fill buffer to the instruction cache (not shown).



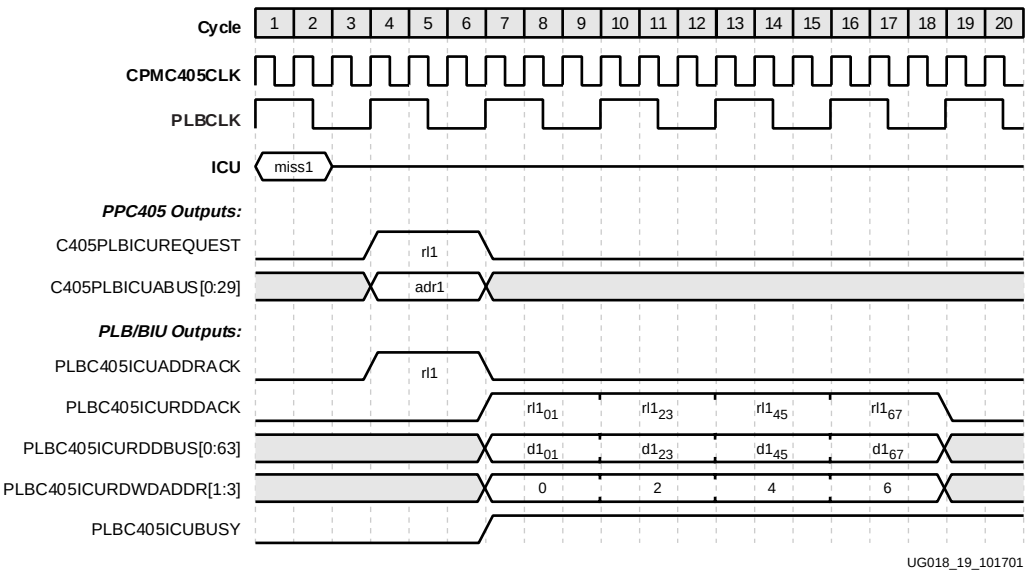
UG018\_18\_101701

Figure 2-12: ISPLB 2:1 Core-to-PLB Line Fetch

### ISPLB 3:1 Core-to-PLB Line Fetch

The timing diagram in **Figure 2-13** shows an eight-word line fetch in a system with a PLB clock that runs at one third the frequency of the PPC405 clock.

The line read (r1) is requested by the ICU in PLB cycle 2, which corresponds to PPC405 cycle 4. The BIU responds in the same cycle. Instructions are sent from the BIU to the ICU fill buffer in PLB cycles 3 through 6 (PPC405 cycles 7 through 18). After all instructions associated with this line are read, the line is transferred by the ICU from the fill buffer to the instruction cache (not shown).



UG018\_19\_101701

Figure 2-13: ISPLB 3:1 Core-to-PLB Line Fetch

## ISPLB Aborted Fetch Request

The timing diagram in [Figure 2-14](#) shows an aborted fetch request. The request is aborted because of an instruction-flow change, such as a taken branch or an interrupt. It shows the earliest-possible subsequent fetch-request that can be produced by the ICU.

The first line read (r1) is requested by the ICU in cycle 3 in response to a cache miss (represented by the miss1 transaction in cycles 1 and 2). The BIU responds in the same cycle the request is made by the ICU. However, the processor also aborts the request in cycle 3, possibly because a branch was mispredicted or an interrupt occurred. Therefore, the BIU ignores the request and does not transfer instructions associated with the request.

The change in control flow causes the ICU to fetch instructions from a non-sequential address. The second line read (r2) is requested by the ICU in cycle 7 in response to a cache miss of the new instructions. (represented by the miss2 transaction in cycles 5 and 6). Instructions are sent from the BIU to the ICU fill buffer in cycles 8 through 11.

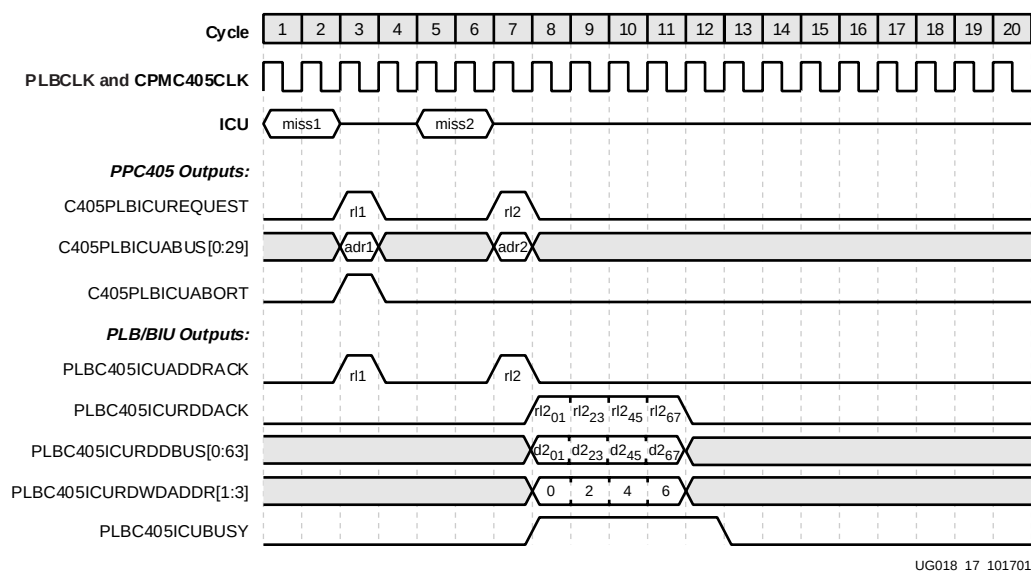


Figure 2-14: ISPLB Aborted Fetch Request

## Data-Side Processor Local Bus Interface

The data-side processor local bus (DSPLB) interface enables the PPC405 data cache unit (DCU) to load (read) and store (write) data from any memory device connected to the processor local bus (PLB). This interface has a dedicated 32-bit address bus output, a dedicated 64-bit read-data bus input, and a dedicated 64-bit write-data bus output. The interface is designed to attach as a master to a 64-bit PLB, but it also supports attachment as a master to a 32-bit PLB. The interface is capable of one data transfer (64 or 32 bits) every PLB cycle.

At the chip level, the DSPLB can be combined with the instruction-side read-data bus (also a PLB master) to create a shared read-data bus. This is done if a single PLB arbiter services both PLB masters and the PLB arbiter implementation only returns data to one PLB master at a time.

Refer to *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*) for more information on the operation of the PPC405 DCU.

### Data-Side PLB Operation

Data-access (read and write) requests are produced by the DCU and communicated over the PLB interface. A request occurs when an access misses the data cache or the memory location that is accessed is non-cacheable. A data-access request contains the following information:

- The request is indicated by C405PLBDCUREQUEST (page 969).
- The type of request (read or write) is indicated by C405PLBDCURNW (page 970).
- The target address of the data to be accessed is specified by the address bus, C405PLBDCUABUS[0:31] (page 970).
- The transfer size is specified as a single word or as eight words (cache line) using C405PLBDCUSIZE2 (page 970). The remaining bits of the transfer size (0, 1, and 3) must be tied to zero at the PLB arbiter.
- The byte enables for single-word accesses are specified using C405PLBDCUBE[0:7] (page 972). The byte enables specify one, two, three, or four contiguous bytes in either the upper or lower four byte word of the 64-bit data bus. The byte enables are not used by the processor during line transfers and must be ignored by the PLB slave.
- The cacheability storage attribute is indicated by C405PLBDCUCACHEABLE (page 970). Cacheable transfers are performed using word or line transfer sizes.
- The write-through storage attribute is indicated by C405PLBDCUWRITETHRU (page 971).
- The guarded storage attribute is indicated by C405PLBDCUGUARDED (page 971).
- The user-defined storage attribute is indicated by C405PLBDCUU0ATTR (page 971).
- The request priority is indicated by C405PLBDCUPRIORITY[0:1] (page 973). The PLB arbiter uses this information to prioritize simultaneous requests from multiple PLB masters.

The processor can abort a PLB data-access request using C405PLBDCUABORT (page 973). This occurs only when the processor is reset.

Data is returned to the DCU by a PLB slave device over the PLB interface. The response to a data-access request contains the following information:

- The address of the data-access request is acknowledged by the PLB slave using PLBC405DCUADDRACK (page 975).
- Data sent during a read transfer from the PLB slave to the DCU over the read-data bus are indicated as valid using PLBC405DCURDDACK (page 977). Data sent during a write transfer from the DCU to the PLB slave over the write-data bus are indicated as valid using PLBC405DCUWRDACK (page 978).

- The PLB-slave bus width, or size (32-bit or 64-bit), is specified by PLBC405DCUSSIZE1 (page 976). The PLB slave is responsible for packing (during reads) or unpacking (during writes) data bytes from non-word devices so that the information sent to the DCU is presented appropriately, as determined by the transfer size.
- The data transferred between the DCU and the PLB slave is sent as a single word or as an eight-word line transfer, as specified by the transfer size in the data-access request. Data reads are transferred from the PLB slave to the DCU over the DCU read-data bus, PLBC405DCURDDBUS[0:63] (page 977). Data writes are transferred from the DCU to the PLB slave over the DCU write-data bus, PLBC405DCUWRDBUS[0:63] (page 974). Data transfers operate as follows:
  - A word transfer moves the entire word specified by the address of the data-access request. The specific bytes being accessed are indicated by the byte enables, C405PLBDCUBE[0:7] (page 972). The word is transferred using one transfer operation.
  - An eight-word line transfer moves the eight-word cache line aligned on the address specified by C405PLBDCUABUS[0:26]. This cache line contains the target data accessed by the DCU. The cache line is transferred using four doubleword or eight word transfer operations, depending on the PLB slave bus width (64-bit or 32-bit, respectively). The byte enables are not used by the processor for this type of transfer and they must be ignored by the PLB slave.
- The words read during a data-read transfer can be sent from the PLB slave to the DCU in any order (target-word-first, sequential, other). This transfer order is specified by PLBC405DCURDWDADDR[1:3] (page 977). For data-write transfers, data is transferred from the DCU to the PLB slave in ascending-address order.

## Interaction with the DCU Fill Buffer

As mentioned above, the PLB slave can transfer data to the DCU in any order (target-word-first, sequential, other). When data is received by the DCU from the PLB slave, it is placed in the DCU fill buffer. When the DCU receives the target (requested) data, it forwards it immediately from the fill buffer to the load/store unit so that pipeline stalls due to load-miss delays are minimized. This operation is referred to as a *bypass*. The remaining data is received from the PLB slave and placed in the fill buffer. Subsequent data is read from the fill buffer if the data is already present in the buffer. For the best possible software performance, the PLB slave should be designed to return the target word first.

Non-cacheable data is usually transferred as a single word. Software can indicate that non-cacheable reads be loaded using an eight-word line transfer by setting the *load-word-as-line bit* in the core-configuration register (CCR0[LWL]) to 1. This enables non-cacheable reads to take advantage of the PLB line-transfer protocol to minimize PLB-arbitration delays and bus delays associated with multiple, single-word transfers. The transferred data is placed in the DCU fill buffer, but not in the data cache. Subsequent data reads from the same non-cacheable line are read from the fill buffer instead of requiring a separate arbitration and transfer sequence across the PLB. Data in the fill buffer is read with the same performance as a cache hit. The non-cacheable line remains in the fill buffer until the fill buffer is needed by another line transfer.

Non-cacheable reads from guarded storage and all non-cacheable writes are transferred as a single word, regardless of the value of CCR0[LWL].

Cacheable data is transferred as a single word or as an eight-word line, depending on whether the transfer allocates a cache line. Transfers that allocate cache lines use eight-word transfer sizes. Transfers that do not allocate cache lines use a single-word transfer size. Line allocation of cacheable data is controlled by the core-configuration register. The *load without allocate* bit CCR0[LWOA] controls line allocation for cacheable loads and the *store without allocate* bit CCR0[SWOA] controls line allocation for cacheable stores. Clearing the appropriate bit to 0 enables line allocation (this is the default) and setting the bit to 1

disables line allocation. The **dcbt** and **dcbst** instructions always allocate a cache line and ignore the CCR0 bits.

Data read during an eight-word line transfer (one that allocates a cache line) is placed in the DCU fill buffer as it is received from the PLB slave. Cacheable writes that allocate a cache line also cause an eight-word read transfer from the PLB slave. The cacheable write replaces the appropriate bytes in the fill buffer after they are read from the PLB.

Subsequent data accesses to and from the same cacheable line access the fill buffer during the time the remaining bytes are transferred from the PLB slave. When the fill buffer is full, its contents are transferred to the data cache.

An eight-word line-write transfer occurs when the fill buffer replaces an existing data-cache line containing modified data. The existing cache line is written to memory before it is replaced with the fill-buffer contents. The write is performed using a separate PLB transaction than the previous transfer that caused the replacement. Execution of the **dcbf** and **dcbst** instructions also cause an eight-word line write.

## Address Pipelining

The DCU can overlap a data-access request with a previous request. This process, known as *address pipelining*, enables a second address to be presented to a PLB slave while the slave is transferring data associated with the first address. Address pipelining can occur if a data-access request is produced before all data from a previous request are transferred by the slave. This capability maximizes PLB-transfer throughput by reducing dead cycles between multiple requests. The DCU can pipeline up to two read requests and one write request (multiple write requests cannot be pipelined). A pipelined request is communicated over the PLB two or more cycles after the prior request is acknowledged by the PLB slave.

## Unaligned Accesses

If necessary, the processor automatically decomposes accesses to unaligned operands into two data-access requests that are presented separately to the PLB. This occurs if an operand crosses a word boundary (for a word transfer) or a cache line boundary (for an eight-word line transfer). For example, assume software reads the unaligned word at address 0x1F. This word crosses a cache line boundary: the byte at address 0x1F is in one cache line and the bytes at addresses 0x20:0x22 are in another cache line. If neither cache line is in the data cache, two consecutive read requests are presented by the DCU to the PLB slave. If one cache line is already in the data cache, only the missing portion is requested by the DCU.

Because write requests are not address pipelined by the DCU, writes to unaligned data that cross cache line boundaries can take significantly longer than aligned writes.

## Guarded Storage

No bytes can be accessed speculatively from guarded storage. The PLB slave must return only the requested data when guarded storage is read and update only the specified memory locations when guarded storage is written. For single word transfers, only the bytes indicated by the byte enables are transferred. For line transfers, all eight words in the line are transferred.

## Data-Side PLB Interface I/O Signal Table

Figure 2-15 shows the block symbol for the data-side PLB interface. The signals are summarized in Table 2-12.



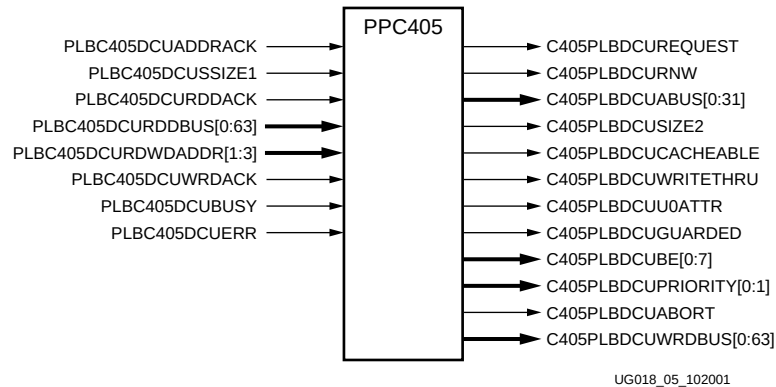


Figure 2-15: Data-Side PLB Interface Block Symbol

Table 2-12: Data-Side PLB Interface I/O Signal Summary

| Signal                  | I/O Type | If Unused                 | Function                                                                           |
|-------------------------|----------|---------------------------|------------------------------------------------------------------------------------|
| C405PLBDCUREQUEST       | O        | No Connect                | Indicates the DCU is making a data-access request.                                 |
| C405PLBDCURNW           | O        | No Connect                | Specifies whether the data-access request is a read or a write.                    |
| C405PLBDCUABUS[0:31]    | O        | No Connect                | Specifies the memory address of the data-access request.                           |
| C405PLBDCUFSIZE2        | O        | No Connect                | Specifies a single word or eight-word transfer size.                               |
| C405PLBDCUCACHEABLE     | O        | No Connect                | Indicates the value of the cacheability storage attribute for the target address.  |
| C405PLBDCUWRITETHRU     | O        | No Connect                | Indicates the value of the write-through storage attribute for the target address. |
| C405PLBDCUU0ATTR        | O        | No Connect                | Indicates the value of the user-defined storage attribute for the target address.  |
| C405PLBDCUGUARDED       | O        | No Connect                | Indicates the value of the guarded storage attribute for the target address.       |
| C405PLBDCUBE[0:7]       | O        | No Connect                | Specifies which bytes are transferred during single-word transfers.                |
| C405PLBDCUPRIORITY[0:1] | O        | No Connect                | Indicates the priority of the data-access request.                                 |
| C405PLBDCUABORT         | O        | No Connect                | Indicates the DCU is aborting an unacknowledged data-access request.               |
| C405PLBDCUWRDBUS[0:63]  | O        | No Connect                | The DCU write-data bus used to transfer data from the DCU to the PLB slave.        |
| PLBC405DCUADDRACK       | I        | 0                         | Indicates a PLB slave acknowledges the current data-access request.                |
| PLBC405DCUFSIZE1        | I        | 0                         | Specifies the bus width (size) of the PLB slave that accepted the request.         |
| PLBC405DCURDDACK        | I        | 0                         | Indicates the DCU read-data bus contains valid data for transfer to the DCU.       |
| PLBC405DCURDDBUS[0:63]  | I        | 0x0000_0000<br>_0000_0000 | The DCU read-data bus used to transfer data from the PLB slave to the DCU.         |

Table 2-12: Data-Side PLB Interface I/O Signal Summary (Continued)

| Signal                  | I/O Type | If Unused | Function                                                                                               |
|-------------------------|----------|-----------|--------------------------------------------------------------------------------------------------------|
| PLBC405DCURDWDADDR[1:3] | I        | 0b000     | Indicates which word or doubleword of an eight-word line transfer is present on the DCU read-data bus. |
| PLBC405DCUWRDACK        | I        | 0         | Indicates the data on the DCU write-data bus is being accepted by the PLB slave.                       |
| PLBC405DCUBUSY          | I        | 0         | Indicates the PLB slave is busy performing an operation requested by the DCU.                          |
| PLBC405DCUERR           | I        | 0         | Indicates an error was detected by the PLB slave during the transfer of data to or from the DCU.       |

## Data-Side PLB Interface I/O Signal Descriptions

The following sections describe the operation of the data-side PLB interface I/O signals.

Throughout these descriptions and unless otherwise noted, the term *clock* refers to the PLB clock signal, PLBCLK (see [page 1018](#) for information on this clock signal). The term *cycle* refers to a PLB cycle. To simplify the signal descriptions, it is assumed that PLBCLK and the PPC405 clock (CPMC405CLOCK) operate at the same frequency.

### C405PLBDCUREQUEST (output)

When asserted, this signal indicates the DCU is presenting a data-access request to a PLB slave device. The PLB slave asserts PLBC405DCUADDRACK to acknowledge the request. The request can be acknowledged in the same cycle it is presented by the DCU. The request is deasserted in the cycle after it is acknowledged by the PLB slave. When deasserted, no unacknowledged data-access request exists.

The following output signals contain information for the PLB slave device and are valid when the request is asserted. The PLB slave must latch these signals by the end of the same cycle it acknowledges the request:

- C405PLBDCURNW, which specifies whether the data-access request is a read or a write.
- C405PLBDCUABUS[0:31], which contains the address of the data-access request.
- C405PLBDCUSIZE2, which indicates the transfer size of the data-access request.
- C405PLBDCUCACHEABLE, which indicates whether the data address is cacheable.
- C405PLBDCUWRITETHRU, which specifies the caching policy of the data address.
- C405PLBDCUU0ATTR, which indicates the value of the user-defined storage attribute for the instruction-fetch address.
- C405PLBDCUGUARDED, which indicates whether the data address is in guarded storage.

If the transfer size is a single word, C405PLBDCUBE[0:7] is also valid when the request is asserted. These signals specify which bytes are transferred between the DCU and PLB slave. If the transfer size is an eight-word line, C405PLBDCUBE[0:7] is not used and must be ignored by the PLB slave.

C405PLBDCUPRIORITY[0:1] is valid when the request is asserted. This signal indicates the priority of the data-access request. It is used by the PLB arbiter to prioritize simultaneous requests from multiple PLB masters.

The DCU supports up to three outstanding requests over the PLB (two reads and one write). The DCU can make a subsequent request after the current request is acknowledged. The DCU deasserts C405PLBDCUREQUEST for at least one cycle after the current request is acknowledged and before the subsequent request is asserted.

If the PLB slave supports address pipelining, it must respond to multiple requests in the order they are presented by the DCU. All data associated with a prior request must be transferred before any data associated with a subsequent request is transferred. Multiple write requests are not pipelined. The DCU does not present a second write request until at least two cycles after the last write acknowledge (PLBC405DCUWRDACK) is sent from the PLB slave to the DCU, completing the first request.

The DCU only aborts a data-access request if the processor is reset. The DCU removes a request by asserting C405PLBDCUABORT while the request is asserted. In the next cycle the request is deasserted and remains deasserted until after the processor is reset.

### C405PLBDCURNW (output)

When asserted, this signal indicates the DCU is making a read request. When deasserted, this signal indicates the DCU is making a write request. This signal is valid when the DCU is presenting a data-access request to the PLB slave. The signal remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405DCUADDRACK to acknowledge the request).

### C405PLBDCUABUS[0:31] (output)

This bus specifies the memory address of the data-access request. The address is valid during the time the data-access request signal (C405PLBDCUREQUEST) is asserted. It remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405DCUADDRACK to acknowledge the request).

C405PLBDCUSIZE2 indicates the data-access transfer size. If an eight-word transfer size is used, memory-address bits [0:26] specify the aligned eight-word cache line to be transferred. If a single word transfer size is used, the byte enables (C405PLBDCUBE[0:7]) specify which bytes on the data bus are involved in the transfer.

### C405PLBDCUSIZE2 (output)

This signal specifies the transfer size of the data-access request. When asserted, an eight-word transfer size is specified. When deasserted, a single word transfer size is specified. This signal is valid when the DCU is presenting a data-access request to the PLB slave. The signal remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405DCUADDRACK to acknowledge the request).

A single word transfer moves one to four consecutive data bytes beginning at the memory address of the data-access request. For this transfer size, C405PLBDCUBE[0:7] specify which bytes on the data bus are involved in the transfer.

An eight-word line transfer moves the cache line aligned on the address specified by C405PLBDCUABUS[0:26]. This cache line contains the target data accessed by the DCU. The cache line is transferred using four doubleword or eight word transfer operations, depending on the PLB slave bus width (64-bit or 32-bit, respectively).

The words moved during an eight-word line transfer can be sent from the PLB slave to the DCU in any order (target-word-first, sequential, other). This transfer order is specified by PLBC405DCURDWDADDR[1:3].

### C405PLBDCUCACHEABLE (output)

This signal indicates whether the accessed data is cacheable. It reflects the value of the cacheability storage attribute for the target address. The data is non-cacheable when the signal is deasserted (0). The data is cacheable when the signal is asserted (1). This signal is valid when the DCU is presenting a data-access request to the PLB slave. The signal remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405DCUADDRACK to acknowledge the request).

Non-cacheable data is usually transferred as a single word. Software can indicate that non-cacheable reads be loaded using an eight-word line transfer by setting the *load-word-as-line bit* in the core-configuration register (CCR0[LWL]) to 1. This enables non-cacheable reads

to take advantage of the PLB line-transfer protocol to minimize PLB-arbitration delays and bus delays associated with multiple, single-word transfers. The transferred data is placed in the DCU fill buffer, but not in the data cache. Subsequent data reads from the same non-cacheable line are read from the fill buffer instead of requiring a separate arbitration and transfer sequence across the PLB. Data in the fill buffer are read with the same performance as a cache hit. The non-cacheable line remains in the fill buffer until the fill buffer is needed by another line transfer.

Cacheable data is transferred as a single word or as an eight-word line, depending on whether the transfer allocates a cache line. Transfers that allocate cache lines use an eight-word transfer size. Transfers that do not allocate cache lines use a single-word transfer size. Line allocation of cacheable data is controlled by the core-configuration register. The *load without allocate* bit CCR0[LWOA] controls line allocation for cacheable loads and the *store without allocate* bit CCR0[SWOA] controls line allocation for cacheable stores. Clearing the appropriate bit to 0 enables line allocation (this is the default) and setting the bit to 1 disables line allocation. The **dcbt** and **dcbtst** instructions always allocate a cache line and ignore the CCR0 bits.

### C405PLBDCUWRITETHRU (output)

This signal indicates whether the accessed data is in write-through or write-back cacheable memory. It reflects the value of the write-through storage attribute which controls the caching policy of the target address. The data is in write-back memory when the signal is deasserted (0). The data is in write-through memory when the signal is asserted (1). This signal is valid when the DCU is presenting a data-access request to the PLB slave and when the data cacheability signal is asserted. The signal remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405DCUADDRACK to acknowledge the request).

The system designer can use this signal in systems that require shared memory coherency. Stores to write-through memory update both the data cache and system memory. Stores to write-back memory update the data cache but not system memory. Write-back memory locations are updated in system memory when a cache line is flushed due to a line replacement or by executing a **dcbf** or **dcbst** instruction. See the *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*) for more information on memory coherency and caching policy.

### C405PLBDCUU0ATTR (output)

This signal reflects the value of the user-defined (U0) storage attribute for the target address. The accessed data is not in a memory location characterized by this attribute when the signal is deasserted (0). It is in a memory location characterized by this attribute when the signal is asserted (1). This signal is valid when the DCU is presenting a data-access request to the PLB slave. The signal remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405DCUADDRACK to acknowledge the request).

The system designer can use this signal to assign special behavior to certain memory addresses. Its use is optional.

### C405PLBDCUGUARDED (output)

This signal indicates whether the accessed data is in guarded storage. It reflects the value of the guarded storage attribute for the target address. The data is not in guarded storage when the signal is deasserted (0). The data is in guarded storage when the signal is asserted (1). This signal is valid when the DCU is presenting a data-access request to the PLB slave. The signal remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405DCUADDRACK to acknowledge the request).

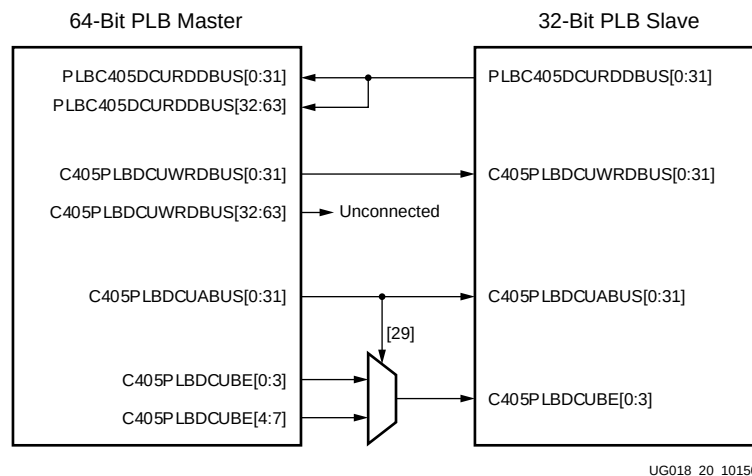
No bytes are accessed speculatively from guarded storage. The PLB slave must return only the requested data when guarded storage is read and update only the specified memory

locations when guarded storage is written. For single word transfers, only the bytes indicated by the byte enables are transferred. For line transfers, all eight words in the line are transferred.

### C405PLBDCUBE[0:7] (output)

These signals, referred to as byte enables, indicate which bytes on the DCU read-data bus or write-data bus are valid during a word transfer. The byte enables are not used by the DCU during line transfers and must be ignored by the PLB slave. The byte enables are valid when the DCU is presenting a data-access request to the PLB slave. They remain valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405DCUADDRACK to acknowledge the request).

Attachment of a 32-bit PLB slave to the DCU (a 64-bit PLB master) requires the connections shown in [Figure 2-16](#). These connections enable the byte enables to be presented properly to the 32-bit slave. Address bit 29 is used to select between the upper byte enables [0:3] and the lower byte enables [4:7] when making a request to the 32-bit slave. Words are always transferred to the 32-bit PLB slave using write-data bus bits [0:31], so bits [32:63] are not connected. The 32-bit read-data bus from the PLB slave is attached to both the high and low words of the 64-bit read-data bus into the DCU.



UG018\_20\_101501

Figure 2-16: Attachment of DSPLB Between 32-Bit Slave and 64-Bit Master

[Table 2-13](#) shows the possible values that can be presented by the byte enables and how they are interpreted by the PLB slave. All encodings of the byte enables not shown are invalid and are not generated by the DCU. The column headed “32-Bit PLB Slave Data Bus” assumes an attachment to a 64-bit PLB master as shown in [Figure 2-16](#).

Table 2-13: Interpretation of DCU Byte Enables During Word Transfers

| Byte Enables [0:7] | 32-Bit PLB Slave Data Bus |      | 64-Bit PLB Slave Data Bus |      |
|--------------------|---------------------------|------|---------------------------|------|
|                    | Valid Bytes               | Bits | Valid Bytes               | Bits |
| 1000_0000          | Byte 0                    | 0:7  | Byte 0                    | 0:7  |
| 1100_0000          | Bytes 0:1 (Halfword 0)    | 0:15 | Bytes 0:1 (Halfword 0)    | 0:15 |
| 1110_0000          | Bytes 0:2                 | 0:23 | Bytes 0:2                 | 0:23 |
| 1111_0000          | Bytes 0:3 (Word 0)        | 0:31 | Bytes 0:3 (Word 0)        | 0:31 |
| 0100_0000          | Byte 1                    | 8:15 | Byte 1                    | 8:15 |
| 0110_0000          | Bytes 1:2                 | 8:23 | Bytes 1:2                 | 8:23 |

Table 2-13: Interpretation of DCU Byte Enables During Word Transfers (Continued)

| Byte Enables [0:7] | 32-Bit PLB Slave Data Bus |       | 64-Bit PLB Slave Data Bus |       |
|--------------------|---------------------------|-------|---------------------------|-------|
|                    | Valid Bytes               | Bits  | Valid Bytes               | Bits  |
| 0111_0000          | Bytes 1:3                 | 8:31  | Bytes 1:3                 | 8:31  |
| 0010_0000          | Byte 2                    | 16:23 | Byte 2                    | 16:23 |
| 0011_0000          | Bytes 2:3 (Halfword 1)    | 16:31 | Bytes 2:3 (Halfword 1)    | 16:31 |
| 0001_0000          | Byte 3                    | 24:31 | Byte 3                    | 24:31 |
| 0000_1000          | Byte 0                    | 0:7   | Byte 4                    | 32:39 |
| 0000_1100          | Bytes 0:1 (Halfword 0)    | 0:15  | Bytes 4:5 (Halfword 2)    | 32:47 |
| 0000_1110          | Bytes 0:2                 | 0:23  | Bytes 4:6                 | 32:55 |
| 0000_1111          | Bytes 0:3 (Word 0)        | 0:31  | Bytes 4:7 (Word 1)        | 32:63 |
| 0000_0100          | Byte 1                    | 8:15  | Byte 5                    | 40:47 |
| 0000_0110          | Bytes 1:2                 | 8:23  | Bytes 5:6                 | 40:55 |
| 0000_0111          | Bytes 1:3                 | 8:31  | Bytes 5:7                 | 40:63 |
| 0000_0010          | Byte 2                    | 16:23 | Byte 6                    | 48:55 |
| 0000_0011          | Bytes 2:3 (Halfword 1)    | 16:31 | Bytes 6:7 (Halfword 3)    | 48:63 |
| 0000_0001          | Byte 3                    | 24:31 | Byte 7                    | 56:63 |

### C405PLBDCUPRIORITY[0:1] (output)

These signals are used to specify the priority of the data-access request. Table 2-8 shows the encoding of the 2-bit PLB-request priority signal. The priority is valid when the DCU is presenting a data-access request to the PLB slave. It remains valid until the cycle following acknowledgement of the request by the PLB slave (the PLB slave asserts PLBC405DCUADDRACK to acknowledge the request).

Table 2-14: PLB-Request Priority Encoding

| Bit 0 | Bit 1 | Definition                            |
|-------|-------|---------------------------------------|
| 0     | 0     | Lowest PLB-request priority.          |
| 0     | 1     | Next-to-lowest PLB-request priority.  |
| 1     | 0     | Next-to-highest PLB-request priority. |
| 1     | 1     | Highest PLB-request priority.         |

Bit 1 of the request priority is controlled by the DCU. It is asserted whenever a data-read request is presented on the PLB. The DCU can also assert this bit if the processor stalls due to an unacknowledged request. Software controls bit 0 of the request priority by writing the appropriate value into the *DCU PLB-priority bit 1* of the core-configuration register (CCR0[DPP1]).

If the least significant bits of the DCU and ICU PLB priority signals are 1 and the most significant bits are equal, the PLB arbiter should let the DCU win the arbitration. This generally results in better processor performance.

### C405PLBDCUABORT (output)

When asserted, this signal indicates the DCU is aborting the current data-access request. It is used by the DCU to abort a request that has not been acknowledged, or is in the process



of being acknowledged by the PLB slave. The data-access request continues normally if this signal is not asserted. This signal is only valid during the time the data-access request signal is asserted. It must be ignored by the PLB slave if the data-access request signal is not asserted. In the cycle after the abort signal is asserted, the data-access request signal is deasserted and remains deasserted for at least one cycle.

If the abort signal is asserted in the same cycle that the data-access request is acknowledged by the PLB slave (PLBC405DCUADDRACK is asserted), the PLB slave is responsible for ensuring that the transfer does not proceed further. The PLB slave must not assert the DCU read-data bus acknowledgement signal for an aborted request. It is possible for a PLB slave to return the first write acknowledgement when acknowledging an aborted data-write request. In this case, memory must not be updated by the PLB slave and no further write acknowledgements can be presented by the PLB slave for the aborted request.

The DCU only aborts a data-access request when the processor is reset. Such an abort can occur during an address-pipelined data-access request while the PLB slave is responding to a previous data-access request. If the PLB is not also reset (as is the case during a core reset), the PLB slave is responsible for completing the previous request and aborting the new (pipelined) request.

### PLBC405DCUWRDBUS[0:63] (output)

This write-data bus contains the data transferred from the DCU to a PLB slave during a write transfer. The operation of this bus depends on the transfer size, as follows:

- During a single word write, the write-data bus is valid when the write request is presented by the DCU. The data remains valid until the PLB slave accepts the data. The PLB slave asserts the write-data acknowledgement signal when it latches data transferred on the write-data bus, indicating that it accepts the data. This completes the word write.

The DCU replicates the data on the high and low words of the write data bus (bits [0:31] and [32:63], respectively) during a single word write. The byte enables indicate which bytes on the high word or low word are valid and should be latched by the PLB slave.

- During an eight-word line transfer, the write-data bus is valid when the write request is presented by the DCU. The data remains valid until the PLB slave accepts the data. The PLB slave asserts the write-data acknowledgement signal when it latches data transferred on the write-data bus, indicating that it accepts the data. In the cycle after the PLB slave accepts the data, the DCU presents the next word or doubleword of data (depending on the PLB slave size). Again, the PLB slave asserts the write-data acknowledgement signal when it latches data transferred on the write-data bus, indicating that it accepts the data. This continues until all eight words are transferred to the PLB slave.

Data is transferred from the DCU to the PLB slave in ascending address order. Word 0 (lowest address of the cache line) is transferred first and word 7 (highest address) is transferred last. The byte enables are not used during a line transfer and must be ignored by the PLB slave.

The location of data on the write-data bus depends on the size of the PLB slave, as follows:

- If the slave has a 64-bit bus, the DCU transfers even words (words 0, 2, 4, and 6) on write-data bus bits [0:31] and odd words (words 1, 3, 5, and 7) on write-data bus bits [32:63]. Four doubleword writes are required to complete the eight-word line transfer. The first transfer writes words 0 and 1, the second transfer writes words 2 and 3, and so on.
- If the slave has a 32-bit bus, the DCU transfers all words on write-data bus bits [0:31]. Eight doubleword writes are required to complete the eight-word line

transfer. The first transfer writes word 0, the second transfer writes word 1, and so on.

**Table 2-15** summarizes the location of words on the write-data bus during an eight-word line transfer.

**Table 2-15: Contents of DCU Write-Data Bus During Eight-Word Line Transfer**

| PLB-Slave Size | Transfer | DCU Write-Data Bus [0:31] | DCU Write-Data Bus [32:63] |
|----------------|----------|---------------------------|----------------------------|
| 32-Bit         | First    | Word 0                    | Not Applicable             |
|                | Second   | Word 1                    |                            |
|                | Third    | Word 2                    |                            |
|                | Fourth   | Word 3                    |                            |
|                | Fifth    | Word 4                    |                            |
|                | Sixth    | Word 5                    |                            |
|                | Seventh  | Word 6                    |                            |
|                | Eighth   | Word 7                    |                            |
| 64-Bit         | First    | Word 0                    | Word 1                     |
|                | Second   | Word 2                    | Word 3                     |
|                | Third    | Word 4                    | Word 5                     |
|                | Fourth   | Word 6                    | Word 7                     |

### PLBC405DCUADDRACK (input)

When asserted, this signal indicates the PLB slave acknowledges the DCU data-access request (indicated by the DCU assertion of C405PLBDCUREQUEST). When deasserted, no such acknowledgement exists. A data-access request can be acknowledged by the PLB slave in the same cycle the request is asserted by the DCU. The PLB slave must latch the following data-access request information in the same cycle it asserts the request acknowledgement:

- C405PLBDCURNW, which specifies whether the data-access request is a read or a write.
- C405PLBDCUABUS[0:31], which contains the address of the data-access request.
- C405PLBDCUSIZE2, which indicates the transfer size of the data-access request.
- C405PLBDCUCACHEABLE, which indicates whether the data address is cacheable.
- C405PLBDCUWRITETHRU, which specifies the caching policy of the data address.
- C405PLBDCUU0ATTR, which indicates the value of the user-defined storage attribute for the instruction-fetch address.
- C405PLBDCUGUARDED, which indicates whether the data address is in guarded storage.

During the acknowledgement cycle, the PLB slave must return its bus width indicator (32 bits or 64 bits) using the PLBC405DCUFSIZE1 signal.

The acknowledgement signal remains asserted for one cycle. In the next cycle, both the data-access request and acknowledgement are deasserted. The PLB slave can begin receiving data from the DCU in the same cycle the address is acknowledged. Data can be sent to the DCU beginning in the cycle after the address acknowledgement. The PLB slave must abort a DCU request (move no data) if the DCU asserts C405PLBDCUABORT in the same cycle the PLB slave acknowledges the request.



The DCU supports up to three outstanding requests over the PLB (two read and one write). The DCU can make a subsequent request after the current request is acknowledged. The DCU deasserts C405PLBDCUREQUEST for at least one cycle after the current request is acknowledged and before the subsequent request is asserted.

If the PLB slave supports address pipelining, it must respond to multiple requests in the order they are presented by the DCU. All data associated with a prior request must be moved before data associated with a subsequent request is accessed. The DCU cannot present a third read request until the first read request is completed by the PLB slave, or a second write request until the first write request is completed. Such a request (third read or second write) can be presented two cycles after the last acknowledge is sent from the PLB slave to the DCU, completing the first request (read or write, respectively).

### PLBC405DCUFSIZE1 (input)

This signal indicates the bus width (size) of the PLB slave device that acknowledged the DCU request. A 32-bit PLB slave responded when the signal is deasserted (0). A 64-bit PLB slave responded when the signal is asserted (1). This signal is valid during the cycle the acknowledge signal (PLBC405DCUADDRACK) is asserted.

A 32-bit PLB slave must be attached to a 64-bit PLB master as shown in [Figure 2-16, page 972](#). In this figure, the 32-bit read-data bus from the PLB slave is attached to both the high word and low word of the 64-bit read-data bus at the PLB master. The 32-bit write-data bus into the PLB slave is attached to the high word of the 64-bit write-data bus at the PLB master. The low word of the 64-bit write-data bus is not connected. When a 64-bit PLB master recognizes a 32-bit PLB slave (the size signal is deasserted), data transfers operate as follows:

- During a single word read, data is received by the 64-bit master over the high word (bits 0:31) or the low word (bits 32:63) of the read-data bus as specified by the byte enable signals.
- During an eight-word line read, data is received by the 64-bit master over the high word (bits 0:31) or the low word (bits 32:63) of the read-data bus as specified by bit 3 of the transfer order (PLBC405DCURDWDADDR[1:3]). [Table 2-10, page 952](#), shows the location of data on the DCU read-data bus as a function of transfer order when an eight-word line read from a 32-bit PLB slave occurs.
- During a single word write or an eight-word line write, data is sent by the 64-bit master over the high word (bits 0:31) of the write-data bus. [Table 2-15, page 975](#), shows the order data is transferred to a 32-bit PLB slave during an eight-word line write.

All bits of the read-data bus and write-data bus are directly connected between a 64-bit PLB slave and a 64-bit PLB master. When a 64-bit PLB master recognizes a 64-bit PLB slave (the size signal is asserted), data transfers operate as follows:

- During a single word read, data is received by the 64-bit master over the high word (bits 0:31) or the low word (bits 32:63) of the read-data bus as specified by the byte enable signals.
- During an eight-word line read, data is received by the 64-bit master over the entire read-data bus. [Table 2-10, page 952](#), shows the location of data on the DCU read-data bus as a function of transfer order when an eight-word line read from a 64-bit PLB slave occurs.
- During a single word write, the DCU replicates the data on the high and low words of the write data bus. The byte enables indicate which bytes on the high word or low word are valid and should be latched by the PLB slave.
- During an eight-word line write, data is sent by the 64-bit master over the entire write-data bus. [Table 2-15, page 975](#), shows the order data is transferred to a 64-bit PLB slave during an eight-word line write. Data is written in order of ascending address, so the transfer order signals are not used during a line write.

## PLBC405DCURDDACK (input)

When asserted, this signal indicates the DCU read-data bus contains valid data sent by the PLB slave to the DCU (read data is acknowledged). The DCU latches the data from the bus at the end of the cycle this signal is asserted. The contents of the DCU read-data bus are not valid when this signal is deasserted.

Read-data acknowledgement is asserted for one cycle per transfer. There is no limit to the number of cycles between two transfers. The number of transfers (and the number of read-data acknowledgements) depends on the PLB slave size (specified by PLBC405DCUFSIZE1) and the line-transfer size (specified by C405PLBDCUSIZE2). The number of transfers are summarized as follows:

- Single word reads require one transfer, regardless of the PLB slave size.
- Eight-word line reads require eight transfers when sent from a 32-bit PLB slave.
- Eight-word line reads require four transfers when sent from a 64-bit PLB slave.

## PLBC405DCURDDBUS[0:63] (input)

This read-data bus contains the data transferred from a PLB slave to the DCU. The contents of the bus are valid when the read-data acknowledgement signal is asserted. This acknowledgment is asserted for one cycle per transfer. There is no limit to the number of cycles between two transfers. The bus contents are not valid when the read-data acknowledgement signal is deasserted.

The PLB slave returns data as an aligned word or an aligned doubleword. This depends on the PLB slave size (bus width), as follows:

- When a 32-bit PLB slave responds, an aligned word is sent from the slave to the DCU during each transfer cycle. The 32-bit PLB slave bus should be connected to both the high and low 32 bits of the 64-bit read-data bus (see [Figure 2-16, page 972](#)). This type of connection duplicates the word returned by the slave across the 64-bit bus. The DCU reads either the low 32 bits or the high 32 bits of the 64-bit interface, depending on the value of PLBC405DCURDWDADDR[1:3].
- When a 64-bit PLB slave responds, an aligned doubleword is sent from the slave to the DCU during each transfer cycle. Both words are read from the 64-bit interface by the DCU in this cycle.

For a single word transfer, the bytes enables are used to select the valid data bytes from the aligned word or doubleword. [Table 2-13, page 972](#) shows how the byte enables are interpreted by the processor when reading data during single word transfers from 32-bit and 64-bit PLB slaves. [Table 2-10](#) shows the location of data on the DCU read-data bus as a function of PLB-slave size and transfer order when an eight-word line read occurs.

## PLBC405DCURDWDADDR[1:3] (input)

These signals are used to specify the transfer order. They identify which word or doubleword of an eight-word line transfer is present on the DCU read-data bus when the PLB slave returns instructions to the DCU. The words returned during a line transfer can be sent from the PLB slave to the DCU in any order (target-word-first, sequential, other). The transfer-order signals are valid when the read-data acknowledgement signal (PLBC405DCURDDACK) is asserted. This acknowledgment is asserted for one cycle per transfer. There is no limit to the number of cycles between two transfers. The transfer-order signals are not valid when the read-data acknowledgement signal is deasserted.

These signals are ignored by the processor during single word transfers.

[Table 2-10](#) shows the location of data on the DCU read-data bus as a function of PLB-slave size and transfer order when an eight-word line read occurs. In this table, the “Transfer Order” column contains the possible values of PLBC405DCURDWDADDR[1:3]. For 64-bit PLB slaves, PLBC405DCURDWDADDR[3] should always be 0 during a transfer. In this

case, the transfer order is invalid if this signal asserted. For 32-bit slaves, the connection to a 64-bit master shown in [Figure 2-16, page 972](#) is assumed.

**Table 2-16: Contents of DCU Read-Data Bus During Eight-Word Line Transfer**

| PLB-Slave Size | Transfer Order <sup>1</sup> | DCU Read-Data Bus [0:31] <sup>2</sup> | DCU Read-Data Bus [32:63] <sup>2</sup> |
|----------------|-----------------------------|---------------------------------------|----------------------------------------|
| 32-Bit         | 000                         | Word 0                                | <i>Word 0</i>                          |
|                | 001                         | <i>Word 1</i>                         | Word 1                                 |
|                | 010                         | Word 2                                | <i>Word 2</i>                          |
|                | 011                         | <i>Word 3</i>                         | Word 3                                 |
|                | 100                         | Word 4                                | <i>Word 4</i>                          |
|                | 101                         | <i>Word 5</i>                         | Word 5                                 |
|                | 110                         | Word 6                                | <i>Word 6</i>                          |
|                | 111                         | <i>Word 7</i>                         | Word 7                                 |
| 64-Bit         | 000                         | Word 0                                | Word 1                                 |
|                | 010                         | Word 2                                | Word 3                                 |
|                | 100                         | Word 4                                | Word 5                                 |
|                | 110                         | Word 6                                | Word 7                                 |
|                | xx1                         | Invalid                               |                                        |

Notes:

1. “x” indicates a don’t-care value in PLBC405DCURDWDADDR[1:3].

2. A word shown in *italics* is ignored by the DCU during the transfer.

### PLBC405DCUWRDACK (input)

When asserted, this signal indicates the PLB slave latched the data on the write-data bus sent from the DCU (write data is acknowledged). The DCU holds this data valid until the end of the cycle this signal is asserted. In the following cycle, the DCU presents new data and holds it valid until acknowledged by the PLB slave. This continues until all write data is transferred from the DCU to the PLB slave. If this signal is deasserted, valid data on the write data bus has not been latched by the PLB slave.

Write-data acknowledgement is asserted for one cycle per transfer. There is no limit to the number of cycles between two transfers. The number of transfers (and the number of write-data acknowledgements) depends on the PLB slave size (specified by PLBC405DCUFSIZE1 and the line-transfer size (specified by C405PLBDCUFSIZE2). The number of transfers are summarized as follows:

- Single word writes require one transfer, regardless of the PLB slave size.
- Eight-word line writes require eight transfers when sent to a 32-bit PLB slave.
- Eight-word line writes require four transfers when sent to a 64-bit PLB slave.

### PLBC405DCUBUSY (input)

When asserted, this signal indicates the PLB slave acknowledged and is responding to (is busy with) a DCU data-access request. When deasserted, the PLB slave is not responding to a DCU data-access request.

This signal should be asserted in the cycle after a DCU request is acknowledged by the PLB slave and remain asserted until the request is completed by the PLB slave. For read requests, it should be deasserted in the cycle after the last read-data acknowledgement. For

write requests, it should be deasserted in the cycle after the target memory device is updated by the PLB slave. If multiple requests are initiated and overlap, the busy signal should be asserted in the cycle after the first request is acknowledged and remain asserted until the cycle after the last request is completed.

The processor monitors the busy signal when executing a **sync** instruction. The **sync** instruction requires that all storage operations initiated prior to the **sync** be completed before subsequent instructions are executed. Storage operations are considered complete when there are no pending DCU requests and the busy signal is deasserted.

Following reset, the processor block prevents the DCU from accessing data until the busy signal is deasserted for the first time. This is useful in situations where the processor block is reset by a core reset, but PLB devices are not reset. Waiting for the busy signal to be deasserted prevents data accesses following reset from interfering with PLB activity that was initiated before reset.

### PLBC405DCUERR (input)

When asserted, this signal indicates the PLB slave detected an error when attempting to transfer data to or from the DCU. The error signal should be asserted for only one cycle. When deasserted, no error is detected.

For read operations, this signal should be asserted with the read-data acknowledgement signal that corresponds to the erroneous transfer. For write operations, it is possible for the error to not be detected until some time after the data is accepted by the PLB slave. Thus, the signal can be asserted independently of the write-data acknowledgement signal that corresponds to the erroneous transfer. However, it must be asserted while the busy signal is asserted.

The PLB slave must not terminate data transfers when an error is detected. The processor block is responsible for responding to any error detected by the PLB slave. A machine-check exception occurs if the exception is enabled by software (MSR[ME]=1) and data is transferred between the processor block and a PLB slave while the error signal is asserted.

The PLB slave should latch error information in DCRs so that software diagnostic routines can attempt to report and recover from the error. A bus-error address register (BEAR) should be implemented for storing the address of the access that caused the error. A bus-error syndrome register (BESR) should be implemented for storing information about cause of the error.

## Data-Side PLB Interface Timing Diagrams

The following timing diagrams show typical transfers that can occur on the DSPLB interface between the DCU and a bus-interface unit (BIU). These timing diagrams represent the optimal timing relationships supported by the processor block. The BIU can be implemented using the FPGA processor local bus (PLB) or using customized hardware. Not all BIU implementations support these optimal timing relationships.

### DSPLB Timing Diagram Assumptions

The following assumptions and simplifications were made in producing the optimal timing relationships shown in the timing diagrams:

- Requests are acknowledged by the BIU in the same cycle they are presented by the DCU if the BIU is not busy. This represents the earliest cycle a BIU can acknowledge a request. If the BIU is busy, the request is acknowledged in a later cycle.
- The first read-data acknowledgement for a data read is asserted in the cycle immediately following the read-request acknowledgement. This represents the earliest cycle a BIU can begin transferring data to the DCU in response to a read request. However, the earliest the FPGA PLB begins transferring data is *two cycles* after the read request is acknowledged.
- Subsequent read-data acknowledgements for eight-word line transfers are asserted in the cycle immediately following the prior read-data acknowledgement. This represents the fastest rate at which a BIU can transfer data to the DCU (there is no limit to the number of cycles between two transfers).
- The first write-data acknowledgement for a data write is asserted in the same cycle as the write-request acknowledgement. This represents the earliest cycle a BIU can begin accepting data from the DCU in response to a write request.
- Subsequent write-data acknowledgements for eight-word line transfers are asserted in the cycle immediately following the prior write-data acknowledgement. This represents the fastest rate at which the DCU can transfer data to the BIU (there is no limit to the number of cycles between two transfers).
- All eight-word line reads assume the target data (word) is returned first. Subsequent data in the line is returned sequentially by address, wrapping as necessary to the lower addresses in the same line.
- The transfer of read data from the fill buffer to the data cache (fill operation) takes three cycles. This transfer takes place after all data is read into the fill buffer from the BIU.
- The queuing of data flushed from the data cache (flush operation) takes two cycles. The PPC405 can queue up to two flush operations.
- The BIU size (bus width) is 64 bits, so PLBC405DCUFSIZE1 is not shown.
- No data-access errors occur, so PLBC405DCUERR is not shown.
- The abort signal, C405PLBDCUABORT is shown only in the last example.
- The storage attribute signals are not shown.
- The DCU activity is shown only as an aide in describing the examples. The occurrence and duration of this activity is not observable on the DSPLB.

The following abbreviations appear in the timing diagrams:

Table 2-17: Key to DSPLB Timing Diagram Abbreviations

| Abbreviation <sup>1</sup>                   | Description                                                                                    | Where Used                                                                                 |                                                                                                  |
|---------------------------------------------|------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| rl#, wl#                                    | Eight-word line read-request or write-request identifier, respectively                         | Request<br>Request acknowledge<br>Read-data acknowledge<br>Write-data acknowledge          | (C405PLBDCUREQUEST)<br>(PLBC405DCUADDRACK)<br>(PLBC405DCURDDACK)<br>(PLBC405DCUWRDACK)           |
| rw#, ww#                                    | Single word read-request or write-request identifier, respectively                             | Request<br>Request acknowledge<br>Read-data acknowledge<br>Write-data acknowledge          | (C405PLBDCUREQUEST)<br>(PLBC405DCUADDRACK)<br>(PLBC405DCURDDACK)<br>(PLBC405DCUWRDACK)           |
| adr#                                        | Data-access request address                                                                    | Request address                                                                            | (C405PLBDCUABUS[0:31])                                                                           |
| d# <sub>#</sub>                             | A doubleword (eight data bytes) transferred as a result of an eight-word line transfer request | DCU read-data bus<br>DCU write-data bus                                                    | (PLBC405DCURDDBUS[0:63])<br>(PLBC405DCUWRDBUS[0:63])                                             |
| d#                                          | A word (four data bytes) transferred as a result of a single word transfer request             | DCU read-data bus<br>DCU write-data bus                                                    | (PLBC405DCURDDBUS[0:63])<br>(PLBC405DCUWRDBUS[0:63])                                             |
| val                                         | Byte enables are valid                                                                         | Byte enables                                                                               | (C405PLBDCUBE[0:7])                                                                              |
| flush#                                      | The DCU is busy performing a flush operation                                                   | DCU                                                                                        |                                                                                                  |
| fill#                                       | The DCU is busy performing a fill operation                                                    | DCU                                                                                        |                                                                                                  |
| Subscripts                                  | Used to identify the data words transferred between the BIU and DCU                            | Read-data acknowledge<br>DCU read-data bus<br>Write-data acknowledge<br>DCU write-data bus | (PLBC405DCURDDACK)<br>(PLBC405DCURDDBUS[0:63])<br>(PLBC405DCUWRDACK)<br>(PLBC405DCUWRDBUS[0:63]) |
| #                                           | Used to identify the order doublewords are sent to the DCU                                     | Transfer order                                                                             | (PLBC405DCURDWDADDR[1:3])                                                                        |
| <b>Notes:</b><br>1. “#” indicates a number. |                                                                                                |                                                                                            |                                                                                                  |

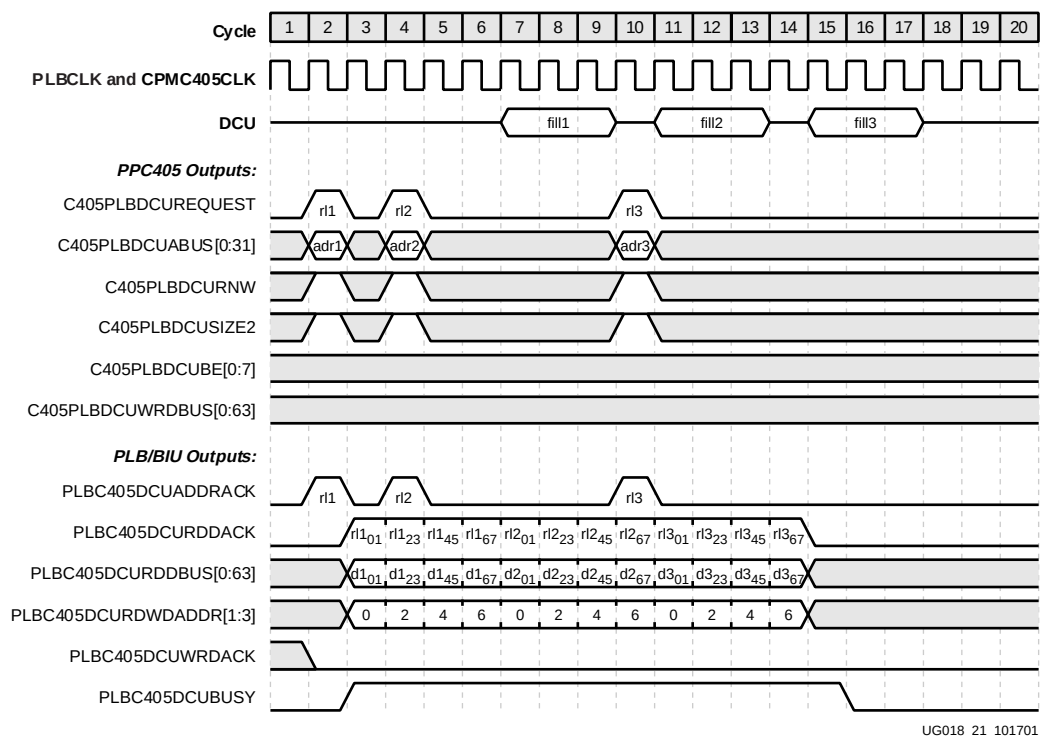
## DSPLB Three Consecutive Line Reads

The timing diagram in **Figure 2-17** shows three consecutive eight-word line reads that are address-pipelined between the DCU and BIU. It provides an example of the fastest speed at which the DCU can request and receive data over the PLB. All reads are cacheable.

The first line read (r1) is requested by the DCU in cycle 2. Data is sent from the BIU to the DCU fill buffer in cycles 3 through 6. After all data associated with this line is read, it is transferred by the DCU from the fill buffer to the data cache. This is represented by the fill1 transaction in cycles 7 through 9.

The second line read (r12) is requested by the DCU in cycle 4. The BIU responds to this request after it has completed all transactions associated with the first request (r11). Data is sent from the BIU to the DCU fill buffer in cycles 7 through 10. After all data associated with this line is read, it is transferred by the DCU from the fill buffer to the data cache. This is represented by the fill2 transaction in cycles 11 through 13.

The third line read (r13) cannot be requested until the first request (r11) is complete. The earliest this request can occur is in cycle 7. However, the request is delayed to cycle 10 because the DCU is busy transferring the fill buffer to the data cache in cycles 7 through 9 (fill1). The BIU responds to the r13 request after it has completed all transactions associated with the second request (r12). Data is sent from the BIU to the DCU fill buffer in cycles 11 through 14. After all data associated with this line is read, it is transferred by the DCU from the fill buffer to the data cache. This is represented by the fill3 transaction in cycles 15 through 17.



UG018\_21\_101701

Figure 2-17: DSPLB Three Consecutive Line Reads



## DSPLB Line Read/Word Read/Line Read

The timing diagram in **Figure 2-18** shows a sequence involving an eight-word line read, a word read, and another an eight-word line read. These requests are address-pipelined between the DCU and BIU. The line reads are cacheable and the word read is not cacheable.

The first line read (rl1) is requested by the DCU in cycle 2 and the BIU responds in the same cycle. Data is sent from the BIU to the DCU fill buffer in cycles 3 through 6. After all data associated with this line is read, it is transferred by the DCU from the fill buffer to the data cache. This is represented by the fill1 transaction in cycles 7 through 9.

The word read (rw2) is requested by the DCU in cycle 4. The BIU responds to this request after it has completed all transactions associated with the first request (rl1). A single word is sent from the BIU to the DCU fill buffer in cycle 7. The DCU uses the byte enables to select the appropriate bytes from the read-data bus. The data is not cacheable, so the fill buffer is not transferred to the data cache after this transaction is completed.

The third line read (rl3) cannot be requested until the first request (rl1) is complete. The earliest this request can occur is in cycle 7. However, the request is delayed to cycle 10 because the DCU is busy transferring the fill buffer to the data cache in cycles 7 through 9 (fill1). The BIU can respond immediately to the rl3 request because all transactions associated with the second request (rw2) are complete. Data is sent from the BIU to the DCU fill buffer in cycles 11 through 14. After all data associated with this line is read, it is transferred by the DCU from the fill buffer to the data cache. This is represented by the fill3 transaction in cycles 15 through 17.

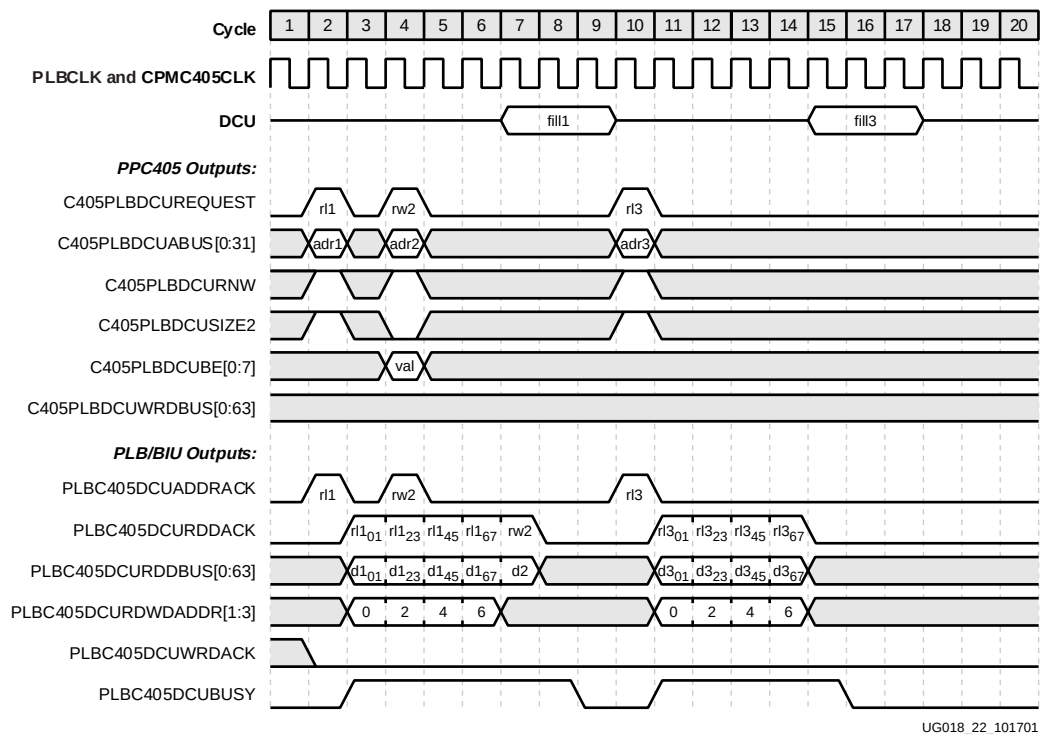


Figure 2-18: DSPLB Line Read/Word Read/Line Read



## DSPLB Three Consecutive Word Reads

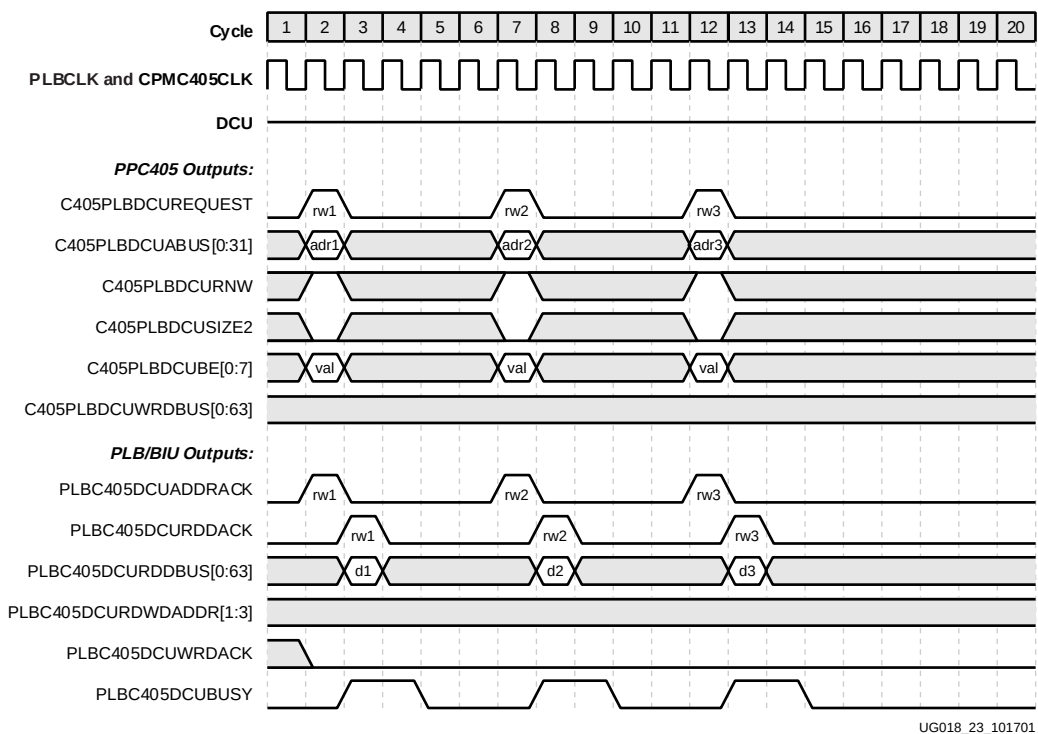
The timing diagram in **Figure 2-19** shows three consecutive word reads. The word reads could be in response to non-cacheable loads or cacheable loads that do not allocate a cache line.

**Figure 2-19** provides an example of the fastest speed at which the PPC405 DCU can request and receive single words over the PLB. The DCU is designed to wait for the current single-word read request to be satisfied before making a subsequent request. This requirement results in the delay between requests shown in the figure. It is possible for other PLB masters to request and receive single words at a faster rate than shown in this example.

The first word read (rw1) is requested by the DCU in cycle 2 and the BIU responds in the same cycle. A single word is sent from the BIU to the DCU in cycle 3. The DCU uses the byte enables to select the appropriate bytes from the read-data bus.

The second word read (rw2) is requested by the DCU in cycle 7 and the BIU responds in the same cycle. A single word is sent from the BIU to the DCU in cycle 8. The DCU uses the byte enables to select the appropriate bytes from the read-data bus.

The third word read (rw3) is requested by the DCU in cycle 12 and the BIU responds in the same cycle. A single word is sent from the BIU to the DCU in cycle 13. The DCU uses the byte enables to select the appropriate bytes from the read-data bus.



UG018\_23\_101701

Figure 2-19: DSPLB Three Consecutive Word Reads

## DSPLB Three Consecutive Line Writes

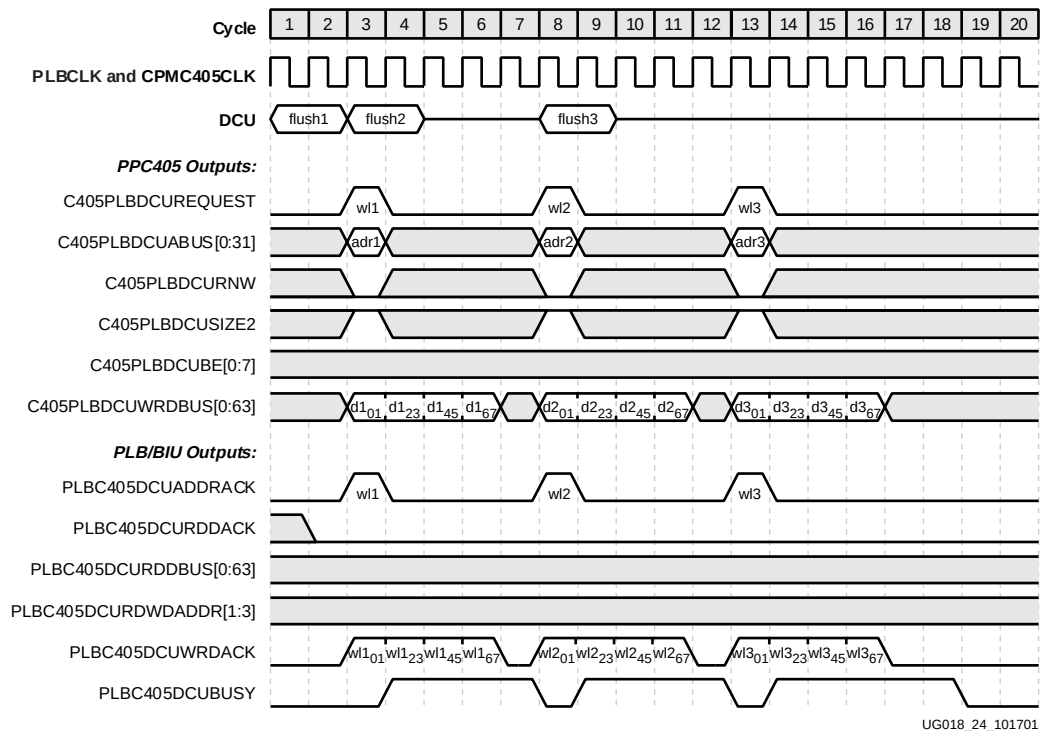
The timing diagram in [Figure 2-20](#) shows three consecutive eight-word line writes. It provides an example of the fastest speed at which the DCU can request and send data over the PLB. All writes are cacheable. Consecutive writes cannot be address pipelined between the DCU and BIU.

The first line write (wl1) is requested by the DCU in cycle 3 in response to a cache flush (represented by the flush1 transaction in cycles 1 through 2). The BIU responds in the same cycle the request is made by the DCU. Data is sent from the DCU to the BIU in cycles 3 through 6.

The second line write (wl2) cannot be started until the first request is complete. This request is made by the DCU in cycle 8 in response to the cache flush in cycles 3 through 4 (flush2). The BIU responds in the same cycle the request is made by the DCU. Data is sent from the DCU to the BIU in cycles 8 through 11.

The DCU can queue two outstanding data-cache flush requests. In this example, a third flush request cannot be queued until the first is complete. The third flush request (flush3) is queued in cycles 8 and 9.

The third line write (wl3) cannot be started until the second request (wl2) is complete. This request is made by the DCU in cycle 13 in response to the flush3 request. The BIU responds in the same cycle the request is made by the DCU. Data is sent from the DCU to the BIU in cycles 13 through 16.



UG018\_24\_101701

Figure 2-20: DSPLB Three Consecutive Line Writes

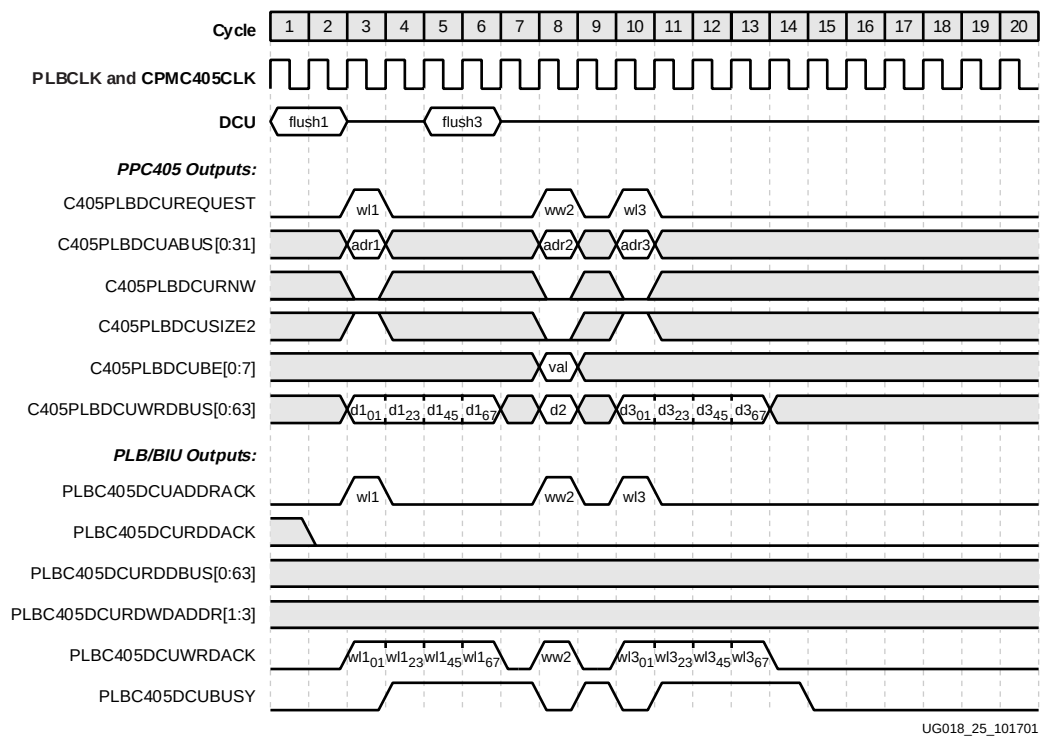
## DSPLB Line Write/Word Write/Line Write

The timing diagram in [Figure 2-21](#) shows a sequence involving an eight-word line write, a word write, and another an eight-word line write. Consecutive writes cannot be address pipelined between the DCU and BIU. The line writes are cacheable. The word writes could be in response to non-cacheable stores, cacheable stores to write-through memory, or cacheable stores that do not allocate a cache line.

The first line write (wl1) is requested by the DCU in cycle 3 in response to a cache flush (represented by the flush1 transaction in cycles 1 through 2). The BIU responds in the same cycle the request is made by the DCU. Data is sent from the DCU to the BIU in cycles 3 through 6.

The word write (ww2) cannot be started until the first request is complete. This request is made by the DCU in cycle 8 and the BIU responds in the same cycle. A single word is sent from the DCU to the BIU in cycle 8. The BIU uses the byte enables to select the appropriate bytes from the write-data bus.

The DCU queues the second flush request, flush3. The second line write (wl3) cannot be started until the second request (ww2) is complete. This request is made by the DCU in cycle 10 in response to the flush3 request. The BIU responds in the same cycle the request is made by the DCU. Data is sent from the DCU to the BIU in cycles 10 through 13.



UG018\_25\_101701

Figure 2-21: DSPLB Line Write/Word Write/Line Write

## DSPLB Three Consecutive Word Writes

The timing diagram in [Figure 2-22](#) shows three consecutive word writes. It provides an example of the fastest speed at which the DCU can request and send single words over the PLB. The word writes could be in response to non-cacheable stores, cacheable stores to write-through memory, or cacheable stores that do not allocate a cache line. Consecutive writes cannot be address pipelined between the DCU and BIU.

The first word write (ww1) is requested by the DCU in cycle 2. The BIU responds in the same cycle the request is made by the DCU. A single word is sent from the DCU to the BIU in cycle 2. The BIU uses the byte enables to select the appropriate bytes from the write-data bus.

The second word write (ww2) is requested after the first write is complete. The DCU makes the request in cycle 4 and the BIU responds in the same cycle. A single word is sent from the DCU to the BIU in cycle 4. The BIU uses the byte enables to select the appropriate bytes from the write-data bus.

The third word write (ww3) is requested after the second write is complete. The DCU makes the request in cycle 6 and the BIU responds in the same cycle. A single word is sent from the DCU to the BIU in cycle 6. The BIU uses the byte enables to select the appropriate bytes from the write-data bus.

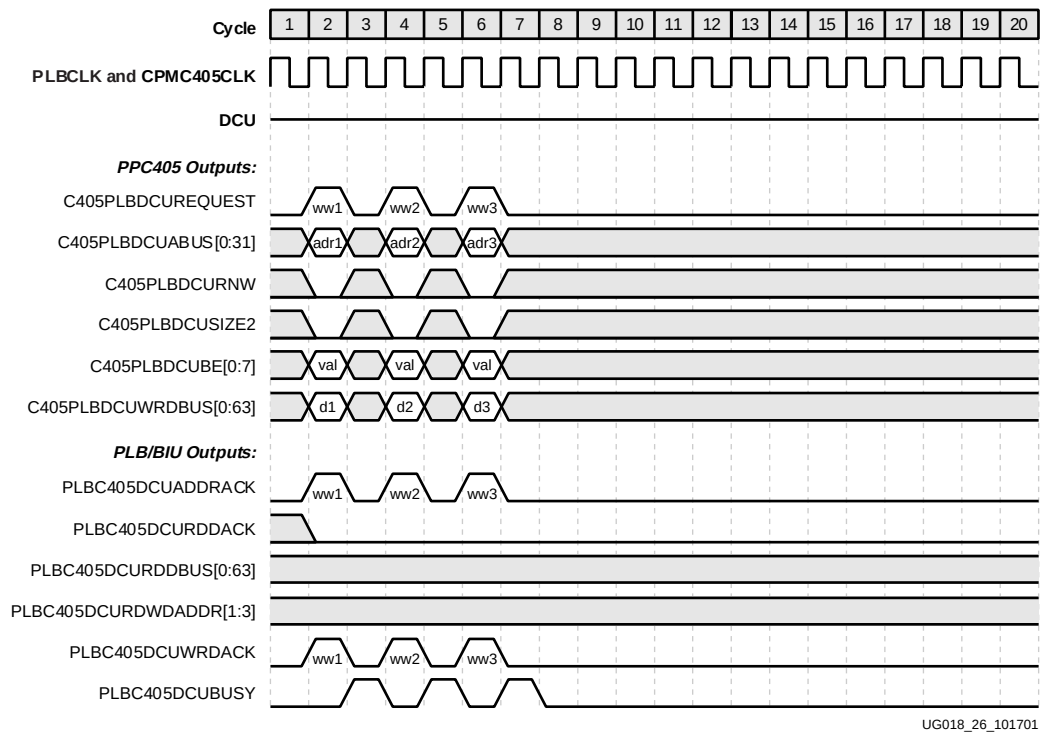


Figure 2-22: DSPLB Three Consecutive Word Writes

## DSPLB Line Write/Line Read/Word Write

The timing diagram in **Figure 2-23** shows a sequence involving an eight-word line write, an eight-word line read, and a word write. It provides an example of address pipelining involving writes and reads. It also demonstrates how read and write operations can overlap due to the split read-data and write-data busses.

The first line write (wl1) is requested by the DCU in cycle 3 in response to a cache flush (represented by the flush1 transaction in cycles 1 through 2). The BIU responds in the same cycle the request is made by the DCU. Data is sent from the DCU to the BIU in cycles 3 through 6.

The first line read (rl2) is address pipelined with the previous line write. The rl2 request is made by the DCU in cycle 5 and the BIU responds in the same cycle. Data is sent from the BIU to the DCU fill buffer in cycles 6 through 9. Because of the split data bus, a read operation overlaps with a previous write operation in cycle 6. After all data associated with this line is read, it is transferred by the DCU from the fill buffer to the data cache. This is represented by the fill2 transaction in cycles 10 through 12.

The word write (ww3) cannot be requested until the first write request (wl1) is complete because address pipelining of multiple write requests is not supported. However, this request is address pipelined with the previous line read request (rl2). The ww3 request is made by the DCU in cycle 8 and the BIU responds in the same cycle. A single word is sent from the DCU to the BIU in cycle 8. The BIU uses the byte enables to select the appropriate bytes from the write-data bus. Because of the split data bus, this write operation overlaps with a read operation from the previous read request (rl2).

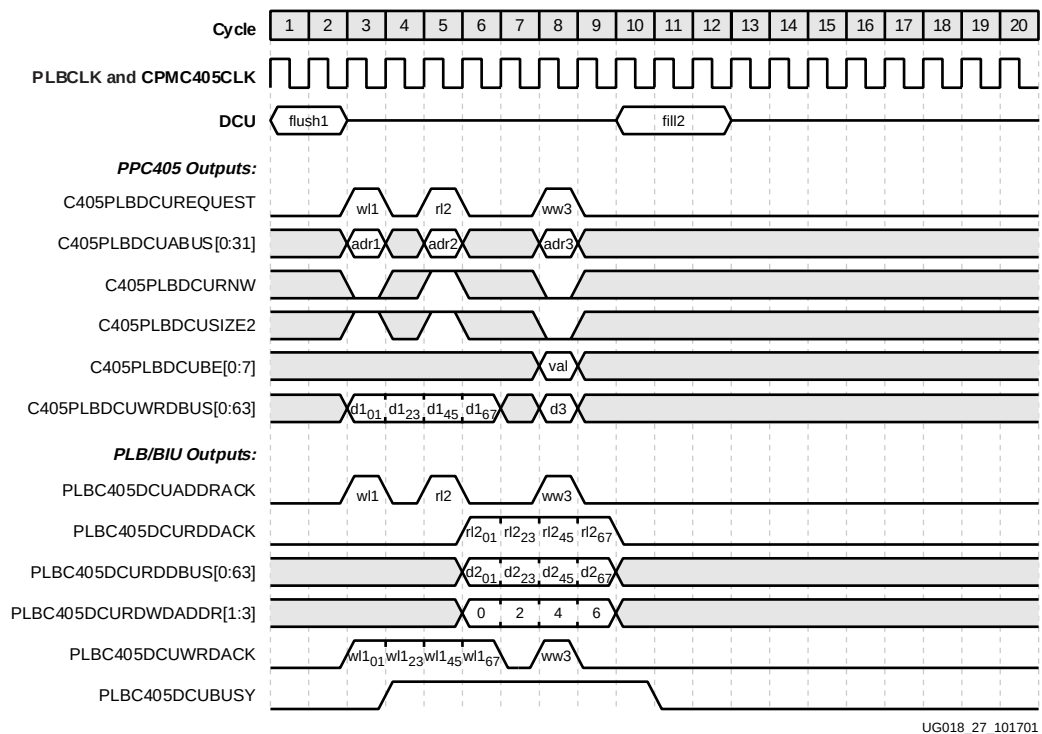


Figure 2-23: DSPLB Line Write/Line Read/Word Write

## DSPLB Word Write/Word Read/Word Write/Line Read

The timing diagram in [Figure 2-24](#) shows a sequence involving a word write, a word read, another word write, and an eight-word line read.

The first word write (ww1) is requested by the DCU in cycle 2 and the BIU responds in the same cycle. A single word is sent from the DCU to the BIU in cycle 2. The BIU uses the byte enables to select the appropriate bytes from the write-data bus.

The first word read (rw2) is requested by the DCU in cycle 4. Even though the previous request is completed in cycle 2, this is the earliest an address pipelined request can be started by the DCU. The BIU responds in the same cycle the rw2 request is made by the DCU. A single word is sent from the BIU to the DCU in cycle 5. The DCU uses the byte enables to select the appropriate bytes from the write-data bus.

The second word write (ww3) is requested by the DCU in cycle 6. Again, this is the earliest an address pipelined request can be started by the DCU. The BIU responds in the same cycle the ww3 request is made by the DCU. A single word is sent from the DCU to the BIU in cycle 6. The BIU uses the byte enables to select the appropriate bytes from the write-data bus.

The line read (rl4) is address pipelined with the word write. The rl4 request is made by the DCU in cycle 8 and the BIU responds in the same cycle. Data is sent from the BIU to the DCU fill buffer in cycles 9 through 12. After all data associated with this line is read, it is transferred by the DCU from the fill buffer to the data cache. This is represented by the fill4 transaction in cycles 13 through 15.

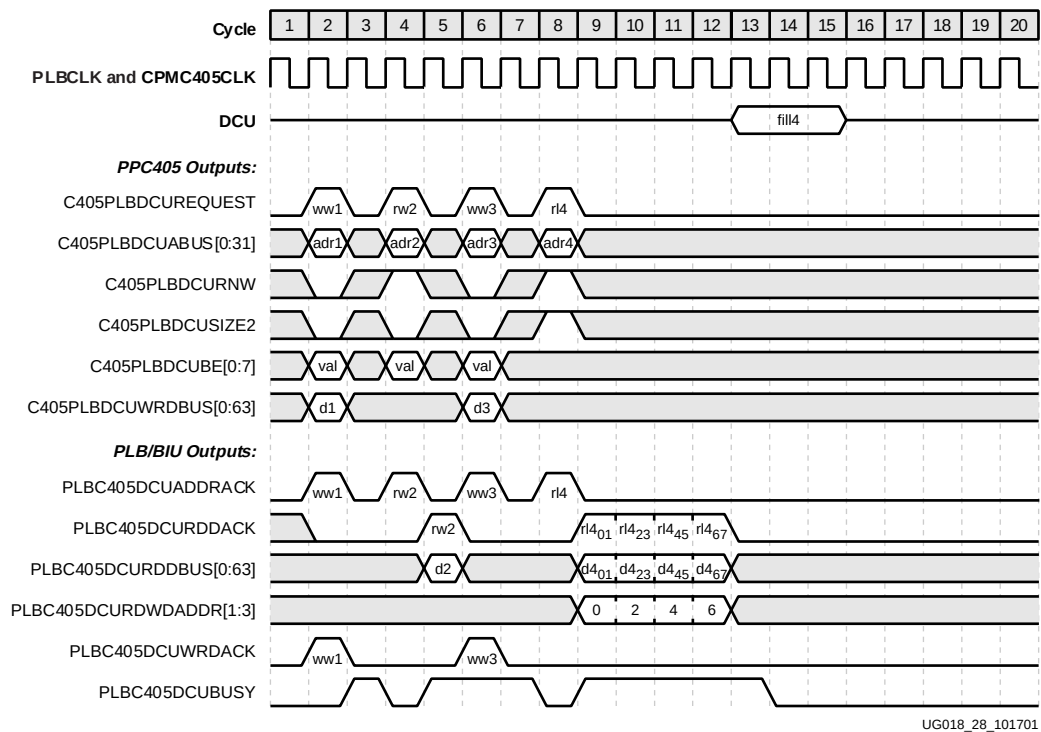


Figure 2-24: DSPLB Word Write/Word Read/Word Write/Line Read

## DSPLB Word Write/Line Read/Line Write

The timing diagram in **Figure 2-25** shows a sequence involving a word write, an eight-word line read, and an eight-word line write. It demonstrates how read and write operations can overlap due to the split read-data and write-data busses.

The word write (ww1) is requested by the DCU in cycle 2 and the BIU responds in the same cycle. A single word is sent from the DCU to the BIU in cycle 2. The BIU uses the byte enables to select the appropriate bytes from the write-data bus.

The line read (rl2) is address pipelined with the previous word write. The rl2 request is made by the DCU in cycle 4 and the BIU responds in the same cycle. Data is sent from the BIU to the DCU fill buffer in cycles 5 through 8. After all data associated with this line is read, it is transferred by the DCU from the fill buffer to the data cache. This is represented by the fill2 transaction in cycles 9 through 11.

The line write (wl3) is address pipelined with the previous line read. The wl3 request is made by the DCU in cycle 6 in response to the cache flush in cycles 4 through 5 (flush3). The BIU responds to the wl3 request in the same cycle it is asserted by the DCU. Data is sent from the DCU to the BIU in cycles 6 through 9. Because of the split data bus, the write operations in cycles 6 through 8 overlap read operations from the previous read request (rl2).

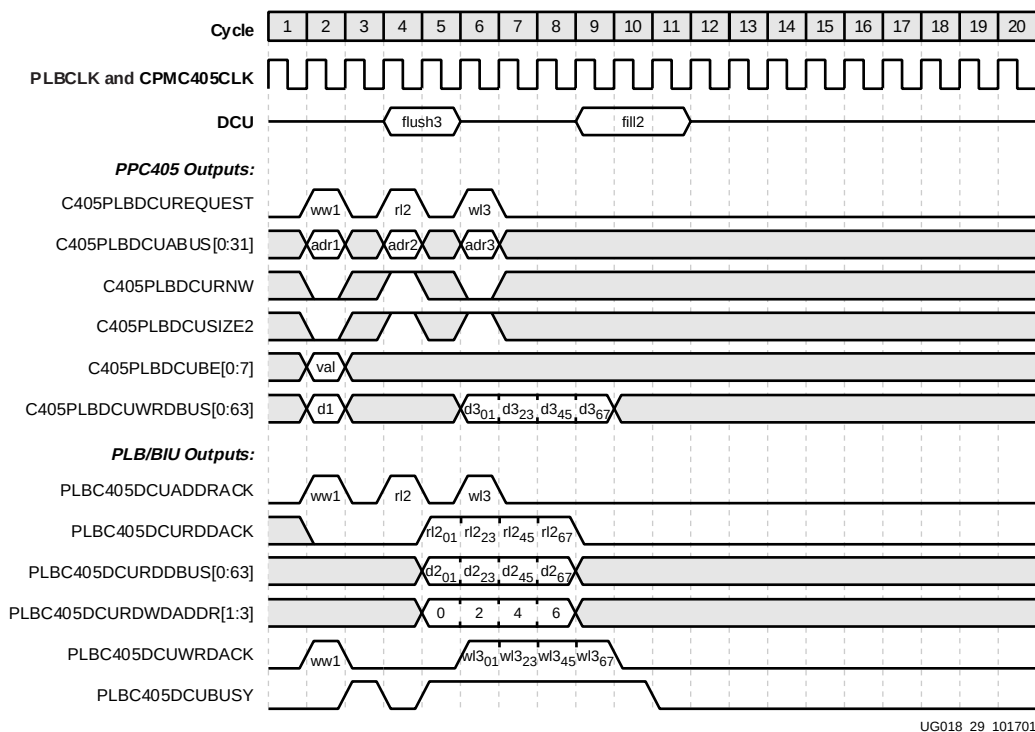


Figure 2-25: DSPLB Word Write/Line Read/Line Write

# DSPLB 2:1 Core-to-PLB Line Read

The timing diagram in [Figure 2-26](#) shows a line read in a system with a PLB clock that runs at one half the frequency of the PPC405 clock.

The line read (rl1) is requested by the DCU in PLB cycle 2, which corresponds to PPC405 cycle 3. The BIU responds in the same cycle. Data is sent from the BIU to the DCU fill buffer in PLB cycles 3 through 6 (PPC405 cycles 5 through 12). After all data associated with this line is read, it is transferred by the DCU from the fill buffer to the data cache. This is represented by the fill1 transaction in PPC405 cycles 13 through 15.

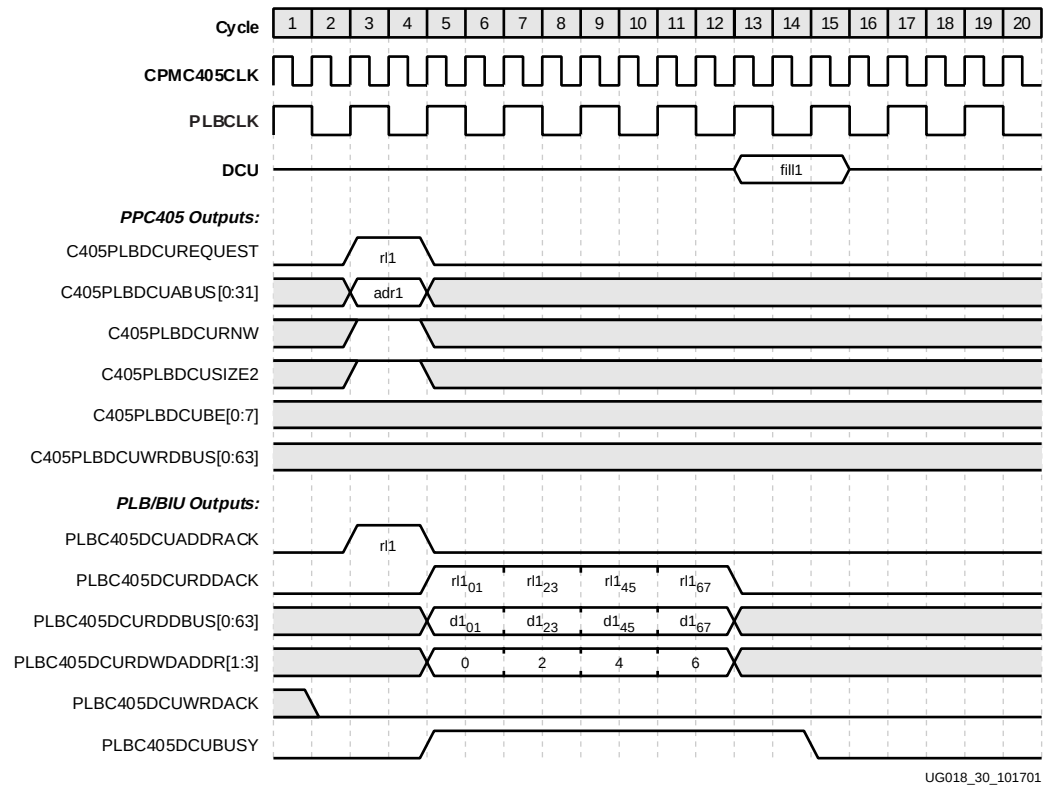


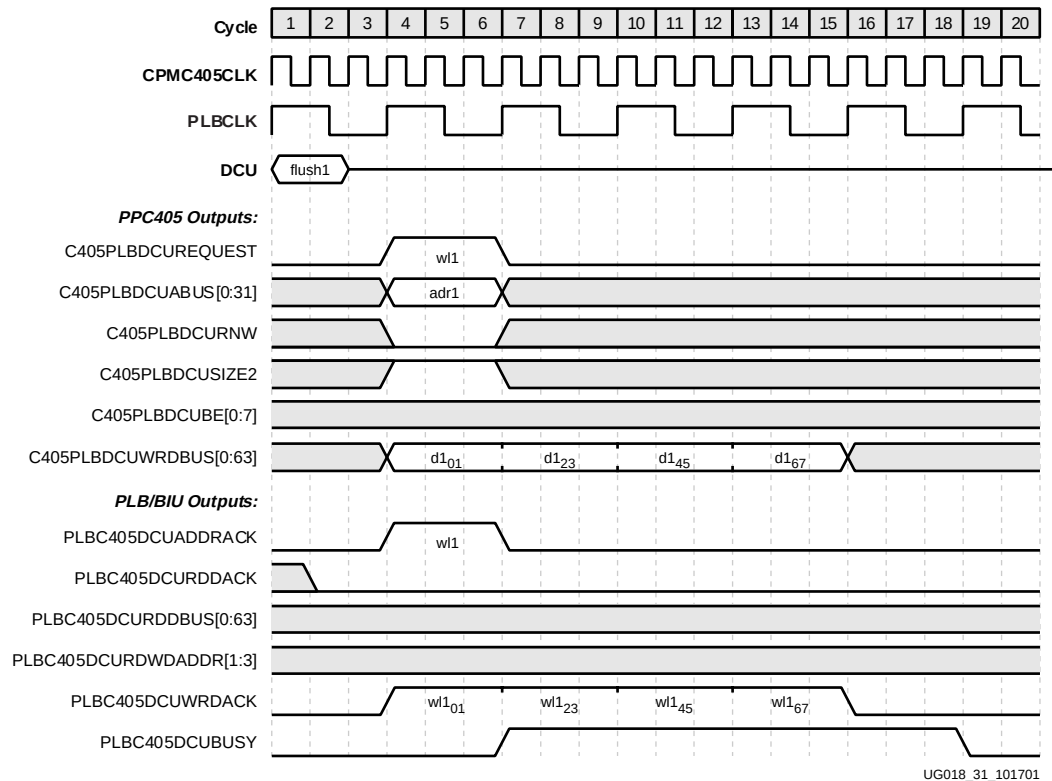
Figure 2-26: DSPLB 2:1 Core-to-PLB Line Read



## DSPLB 3:1 Core-to-PLB Line Write

The timing diagram in **Figure 2-27** shows a line write in a system with a PLB clock that runs at one third the frequency of the PPC405 clock.

The line write (wl1) is requested by the DCU in PLB cycle 2, which corresponds to PPC405 cycle 4. The BIU responds in the same cycle. The request is made in response to a flush in PPC405 cycles 1 and 2 (flush1). Data is sent from the DCU to the BIU in PLB cycles 2 through 5 (PPC405 cycles 4 through 15).



UG018\_31\_101701

Figure 2-27: DSPLB 3:1 Core-to-PLB Line Write

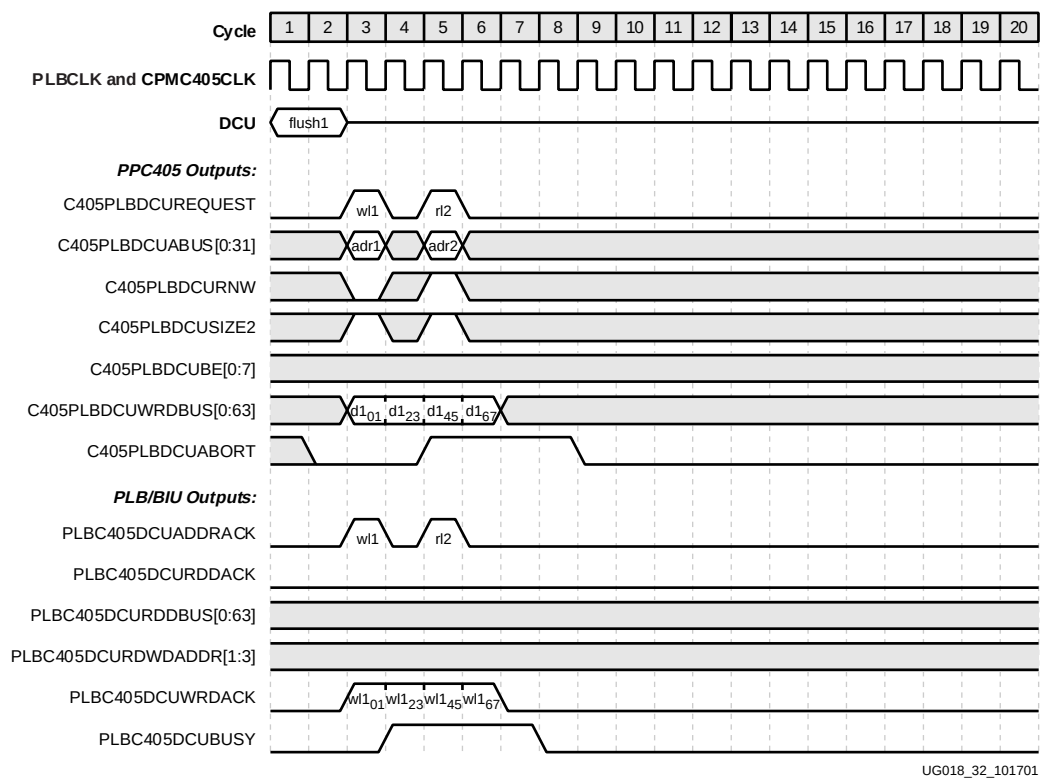
## DSPLB Aborted Data-Access Request

The timing diagram in **Figure 2-28** shows an aborted data-access request. The request is aborted because of a core reset. The BIU is not reset.

A line write (wl1) is requested by the DCU in cycle 3 in response to a cache flush (represented by the flush1 transaction in cycles 1 through 2). The BIU responds in the same cycle the request is made by the DCU. Data is sent from the DCU to the BIU in cycles 3 through 6.

A line read (rl2) is address pipelined with the previous line write. The rl2 request is made by the DCU in cycle 5 and the BIU responds in the same cycle. However, the processor also aborts the request in cycle 5. Therefore, no data is transferred from the BIU to the DCU in response to this request.

Because the BIU is not reset, it must complete the first line write even though the processor asserts the PLB abort signal during the line write.



UG018\_32\_101701

Figure 2-28: DSPLB Aborted Data-Access Request

## Device-Control Register Interface

The device-control register (DCR) interface provides a mechanism for the processor block to initialize and control peripheral devices that reside on the same FPGA chip. For example, the memory-transfer characteristics and address assignments for a bus-interface unit (BIU) can be configured by software using DCRs. The DCRs are accessed using the PowerPC **mfdcr** and **mtdcr** instructions.

The DCR interface consists of the following:

- A 10-bit address bus.
- Separate 32-bit input data and output data busses.
- Separate read and write control signals.
- A read/write acknowledgement signal.

Because the processor block is the only bus master on the bus, the address bus is driven by the processor block and received by each peripheral containing DCRs. The read and write control signals are also distributed to each DCR peripheral.

The preferred implementation of the DCR data bus is as a distributed, multiplexed chain. Each peripheral in the chain has a DCR input-data bus connected to the DCR output-data bus of the previous peripheral in the chain (the first peripheral is attached to the processor block). Each peripheral multiplexes this bus with the outputs of its DCRs and passes the resulting DCR bus as an output to the next peripheral in the chain. The last peripheral in the chain has its DCR output-data bus attached to the processor block DCR input-data interface. This implementation enables future DCR expansion without requiring changes to I/O devices due to additional loading.

There are two options for connecting the acknowledge signals. The acknowledge signals from the DCRs can be latched and forwarded in the chain with the DCR data bus. Alternatively, combinatorial logic, such as OR gates, can be used to combine and forward the acknowledge signal to the processor block.

Figure 2-29, page 995 shows an example DCR chain implementation in an FPGA chip. The acknowledge signal in this example is formed using combinatorial logic (OR gate).

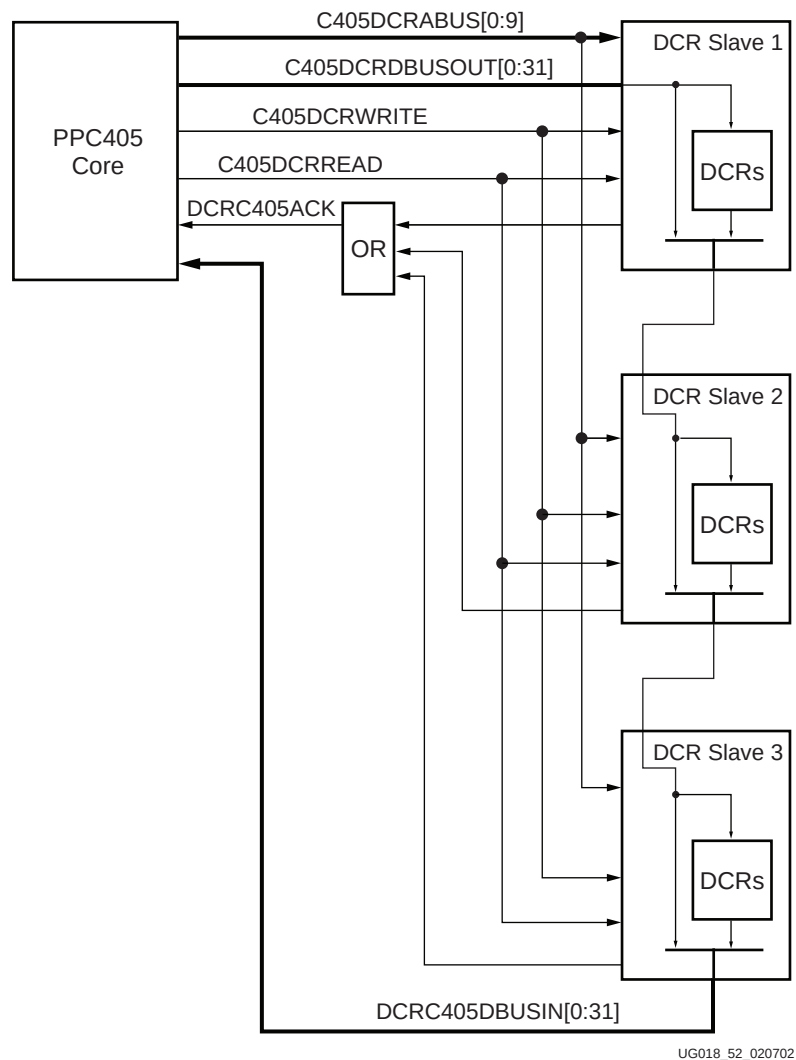


Figure 2-29: DCR Chain Block Diagram

The acknowledge signal is interlocked with the read and write control signals. The interlock mechanism enables the DCR interface to communicate with peripheral devices that are clocked at different frequencies than the PPC405. A DCR access takes at least three PPC405 cycles. If a DCR access is not acknowledged in 64 processor cycles, the access times out. No error occurs when a DCR access times-out. Instead, the processor begins executing the next-sequential instruction.

The interlock mechanism requires that the rising edge of the slower clock (either the PPC405 clock or the peripheral clock) correspond to the rising edge of the faster clock. This means that the clocks for the DCR logic and the clocks for the PPC405 must be derived from a common source. The common source frequency is multiplied or divided before being sent to the PPC405 or DCR logic.

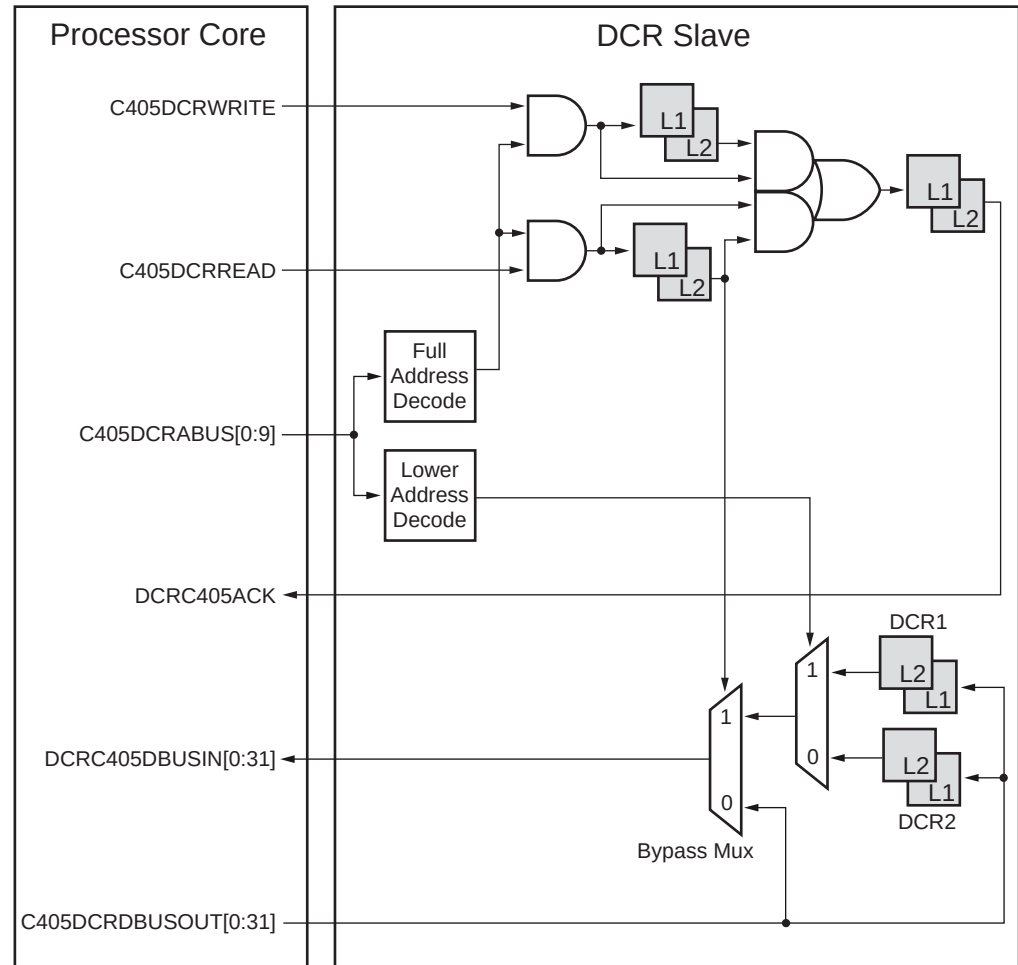
The DCR interface operates in two ways, referred to as *mode 0* and *mode 1* (these are hard wired modes, not programmable modes):

- In mode 0, the PPC405 and FPGA can be clocked at different frequencies without affecting the interface handshaking protocol. In this mode, an acknowledgement follows a read or write operation. The acknowledgement cannot be deasserted until

the read or write signal is deasserted.

- In mode 1, the core and FPGA must run at the same frequency. In this mode, an acknowledgement follows a read or write operation. However, the acknowledgement can be deasserted one cycle after it is asserted. This enables the fastest back-to-back DCR access (three cycles).

Figure 2-30 illustrates a logical implementation of the DCR bus interface. This implementation enables a DCR slave to run at a different clock speed than the PPC405. The acknowledge signal is latched and forwarded with the DCR bus. The bypass multiplexor minimizes data-bus path delays when the DCR is not selected. To ensure reusability across multiple FPGA environments, all DCR slave logic should use the specified implementation.



UG018\_53\_020702

Figure 2-30: DCR Bus Implementation

## DCR Interface I/O Signal Summary

Figure 2-31 shows the block symbol for the DCR interface. The signals are summarized in Table 2-18.

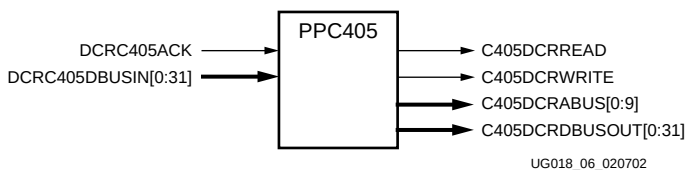


Figure 2-31: DCR Interface Block Symbol

Table 2-18: DCR Interface I/O Signals

| Signal               | I/O Type | If Unused                           | Function                                                   |
|----------------------|----------|-------------------------------------|------------------------------------------------------------|
| C405DCRREAD          | O        | No Connect                          | Indicates a DCR read request occurred.                     |
| C405DCRWRITE         | O        | No Connect                          | Indicates a DCR write request occurred.                    |
| C405DCRABUS[0:9]     | O        | No Connect                          | Specifies the address of the DCR access request.           |
| C405DCRDBUSOUT[0:31] | O        | No Connect or attach to input bus   | The 32-bit DCR write-data bus.                             |
| DCRC405ACK           | I        | 0                                   | Indicates a DCR access has been completed by a peripheral. |
| DCRC405DBUSIN[0:31]  | I        | 0x0000_0000 or attach to output bus | The 32-bit DCR read-data bus.                              |

## DCR Interface I/O Signal Descriptions

The following sections describe the operation of the DCR interface I/O signals.

### C405DCRREAD (output)

When asserted, this signal indicates the processor block is requesting the contents of a DCR (reading from the DCR) in response to the execution of a *move-from DCR* instruction (**mf dcr**). The contents of the DCR address bus are valid when this request is asserted (the request is asserted one processor cycle after the processor block begins driving the DCR address bus). This signal is deasserted two processor cycles after the DCR acknowledge signal is asserted. When deasserted, no DCR read request exists. DCRs must not drive the DCR bus if this signal is not asserted.

The processor block waits up to 64 cycles for a read request to be acknowledged. If a DCR does not acknowledge the request in this time, the read times out. No error occurs when a DCR read times-out. Instead, the processor begins executing the next-sequential instruction.

DCR read requests are not interrupted by the processor block. If this signal is asserted, only a DCR acknowledgement or read time-out cause the signal to be deasserted.

This signal is deasserted during reset.

### C405DCRWRITE (output)

When asserted, this signal indicates the processor block is requesting that the contents of a DCR be updated (writing to the DCR) in response to the execution of a *move-to DCR* instruction (**mt dcr**). The contents of the DCR address bus are valid when this request is asserted (the request is asserted one processor cycle after the processor block begins driving the DCR address bus). This signal is deasserted two processor cycles after the DCR acknowledge signal is asserted. When deasserted, no DCR write request exists.

The processor block waits up to 64 cycles for a write request to be acknowledged. If a DCR does not acknowledge the request in this time, the write times out. No error occurs when a DCR write times-out. Instead, the processor begins executing the next-sequential instruction.

DCR write requests are not interrupted by the processor block. If this signal is asserted, only a DCR acknowledgement or write time-out cause the signal to be deasserted.

This signal is deasserted during reset.

### C405DCRABUS[0:9] (output)

This bus specifies the address of the DCR access request. This bus remains stable during the execution of a **mfdcr** or **mtddcr** instruction. However, the contents of this bus are valid only when either a DCR read request or DCR write request are asserted by the processor. The processor does not begin driving a new DCR address until the DCR acknowledge signal corresponding to the previous DCR access has been deasserted for at least one cycle.

The address driven by this bus corresponds to the DCR number (DCRN) and not the split DCR field (DCRF) encoded in the **mfdcr** or **mtddcr** instruction. For example, if the DCRN is 0x2AA, the DCR address bus is driven with the value 0x2AA. However, the DCRF encoded by the DCR instruction is 0x155. See the *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*) for more information on these instructions.

### C405DCRDBUSOUT[0:31] (output)

This write-data bus is driven by the processor block when a **mtddcr** or **mfdcr** instruction is executed. Its contents are valid only when a DCR write-request or DCR read-request is asserted. When a **mtddcr** instruction is executed, this bus contains the data to be written into a DCR. When a **mfdcr** instruction is executed, this bus contains the value 0x0000\_0000.

During reset, this bus is driven with the value 0x0000\_0000. Peripherals can use this value to initialize the DCRs.

### DCRC405ACK (input)

When asserted, this signal indicates a peripheral device acknowledges the processor block request for DCR access. For a read-access request, the peripheral device should assert this signal when the DCR read-data bus is driven with the appropriate DCR contents (the bus contains valid data). For a write-access request, the peripheral device should assert this signal when the DCR write-data bus is latched into the appropriate DCR. Peripheral devices should assert this signal only when all of the following are true:

- They contain the accessed DCR.
- A valid read-access or write-access DCR request exists.
- The peripheral device has driven the DCR bus (read access) or latched the DCR bus (write access).

Deasserting the acknowledge signal after it has been asserted depends on the DCR interface mode (these are hard wired modes, not programmable modes):

- In mode 0, the acknowledgement cannot be deasserted until the read or write signal is deasserted. This enables the PPC405 and FPGA to be clocked at different frequencies without affecting the interface handshaking protocol.
- In mode 1, the acknowledge signal should be deasserted in the cycle after it is asserted (the peripheral device and the PPC405 are clocked at the same frequency). It is not necessary to wait for the read-access or write-access signal to be deasserted. This enables the fastest back-to-back DCR access (three cycles).

### DCRC405DBUSIN[0:31] (input)

This read-data bus is latched (read) by the processor block when a peripheral device asserts the DCR acknowledge signal in response to a DCR read-access request. A

peripheral device must drive this bus only when it contains the accessed DCR and the DCR read-access signal is asserted by the processor block.

Peripheral devices should drive only the bits implemented by the specified DCR. A value of 0x0000\_0000 is driven onto the DCR write-data bus by the processor block during a read-access request. This value is passed along the DCR chain until modified by the appropriate peripheral. The end of the DCR chain is attached to the DCR read-data bus input to the processor block. Thus, the processor reads the updated value of all implemented bits, and unimplemented (and unattached) bits retain a value of 0.

# DCR Interface Timing Diagrams

The following timing diagrams show typical transfers that can occur on the DCR interface using the two interface modes. Unless otherwise noted, optimal timing relationships are used to improve the readability of the timing diagrams. The assertion of C405DCRREAD/C405DCRWRITE refers to a read or write operation, not both. The processor block cannot perform a simultaneous read and write of the DCR bus.

## DCR Interface Mode 0, 1:1 Clocking, Latched Acknowledge

The example in Figure 2-32 assumes the following:

- The PPC405 and the peripheral containing the DCR are clocked at the same frequency.
- The acknowledge signal is latched and forwarded with the DCR bus as shown in Figure 2-30, page 996.
- After the acknowledge signal is asserted, it is not deasserted until the appropriate read-access or write-access request signal is deasserted (mode 0 interface operation).

Using these assumptions, the fastest back-to-back DCR access occurs every seven cycles. The implementation of the acknowledge signal causes it to be asserted two cycles after the access request. It is deasserted in the cycle after the access request is deasserted.

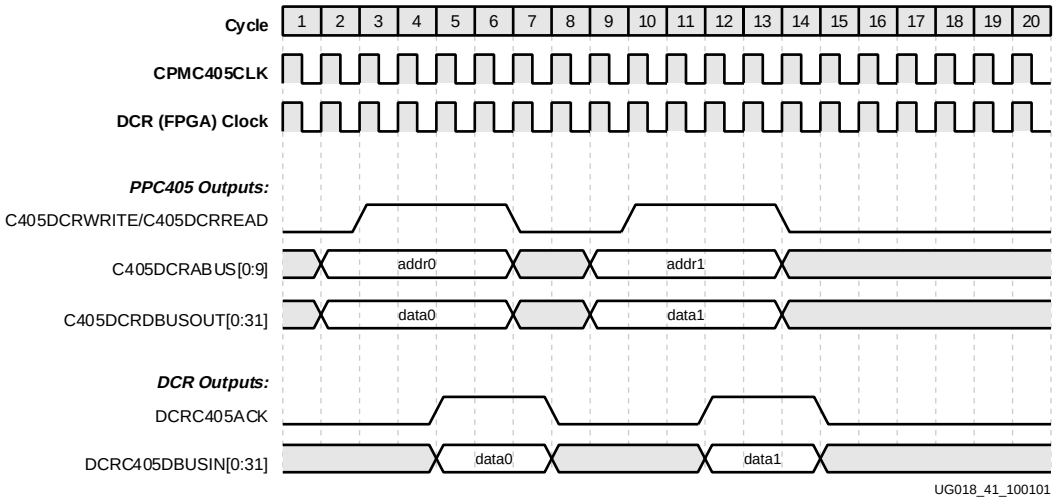


Figure 2-32: DCR Interface Mode 0, 1:1 Clocking, Latched Acknowledge



## DCR Interface Mode 0, 1:1 Clocking, Combinatorial Acknowledge

The example in Figure 2-33 assumes the following:

- The PPC405 and the peripheral containing the DCR are clocked at the same frequency.
- The acknowledge signal is generated by combinatorial logic using the acknowledge signal from each peripheral, as shown in Figure 2-29, page 995.
- After the acknowledge signal is asserted, it is not deasserted until the appropriate read-access or write-access request signal is deasserted (mode 0 interface operation).

Using these assumptions, the fastest back-to-back DCR access occurs every four cycles. The implementation of the acknowledge signal causes it to be asserted in the same cycle as the access request. It is assumed that the selected DCR can latch/drive the DCR bus in the same cycle. The combinatorial logic enables the acknowledge signal to be deasserted in the same cycle that the access request is deasserted.

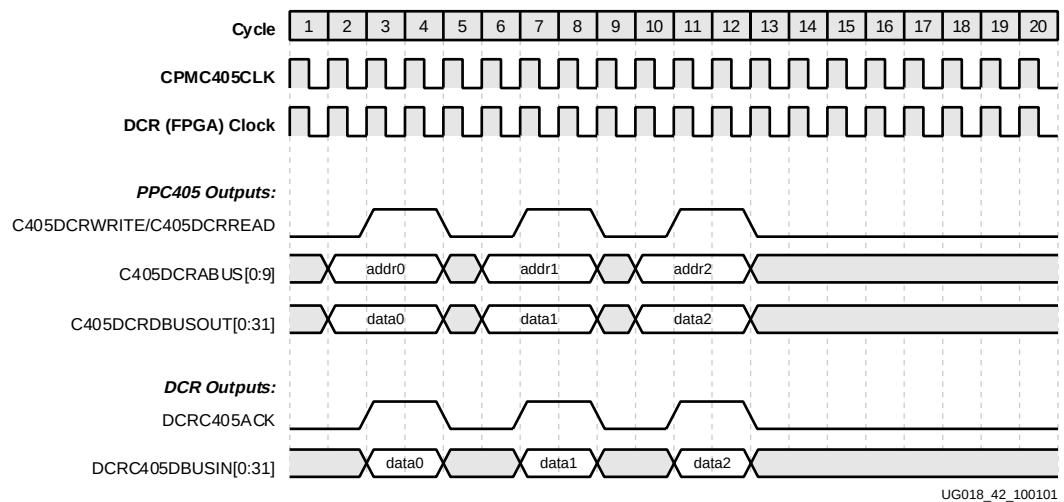


Figure 2-33: DCR Interface Mode 0, 1:1 Clocking, Combinatorial Acknowledge

# DCR Interface Mode 0, 2:1 Clocking, Latched Acknowledge

The example in Figure 2-34 assumes the following:

- The PPC405 is clocked at twice the frequency of the peripheral containing the DCR.
- The acknowledge signal is latched and forwarded with the DCR bus as shown in Figure 2-30, page 996.
- After the acknowledge signal is asserted, it is not deasserted until the appropriate read-access or write-access request signal is deasserted (mode 0 interface operation).

Using these assumptions, the fastest back-to-back DCR access occurs every ten PPC405 cycles. The implementation of the acknowledge signal causes it to be asserted two DCR cycles (four PPC405 cycles) after the access request. It is deasserted in the DCR cycle after the access request is deasserted.

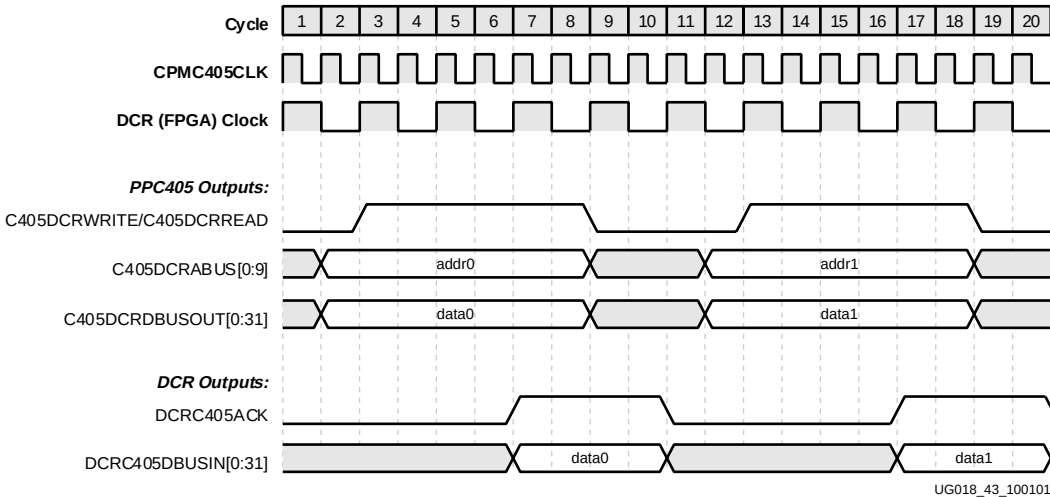


Figure 2-34: DCR Interface Mode 0, 2:1 Clocking, Latched Acknowledge

## DCR Interface Mode 0, 1:2 Clocking, Latched Acknowledge

The example in [Figure 2-35](#) assumes the following:

- The PPC405 is clocked at half the frequency of the peripheral containing the DCR.
- The acknowledge signal is latched and forwarded with the DCR bus as shown in [Figure 2-30, page 996](#).
- After the acknowledge signal is asserted, it is not deasserted until the appropriate read-access or write-access request signal is deasserted (mode 0 interface operation).

Using these assumptions, the fastest back-to-back DCR access occurs every six PPC405 (twelve DCR) cycles. The implementation of the acknowledge signal causes it to be asserted two DCR cycles (one PPC405 cycles) after the access request. It is deasserted in the DCR cycle after the access request is deasserted.

The processor block does not present a new DCR address until at least one cycle after the acknowledge is deasserted. In this example, the PPC405 and DCR clocks are not interlocked (the rising edges do not coincide) when the acknowledge is deasserted. The clocks interlock one-half a PPC405 cycle later, but the processor block must wait an entire additional cycle before it can present a new DCR address.

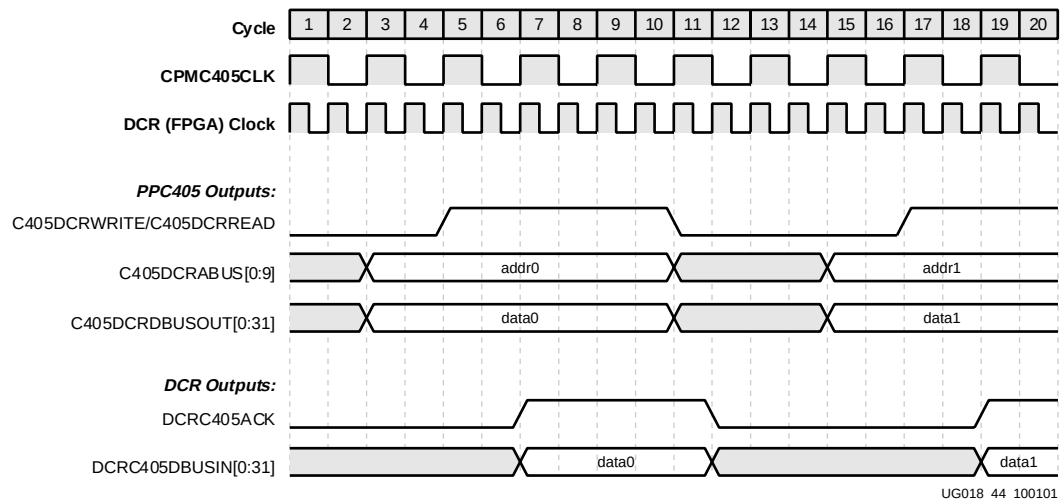


Figure 2-35: DCR Interface Mode 0, 1:2 Clocking, Latched Acknowledge

# DCR Interface Mode 1, 1:1 Clocking, Combinatorial Acknowledge

The example in [Figure 2-36](#) assumes the following:

- The PPC405 and the peripheral containing the DCR are clocked at the same frequency.
- The acknowledge signal is generated by combinatorial logic using the acknowledge signal from each peripheral, as shown in [Figure 2-29, page 995](#).
- The acknowledge signal is deasserted in the cycle after it is asserted (mode 1 interface operation).

Using these assumptions, the fastest back-to-back DCR access occurs every three cycles. This represents the fastest speed at which the processor can present successive DCR requests on the DCR interface.

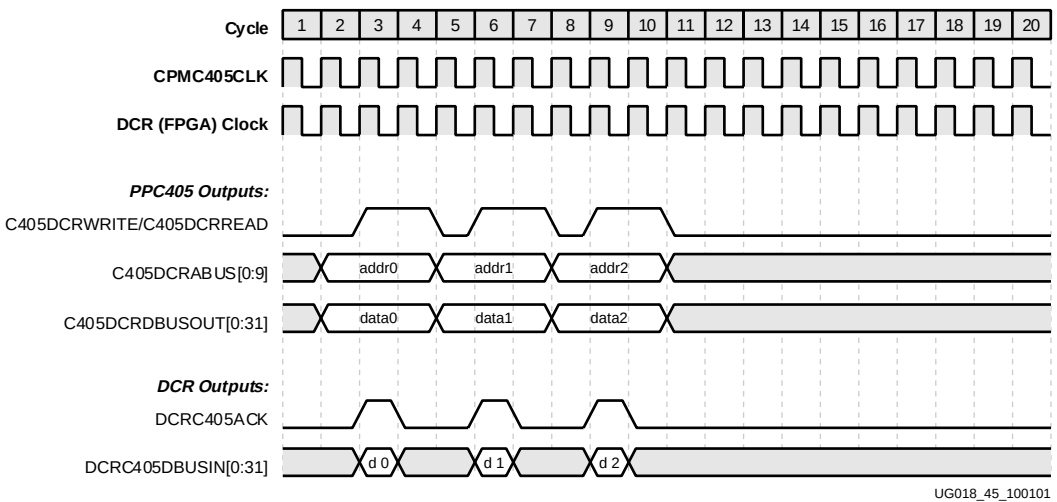


Figure 2-36: DCR Interface Mode 1, 1:1 Clocking, Combinatorial Acknowledge

## External Interrupt Controller Interface

The PowerPC embedded-environment architecture defines two classes of interrupts: critical and noncritical. The interrupt handler for an external critical interrupt is located at exception-vector offset 0x0100. The interrupt handler for an external noncritical interrupt is located at exception-vector offset 0x0200. Generally, the processor prioritizes critical interrupts ahead of noncritical interrupts when they occur simultaneously (certain debug exceptions are handled at a lower priority). Critical interrupts use a different save/restore register pair (SRR2 and SRR3) than is used by noncritical interrupts (SRR0 and SRR1). This enables a critical interrupt to interrupt a noncritical-interrupt handler. The state saved by the noncritical interrupt is not overwritten by the critical interrupt. See the *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*) for more information on exception and interrupt processing.

Logic external to the processor block can be used to cause critical and noncritical interrupts. External interrupt sources are collected by the external interrupt controller (EIC) and presented to the processor block as either a critical or noncritical interrupt. Once an external interrupt request is asserted, the EIC must keep the signal asserted until software deasserts it. This is typically done by writing to a DCR in the EIC peripheral logic.

Software can enable and disable external interrupts using the following bits in the machine-state register MSR:

- Noncritical interrupts are controlled by MSR[EE]. When set to 1, noncritical interrupts are enabled. When cleared to 0, they are disabled.
- Critical interrupts are controlled by MSR[CE]. When set to 1, critical interrupts are enabled. When cleared to 0, they are disabled.

The states of the EE and CE bits are reflected by output signals on the processor block CPM interface. See **Clock and Power Management Interface**, page 933, for more information.

An external interrupt is considered pending if it occurs while the corresponding class is disabled. The EIC continues to assert the interrupt request. When software later enables the interrupt class, the interrupt occurs and the interrupt handler deasserts the request by writing to a DCR in the EIC.

### EIC Interface I/O Signal Summary

Figure 2-37 shows the block symbol for the EIC interface. The signals are summarized in Table 2-19.

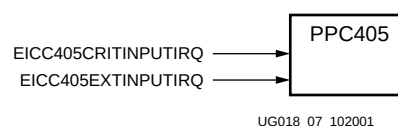


Figure 2-37: EIC Interface Block Symbol

Table 2-19: EIC Interface I/O Signals

| Signal              | I/O Type | If Unused | Function                                              |
|---------------------|----------|-----------|-------------------------------------------------------|
| EICC405CRITINPUTIRQ | I        | 0         | Indicates an external critical interrupt occurred.    |
| EICC405EXTINPUTIRQ  | I        | 0         | Indicates an external noncritical interrupt occurred. |

### EIC Interface I/O Signal Descriptions

The following sections describe the operation of the EIC interface I/O signals.

### EICC405CRITINPUTIRQ (input)

When asserted, this signal indicates the EIC is requesting that the processor block respond to an external critical interrupt. When deasserted, no request exists. The EIC is responsible for collecting critical interrupt requests from other peripherals and presenting them as a single request to the processor block. Once asserted, this signal remains asserted by the EIC until software deasserts the request (this is typically done by writing to a DCR in the EIC).

### EICC405EXTINPUTIRQ (input)

When asserted, this signal indicates the EIC is requesting that the processor block respond to an external noncritical interrupt. When deasserted, no request exists. The EIC is responsible for collecting noncritical interrupt requests from other peripherals and presenting them as a single request to the processor block. Once asserted, this signal remains asserted by the EIC until software deasserts the request (this is typically done by writing to a DCR in the EIC).

## JTAG Interface

The JTAG (Joint Test Action Group) interface provides access to the processor block JTAG controller. This controller supports the JTAG test access port, boundary-scan capabilities, and user-specific instructions. The controller also provides the ability for an external debug tool (RISCWatch, for example) to control the processor for debugging purposes.

The JTAG controller contains the following logical structures:

- The *JTAG test-access port* (TAP) is used to load and unload JTAG instructions and data. The standard TAP has four signals that control the circuit blocks and their operation. The four TAP signals are:
  - TCK—This is the JTAG *test clock*. It controls the sequencing of the TAP controller and when data is moved into and out of the JTAG registers. Only one JTAG register (instruction or data) is loaded from TDI and output to TDO for any JTAG operation.
  - TMS—This is the JTAG *test-mode select* signal. It determines the next state of the TAP controller. TMS causes a state transition in the TAP controller on the rising edge of TCK.
  - TDI—This is the JTAG *test-data in* signal. It is a serial data input to all JTAG instruction and data registers. The current state of the TAP controller and the contents of the JTAG instruction register (IR) determine which JTAG register is loaded from the TDI. TDI is loaded into a JTAG register on the rising edge of TCK.
  - TDO—This is the JTAG *test-data out* signal. It is a serial data output from all JTAG instruction and data registers. The current state of the TAP controller and the contents of the JTAG instruction register determine which JTAG register has its contents unloaded onto the TDO. The state of TDO changes on the falling edge of TCK.
- The *TAP controller* is a 16-state finite-state machine (FSM) that controls the loading of data from the TDI input signal into the various JTAG registers. It responds to control sequences supplied through the TAP. It also generates the clocks and control signals required by the other circuit blocks. There are two basic paths through the TAP state machine: one for shifting information to the instruction register and one for shifting data into the data register. The state of the TMS input signal at the rising edge of TCK determines the sequence of transitions through these paths in the state machine.
- The *instruction register* (IR) is loaded with JTAG instructions that perform certain operations. It is a shift register that is serially loaded from the TDI input. It is serially written out through the TDO output.
- The *data registers* (DR) are a collection of shift registers that are used in JTAG operations. Stimuli required by a JTAG operation are serially loaded from the TDI input. After an operation is performed, the results are serially written out through the TDO output.

With the exception of the JTAG test reset ( $\overline{\text{TRST}}$ ) signal, the processor block JTAG interface adheres to IEEE Standard 1149.1. This standard defines  $\overline{\text{TRST}}$  as an optional signal. On the processor block, however, this input signal (JTGC405TRSTNEG) must be connected so that the JTAG and debug logic are reset during a power-on reset.

FPGA implementations that include a second TAP controller for IEEE compatibility should connect the instruction registers in series, with the processor block IR closest to the processor TDI input. The second IR is three bits and the processor block IR is four bits, so the resulting IR provides a 7-bit instruction code. This implementation is compatible with that of standard products based on the PPC405 and facilitates portability of debugging tools.

The 7-bit instruction code supported by PPC405 standard products is shown in [Table 2-20](#). In this table, the “Code” column shows the instruction code in binary form. The most

significant four bits of this code are located in the PPC405 IR and the least significant three bits are in the second IR.

**Table 2-20: PPC405 Standard Product JTAG Instruction Codes**

| Instruction    | Code                                                                                         | Description                        |
|----------------|----------------------------------------------------------------------------------------------|------------------------------------|
| EXTEST         | 1111_000                                                                                     | IEEE 1149.1 standard               |
|                | 1111_001                                                                                     | Reserved                           |
| SAMPLE/PRELOAD | 1111_010                                                                                     | IEEE 1149.1 standard               |
| IDCODE         | 1111_011                                                                                     | IEEE 1149.1 standard               |
| PRIVATE        | 0101_100<br>0111_100<br>1001_100<br>1010_100<br>1011_100<br>1100_100<br>1101_100<br>1110_100 | PPC405 hardware debug instructions |
| HIGHZ          | 1111_101                                                                                     | IEEE 1149.1a-1993 optional         |
| CLAMP          | 1111_110                                                                                     | IEEE 1149.1a-1993 optional         |
| BYPASS         | 1111_111                                                                                     | IEEE 1149.1 standard               |

The DBGC405DEBUGHALT signal ([page 1012](#)) is useful when connecting RISCWatch or another external debugger to the JTAG interface. This signal provides additional debug capabilities that are required if power management that disables clock zones is implemented. The signal is optional on the RISCWatch interface.

## JTAG Interface I/O Signal Table

[Figure 2-38](#) shows the block symbol for the JTAG interface. The signals are summarized in [Table 2-21](#). See [Appendix A, RISCWatch and RISCTrace Interfaces](#) for information on attaching a RISCWatch to the JTAG interface signals.

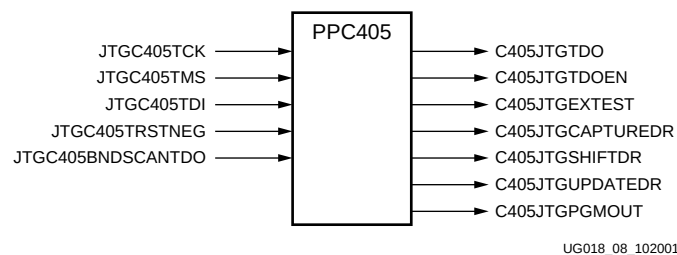


**Figure 2-38: JTAG Interface Block Symbol**



Table 2-21: JTAG Interface I/O Signals

| Signal            | I/O Type | If Unused       | Function                                                                                        |
|-------------------|----------|-----------------|-------------------------------------------------------------------------------------------------|
| JTGC405TCK        | I        | See IEEE 1149.1 | JTAG TCK (test clock).                                                                          |
| JTGC405TMS        | I        | 1               | JTAG TMS (test-mode select).                                                                    |
| JTGC405TDI        | I        | 1               | JTAG TDI (test-data in).                                                                        |
| JTGC405TRSTNEG    | I        | Required        | JTAG $\overline{\text{TRST}}$ (test reset).                                                     |
| JTGC405BNDSCANTDO | I        | 0               | JTAG boundary scan input from the previous boundary scan element TDO output.                    |
| C405JTGTD0        | O        | No Connect      | JTAG TDO (test-data out).                                                                       |
| C405JTGTD0EN      | O        | No Connect      | Indicates the JTAG TDO signal is enabled.                                                       |
| C405JTGEXTEST     | O        | No Connect      | Indicates the JTAG EXTEST instruction is selected.                                              |
| C405JTGCAPTUREDR  | O        | No Connect      | Indicates the TAP controller is in the capture-DR state.                                        |
| C405JTGSHIFTDR    | O        | No Connect      | Indicates the TAP controller is in the shift-DR state.                                          |
| C405JTGUPDATER    | O        | No Connect      | Indicates the TAP controller is in the update-DR state.                                         |
| C405JTGPGMOUT     | O        | No Connect      | Indicates the state of a general purpose program bit in the JTAG debug control register (JDCR). |

## JTAG Interface I/O Signal Descriptions

The following sections describe the operation of the JTAG interface I/O signals.

### JTGC405TCK (input)

This input is the JTAG TCK signal. This clock operates independently of the PPC405 source clock (CPMC405CLOCK) so that test operations can be synchronized between various system components. Synchronization of JTAG interface signals occurs on both the rising and falling edges of this clock. The TMS and TDI signals are latched on the rising edge of TCK. TDO is valid on the falling edge of TCK. The maximum valid TCK frequency is one-half the PPC405 source clock frequency.

### JTGC405TMS (input)

This input is the JTAG TMS signal. It is latched by the processor on the rising edge of TCK. The value of the signal is typically changed by external logic on the falling edge of TCK.

The TAP state machine uses the sequence of values applied to the TMS input to generate clock and control signals required by other test logic. For example, the TAP state machine determines the destination of data received on the TDI signal. When the TMS signal is not driven (operated) by an external source it should be held to a value of 1. This is done by attaching a pull-up resistor to the signal.

### JTGC405TDI (input)

This input is the JTAG TDI signal. It is latched by the processor on the rising edge of TCK. The value of the signal is typically changed by external logic on the falling edge of TCK.

Data received on this input signal is placed into the IR or the appropriate DR as specified by the TAP state machine. When the TDI signal is not driven (operated) by an external source it should be held to a value of 1. This is done by attaching a pull-up resistor to the signal.

### JTGC405TRSTNEG (input)

This input is the JTAG test reset ( $\overline{\text{TRST}}$ ) signal. It can be connected to the chip-level  $\overline{\text{TRST}}$  signal. Although optional in IEEE Standard 1149.1, this signal is automatically used by the processor block during power-on reset to properly reset all processor block logic, including the JTAG and debug logic. When deasserted, no JTAG test reset exists.

This is a negative active signal.

### JTGC405BNDSCANTDO (input)

This input enables the processor block to be included in the JTAG boundary scan ring. It is connected to a TDO output from another core or FPGA logic unit on the same chip, or it can be connected to the chip-level TDI input. Configuration of the TAP state machine enables data to be scanned into this input signal and out of the processor block TDO output signal. The processor block TDO output signal is connected to the boundary scan input of the next element in the ring (or the chip-level TDO output).

### C405JTGTDO (output)

This output is the JTAG test-data out (TDO) signal. It is driven by the processor with a new value on the falling edge of the JTAG clock. When data is not being shifted through the chip by the processor, this output should be placed in the high-impedance state. An enable signal (below) is provided for controlling a three-state driver that sends this signal elsewhere in the chip.

Data transmitted on this output signal comes from either the IR, a DR, or the boundary scan TDO input signal (above). The source of the data is determined by the TAP state machine.

### C405JTGTDOEN (output)

When asserted, this signal enables a three-state driver to send the TDO output signal (above) to other locations on the chip. When deasserted, the three-state driver should be placed in the high-impedance state. Such a driver is implemented on the FPGA chip external to the processor block.

### C405JTGEXTEST (output)

When asserted, this signal indicates that the JTAG external test (EXTEST) instruction is selected. This instruction enables testing of off-chip circuitry and board-level interconnections using the TDO output signal. In an FPGA implementation of a system-on-chip product, the instruction can be used to test on-chip circuitry and interconnections. When the signal is deasserted, the EXTEST instruction is not selected.

### C405JTGCAPTUREDR (output)

When asserted, this signal indicates the TAP controller is in the capture-DR state (this is one of the 16 standard TAP controller states). In this state, data is loaded from parallel inputs into the currently selected DR on the rising edge of TCK. If the register does not have parallel inputs, this state is ignored and the register retains its current value. JTAG boundary scan logic can use this signal as an indication to perform a similar function elsewhere on the chip. When deasserted, the TAP controller is not in the capture-DR state.

### C405JTGSHIFTDR (output)

When asserted, this signal indicates the TAP controller is in the shift-DR state (this is one of the 16 standard TAP controller states). In this state, data in the currently selected DR is shifted by one stage (bit) from TDI towards TDO on each rising edge of TCK. JTAG boundary scan logic can use this signal as an indication to shift scan data elsewhere on the chip. When deasserted, the TAP controller is not in the shift-DR state.

### C405JTGUPDATEDR (output)

When asserted, this signal indicates the TAP controller is in the update-DR state (this is one of the 16 standard TAP controller states). In this state, data in the currently selected DR is latched into the parallel outputs of the register on the falling edge of TCK (if the register has such outputs). The parallel outputs of registers with this capability do not change during the shift process. JTAG boundary scan logic can use this signal as an indication to perform a similar function elsewhere on the chip. When deasserted, the TAP controller is not in the update-DR state.

### C405JTGPGMOUT (output)

This signal indicates the state of a general purpose program bit in the JTAG debug control register (JDCR). This register is set through the external JTAG interface. This bit is intended for use by external JTAG debuggers. Its function and operation is determined by the external application.

## Debug Interface

The debug interface enables an external debugging tool (such as RISCWatch) to operate the PPC405 debug resources in external-debug mode. External-debug mode can be used to alter normal program execution and it provides the ability to debug system hardware as well as software. The mode supports starting and stopping the processor, single-stepping instruction execution, setting breakpoints, and monitoring processor status. These capabilities are described in the *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*).

### Debug Interface I/O Signal Summary

Figure 2-39 shows the block symbol for the debug interface. The signals are summarized in Table 2-22. See [Appendix A, RISCWatch and RISCTrace Interfaces](#) for information on attaching a RISCWatch to the debug interface signals.

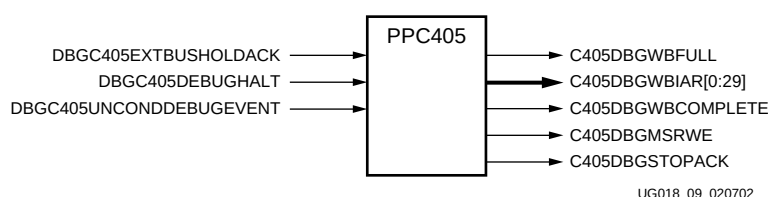


Figure 2-39: Debug Interface Block Symbol

Table 2-22: Debug Interface I/O Signals

| Signal                 | I/O Type | If Unused  | Function                                                                                |
|------------------------|----------|------------|-----------------------------------------------------------------------------------------|
| DBGC405EXTBUSHOLDACK   | I        | 0          | Indicates the bus controller has given control of the bus to an external master.        |
| DBGC405DEBUGHALT       | I        | 0          | Indicates the external debug logic is placing the processor in debug halt mode.         |
| DBGC405UNCONDDEBUEVENT | I        | 0          | Indicates the external debug logic is causing an unconditional debug event.             |
| C405DBGWBFULL          | O        | No Connect | Indicates the PPC405 writeback pipeline stage is full.                                  |
| C405DBGWBIAR[0:29]     | O        | No Connect | The address of the current instruction in the PPC405 writeback pipeline stage.          |
| C405DBGWBCOMPLETE      | O        | No Connect | Indicates the current instruction in the PPC405 writeback pipeline stage is completing. |
| C405DBGMSRWE           | O        | No Connect | Indicates the value of MSR[WE].                                                         |
| C405DBGSTOPACK         | O        | No Connect | Indicates the PPC405 is in debug halt mode.                                             |

### Debug Interface I/O Signal Descriptions

The following sections describe the operation of the debug interface I/O signals.

#### DBGC405EXTBUSHOLDACK (input)

When asserted, this signal indicates that the bus controller (for example, a PLB arbiter) has given control of the bus to an external master. When deasserted, an external master does not have control of the bus. This signal is used by the PPC405 debug logic (and the external debugger) as an indication that the processor might not have control of the bus and

therefore might not be able to respond immediately to certain debug operations. External FPGA logic generates this signal using output signals from the bus controller.

### DBG405DEBUGHALT (input)

When asserted, this signal stops the processor from fetching and executing instructions so that an external debug tool can operate the processor. From this state, known as *debug halt mode*, an external debugger controls the processor using the JTAG interface and the private JTAG hardware debug instructions. The clocks are not stopped. When this signal is deasserted, the processor operates normally.

This signal enables an external debugger to stop the processor without using the JTAG interface. A stop command issued through the JTAG interface (using a private JTAG instruction) is discarded when the processor is reset. The debug halt signal can be asserted during a reset so that the processor is stopped at the first instruction to be executed when reset is exited.

In systems that deactivate the clocks to manage power, the debug halt signal should be used to restart the clocks (if stopped) to enable an external debugger to operate the processor. After the debugger finishes its operation and deasserts the debug halt signal, the clocks can be stopped to return the processor to sleep mode.

This is a positive active signal. However, the debug halt signal produced by the RISCWatch debugger is negative active. FPGA logic that attaches to a RISCWatch debugger must invert the signal before sending it to the PPC405.

### DBG405UNCONDDEBUGEVENT (input)

When asserted, this signal causes an unconditional debug event and sets the UDE bit in the debug-status register (DBSR) to 1. When this signal is deasserted, the processor operates normally. Software can initialize the PPC405 debug resources to perform any of the following operations when an unconditional debug event occurs:

- Cause a debug interrupt in internal debug mode.
- Stop the processor in external debug mode.
- Cause a trigger event on the processor block trace interface.

### C405DBGWBFULL (output)

When asserted, this signal indicates that the PPC405 writeback-pipeline stage is full. It also indicates that writeback instruction-address bus (C405DBGWBIAR[0:29]) contains a valid instruction address. When deasserted, the writeback stage is not full and the contents of the writeback instruction-address bus are not valid.

### C405DBGWBIAR[0:29] (output)

When the writeback-full signal (C405DBGWBFULL) is asserted, this bus contains the address of the instruction in the PPC405 writeback-pipeline stage. If the writeback-full signal is not asserted, the contents of this bus are invalid.

### C405DBGWBCOMPLETE (output)

When asserted, this signal indicates that the instruction in the PPC405 writeback-pipeline stage is completing. The address of the completing instruction is contained on the writeback instruction-address bus (C405DBGWBIAR[0:29]). If the writeback-complete signal is not asserted, the instruction on the writeback instruction-address bus is not completing. The writeback-complete signal is valid only when the writeback-full signal (C405DBGWBFULL) is asserted. The signal is not valid if the writeback-full signal is deasserted.

### C405DBGMSRWE (output)

This signal indicates the state of the MSR[WE] (wait-state enable) bit. When asserted, wait state is enabled (MSR[WE]=1). When deasserted, wait state is disabled (MSR[WE]=0). When in the wait state, the processor stops fetching and executing instructions, and no longer performs memory accesses. The processor continues to respond to interrupts, and can be restarted through the use of external interrupts or timer interrupts. Wait state can also be exited when an external debug tool clears WE or when a reset occurs.

### C405DBGSTOPACK (output)

When asserted, this signal indicates that the PPC405 is in debug halt mode. When deasserted, the processor is not in debug halt mode.

## Trace Interface

The processor uses the trace interface when operating in real-time trace-debug mode. Real-time trace-debug mode supports real-time tracing of the instruction stream executed by the processor. In this mode, debug events are used to cause external trigger events. An external trace tool (such as RISCTrace) uses the trigger events to control the collection of trace information. The broadcast of trace information on the trace interface occurs independently of external trigger events (trace information is always supplied by the processor). Real-time trace-debug does not affect processor performance.

Real-time trace-debug mode is always enabled. However, the trigger events occur only when both internal-debug mode and external debug mode are disabled (DBCR0[IDM]=0 and DBCR0[EDM]=0). Most trigger events are blocked when either of those two debug modes are enabled. See the *PowerPC 405 User Manual* (Vol. 2 (a) of the *Virtex™-II Pro Developer's Kit*) for more information on debug events.

### Trace Interface Signal Summary

Figure 2-40 shows the block symbol for the trace interface. The signals are summarized in Table 2-23. See Appendix A, **RISCWatch and RISCTrace Interfaces** for information on attaching a RISCTrace to the trace interface signals.

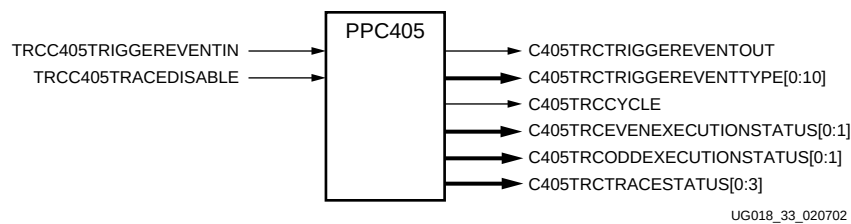


Figure 2-40: Trace Interface Block Symbol

Table 2-23: Trace Interface Signals

| Signal                          | I/O Type | If Unused                 | Function                                                                            |
|---------------------------------|----------|---------------------------|-------------------------------------------------------------------------------------|
| C405TRCTRIGGEREVENTOUT          | O        | Wrap to Trigger Event In  | Indicates a trigger event occurred.                                                 |
| C405TRCTRIGGEREVENTTYPE[0:10]   | O        | No Connect                | Specifies which debug event caused the trigger event.                               |
| C405TRCCYCLE                    | O        | No Connect                | Specifies the trace cycle.                                                          |
| C405TRCEVENEXECUTIONSTATUS[0:1] | O        | No Connect                | Specifies the execution status collected during the first of two processor cycles.  |
| C405TRCODDEXECUTIONSTATUS[0:1]  | O        | No Connect                | Specifies the execution status collected during the second of two processor cycles. |
| C405TRCTRACESTATUS[0:3]         | O        | No Connect                | Specifies the trace status.                                                         |
| TRCC405TRIGGEREVENTIN           | I        | Wrap to Trigger Event Out | Indicates a trigger event occurred and that trace status is to be generated.        |
| TRCC405TRACEDISABLE             | I        | 0                         | Disables trace collection and broadcast.                                            |

## Trace Interface I/O Signal Descriptions

The following sections describe the operation of the trace interface I/O signals.

### C405TRCTRIGGEREVENTOUT (output)

When asserted, this signal indicates that a trigger event occurred. The trigger event is caused by any debug event when both internal-debug mode and external debug mode are disabled (DBCR0[IDM]=0 and DBCR0[EDM]=0). If this signal is deasserted, no trigger event occurred.

FPGA logic can combine this signal with the trigger-event type signals to produce a qualified version of the trigger signal. The qualified signal is wrapped to the trigger-event input signal in the same trace cycle. The external trace tool also monitors the trigger-event input signal to synchronize its own trace collection. This capability can be used to implement various trace collection schemes.

### C405TRCTRIGGEREVENTTYPE[0:10] (output)

These signals are used to identify which debug event caused the trigger event. [Table 2-24](#) shows which debug event corresponds to each bit in the trigger event-type bus. The specified debug event occurred when its corresponding signal is asserted. The debug event did not occur if its corresponding signal is deasserted.

**Table 2-24: Purpose of C405TRCTRIGGEREVENTTYPE[0:10] Signals**

| Bit | Debug Event                          |
|-----|--------------------------------------|
| 0   | Instruction Address Compare 1 (IAC1) |
| 1   | Instruction Address Compare 2 (IAC2) |
| 2   | Instruction Address Compare 3 (IAC3) |
| 3   | Instruction Address Compare 4 (IAC4) |
| 4   | Data Address Compare 1 (DAC1)—Read   |
| 5   | Data Address Compare 1 (DAC1)—Write  |
| 6   | Data Address Compare 2 (DAC2)—Read   |
| 7   | Data Address Compare 2 (DAC2)—Write  |
| 8   | Trap Instruction (TDE)               |
| 9   | Exception Taken (EDE)                |
| 10  | Unconditional (UDE)                  |

FPGA logic can combine these signals with the trigger-event output signal to produce a qualified version of the trigger signal. The qualified signal is wrapped to the trigger-event input signal in the same trace cycle. The external trace tool also monitors the trigger-event input signal to synchronize its own trace collection. This capability can be used to implement various trace collection schemes.

### C405TRCCYCLE (output)

This signal defines the cycle that execution status and trace status are broadcast on the trace interface (this is referred to as the trace cycle). Although the PPC405 collects execution status and trace status every processor cycle, the information is made available to the trace interface once every two cycles. The information collected during those two cycles is broadcast over the trace interface in a single trace cycle. For this reason, the trace cycle is produced by the processor once every two processor clocks. Operating the trace interface in this manner helps reduce the amount of I/O switching during trace collection.



### C405TRCEVENEXECUTIONSTATUS[0:1] (output)

These signals are used to specify the execution status collected during the first of two processor cycles. The PPC405 collects execution status and trace status every processor cycle, but the information is made available to the trace interface once every two cycles. The information collected during those two cycles is broadcast over the trace interface in a single trace cycle.

### C405TRCODDEXECUTIONSTATUS[0:1] (output)

These signals are used to specify the execution status collected during the second of two processor cycles. The PPC405 collects execution status and trace status every processor cycle, but the information is made available to the trace interface once every two cycles. The information collected during those two cycles is broadcast over the trace interface in a single trace cycle.

### C405TRCTRACESTATUS[0:3] (output)

These signals provide additional information required by a trace tool when reconstructing an instruction execution sequence. This information is collected every processor cycle, but it is made available to the trace interface once every two cycles. The information collected during those two cycles is broadcast over the trace interface in a single trace cycle.

### TRCC405TRIGGEREVENTIN (input)

When asserted, this signal indicates that a trigger event occurred. The PPC405 uses this signal to generate additional information that is output on the trace-status bus. This information corresponds to the execution status produced on the even and odd execution-status busses. When deasserted, the information is not generated.

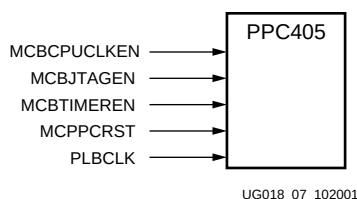
This signal can be produced by FPGA logic using the trigger event output signal. The output signal can be combined with the trigger event-type signals before it is returned as the input signal. This capability can be used to implement various trace collection schemes. The external trace tool should monitor the trigger-event input signal to synchronize its own trace collection.

### TRCC405TRACEDISABLE (input)

When asserted, this signal disables the collection and broadcast of trace information. Trace information already collected by the processor when this signal is asserted is broadcast on the trace interface before tracing is disabled. When deasserted, trace collection and broadcast proceed normally.

## Additional FPGA Specific Signals

Figure 2-41 shows the block symbol for the additional FPGA signals used by the processor block. The signals are summarized in Table 2-25.



UG018\_07\_102001

Figure 2-41: **FPGA-Specific Interface Block Symbol**

Table 2-25: **Additional FPGA I/O Signals**

| Signal      | I/O Type | If Unused | Function                                                                                             |
|-------------|----------|-----------|------------------------------------------------------------------------------------------------------|
| MCBCPUCLKEN | I        | 1         | Indicates the PPC405 clock enable should follow GWE during a partial reconfiguration.                |
| MCBJTAGEN   | I        | 1         | Indicates the JTAG clock enable should follow GWE during a partial reconfiguration.                  |
| MCBTIMEREN  | I        | 1         | Indicates the timer clock enable should follow GWE during a partial reconfiguration.                 |
| MCPPCRST    | I        | 1         | Indicates the processor block should be reset when GSR is asserted during a partial reconfiguration. |
| PLBCLK      | I        | Required  | PLB clock.                                                                                           |

### Additional FPGA I/O Signal Descriptions

The following sections describe the operation of the FPGA I/O signals.

#### MCBCPUCLKEN (input)

When asserted, this signal indicates that the enable for the core clock zone (CPMC405CPUCLKEN) should follow (match the value of) the global write enable (GWE) during the FPGA startup sequence. When deasserted, the enable for the core clock zone ignores (is independent of) the value of GWE.

#### MCBJTAGEN (input)

When asserted, this signal indicates that the enable for the JTAG clock zone (CPMC405JTAGCLKEN) should follow (match the value of) the global write enable (GWE) during the FPGA startup sequence. When deasserted, the enable for the JTAG clock zone ignores (is independent of) the value of GWE.

#### MCBTIMEREN (input)

When asserted, this signal indicates that the enable for the timer clock zone (CPMC405TIMERCLKEN) should follow (match the value of) the global write enable (GWE) during the FPGA startup sequence. When deasserted, the enable for the timer clock zone ignores (is independent of) the value of GWE.

#### MCPPCRST (input)

When asserted, this signal indicates that the processor block should be reset (the core reset signal, RSTC405RESETCORE, is asserted) when the global set reset (GSR) signal is

deasserted during the FPGA startup sequence. When MPPCRST is deasserted, the core reset signal ignores (is independent of) the value of GSR.

### PLBCLK (input)

This signal is the source clock for all PLB logic.

# *PowerPC® 405 OCM Controller*

---

## Introduction

The On-Chip Memory (OCM) controller serves as a dedicated interface between the block RAMs in the FPGA and OCM signals available on the embedded PPC405 core. The OCM signals on the processor block are designed to provide a quick access to a fixed amount of Instruction and Data memory space. The OCM controller, an integral component of the PPC405 core architecture, provides an interface to both the 64-bit Instruction-side Block RAM (ISBRAM) and the 32-bit Data-Side Block RAM (DSBRAM). The FPGA designer can choose to implement only ISBRAM, only DSBRAM, both ISBRAM and DSBRAM, or no ISBRAM and no DSBRAM. The maximum amount of memory addressable by the DSOCM and ISOCM controller ports, utilizing the fourteen least-significant addresses from the processor block, is 64KB and 128KB, respectively. The OCM controller is capable of addressing up to 16MB of DSBRAM and 16MB of ISBRAM at a reduced frequency of operation. The number of block RAMs in the device may limit the maximum amount of OCM supported.

Typical applications for Data-Side OCM (DSOCM) include scratch pad memory, as well as use of the dual-port feature of block RAM to enable a bidirectional data transfer between processor and FPGA. Typical applications for Instruction-Side OCM (ISOCM) include storage of interrupt service routines. One of the primary advantages of OCM comes from the fact that it guarantees a fixed latency of execution. Also, it reduces cache pollution and thrashing, since the cache remains available for caching code from other memory resources.

## Functional Features

### Common Features

- Separate Instruction and Data memory interface between the processor block and BRAMs in FPGA. Eliminates processor local bus (PLB) arbitration between instruction and data-side interfaces to external memory.
- Dedicated interface to Device Control Register (DCR) bus for ISOCM and DSOCM controllers. Dedicated DCR bus loop, inside the processor block, for the OCM controllers. External DCR bus output to the FPGA fabric.
- Multi-cycle mode option for I-side and D-side interfaces. Multi-cycle operation uses an N:1 processor-to-BRAM clock ratio.
- FPGA configurable DCR register addresses within DSOCM and ISOCM controllers.
- Independent 16MB logical memory space available within PPC405 memory map for each of the DSOCM and ISOCM controllers.
- Maximum of 64KB /128KB addressable from DSOCM and ISOCM interface using the fourteen least-significant bits of address outputs from processor block. Maximum of 16MB using all addresses from processor block with increased access time for BRAM.

## Data-Side OCM (DSOCM)

- 32-bit Data Read bus and 32-bit Data Write bus
- Byte write access to DSB RAM support
- Second port of dual port DSB RAM is available to read/write from an FPGA interface
- 22-bit address to DSB RAM port
- 8-bit DCR Registers: DSCNTL, DSARC
- Three alternatives to write into DSB RAM: BRAM initialization, CPU, FPGA hardware using second port. See the **Application Example** section on page 1031.

## Instruction-Side OCM (ISOCM)

The ISOCM interface contains a 64-bit read only port, for instruction fetches, and a 32-bit write only port, to initialize or test the ISB RAM. When implementing the read only port, the user must deassert the write port inputs. The preferred method of initializing the ISB RAM is through the configuration bitstream.

- 64-bit Data Read Only bus (two instructions per cycle)
- 32-bit Data Write Only bus (through DCR)
- Separate 21-bit Read Only and Write Only addresses to ISB RAM
- 8-bit DCR registers: ISCNTL, ISARC
- 32-bit DCR registers: ISINIT, ISFILL
- Two alternatives to write into ISB RAM: BRAM initialization, DCR

## OCM Controller Operation

The OCM controller is of a *distributed* style in that it is split into two blocks, one for the ISOCM interface and the other for DSOCM interface. This arrangement provides the following advantages:

- The overall efficiency of the core is improved by eliminating the need for arbitration between two sets of operations on each side, i.e., Load/Store on D-side and Fetch on I-side.
- Controller performance is improved because there is no need to share a common address and data bus between the I-side and D-side interfaces to the block RAM.
- By keeping the two interfaces separate, it is relatively easy to pick and choose one or the other interface as needed by hardware/software designers.
- The programmer's model is simplified, as there is no requirement to deal with a singular memory space between I-side and D-side interfaces.

## Operational Summary

### DSOCM Controller

The DSOCM controller accepts an address and associated control signals from the processor during a "load" instruction, and passes the valid address to the block RAM interface. For "store" instructions, a valid address from the processor is accompanied by store data in addition to associated control signals. It is important to note here that load instructions have a priority over store instructions at the DSOCM interface.

The DSOCM and ISOCM interfaces are designed to operate independently.

### ISOCM Controller

The ISOCM controller accepts an address and associated control signals from the processor during an "instruction fetch" cycle, and passes the valid address to the block RAM

interface. The instructions in ISBRAM can be stored either by loading the block RAM during FPGA configuration or by using the processor DCR bus.

The DSOCM and ISOCM interfaces are designed to operate independently.

## OCM Registers

There are two registers (DSARC, DSCNTL) in DSOCM and four registers (ISARC, ISCNTL, ISINIT, ISFILL) in ISOCM, which can be utilized by system software to set various attributes on each controller. They are further described in the **Programmer's Model** section, [page 1028](#).

## DCR Interface

The DCR interface serves two purposes:

- Allows the processor to set various attributes on each controller by reading from and writing into DSARC, DSCNTL, ISARC, and ISCNTL
- Allows the processor to write instructions into the ISOCM memory array during system initialization, using ISINIT and ISFILL

A separate DCR chain is used for ISOCM and DSOCM, which is multiplexed inside the processor block with the DCR chain external to the block.

## DSOCM Ports

Refer to [Figure 3-1](#) for the block diagram of DSOCM. All signals are in big endian format. [Figure 3-2, page 1022](#), shows an example of a DSOCM-to-BRAM interface.

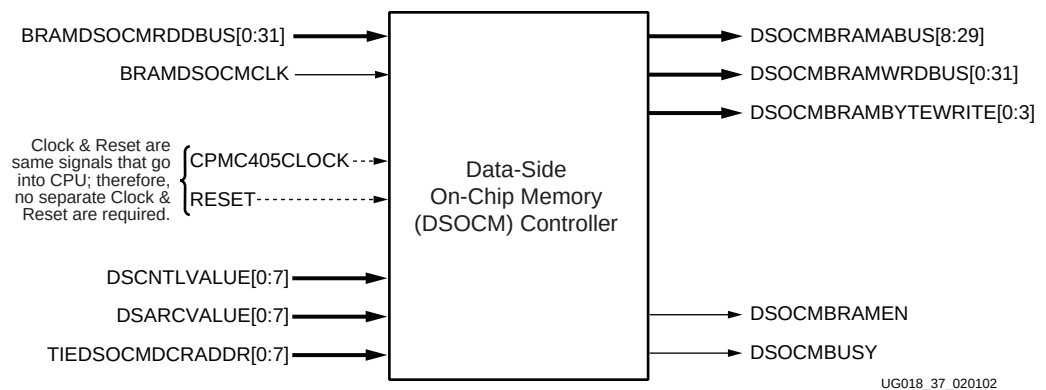


Figure 3-1: DSOCM Block

## Input Ports

### BRAMDSOCMCLK

This signal clocks the DSOCM controller. When in multi-cycle mode, BRAMDSOCMCLK is a 1:N ratio of the processor clock. The Digital Clock Manager (DCM) should be used to generate the processor clock and the DSOCM clock. The BRAMDSOCMCLK must be an integer multiple of the processor block clock CPMC405CLOCK.

### BRAMDSOCMRDDBUS[0:31]

32-bit Read data from block RAMs to DSOCM.

## Output Ports

### DSOCMBRAMABUS[8:29]

Read or Write address from DSOCM to DSBRAM. A write address is accompanied by a write enable signal for each byte lane of data. Corresponds to CPU address bits [8:29].

### DSOCMBRAMWRDBUS[0:31]

32-bit Write data from DSOCM to block RAMs.

### DSOCMBRAMBYTEWRITE[0:3]

There are four Write Enable signals to allow independent byte-wide data writes into block RAMs. DSOCMBRAMBYTEWRITE[0] qualifies writes to DSOCMBRAMWRDBUS[0:7], and DSOCMBRAMBYTEWRITE[3] qualifies writes to DSOCMBRAMWRDBUS[24:31].

### DSOCMBRAMEN

The block RAM Enable signal is asserted for both read and writes to the DSBRAM.

### DSOCMBUSY

This control signal reflects the value of the DSCNTL[2] bit out to the FPGA fabric. This signal can be used for applications that require a mechanism allowing system software to provide a particular control status to FPGA hardware. It is an optional signal and need not be used.

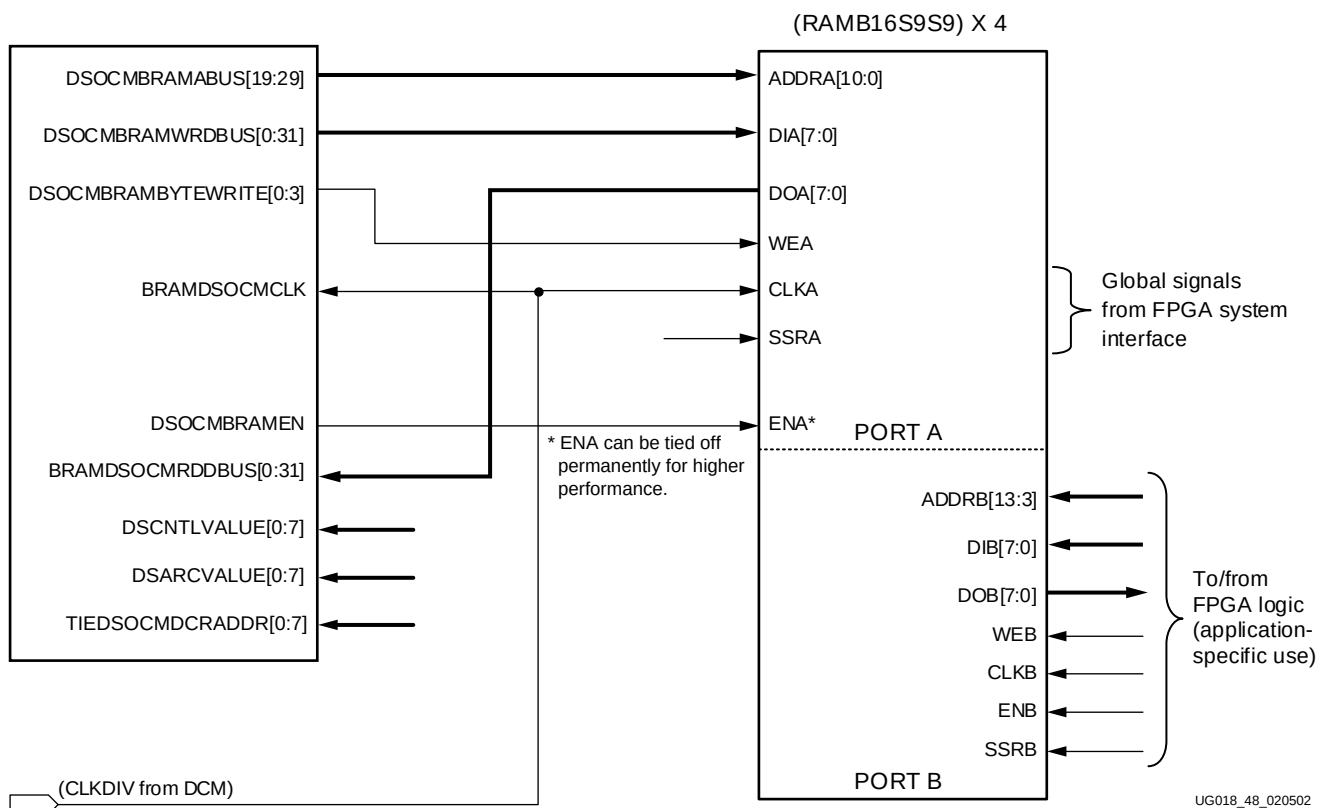


Figure 3-2: DSOCM to BRAM Interface: 8K Example

## DSOCM Attributes

Attributes are inputs to the OCM from the FPGA that must be connected to initialize registers at FPGA power up, or following a reset.

### DSCNTLVALUE[0:7]

Default value that needs to be loaded into DSCNTL register at FPGA power up. See [Figure 3-7, page 1028](#), for register bit definitions.

### DSARCVALUE[0:7]

Default value that needs to be loaded into DSARC register at FPGA power up. See [Figure 3-7, page 1028](#), for register bit definitions.

### TIEDSOCMDCRADDR[0:7]

Top 8 bits of DCR address space for DSOCM DCR registers. The DCR address space is 10 bits wide. The two least significant bits are predefined in DSOCM controller.

For example, if TIEDSOCMDCRADDR = 00 0001 11

then, address of DSARC = 00 0001 1110 = 0x01E  
 address of DSCNTL = 00 0001 1111 = 0x01F

## ISOCM Ports

Refer to [Figure 3-3](#) for the block diagram of ISOCM. All signals are in big endian format. [Figure 3-4, page 1024](#), shows an example of an ISOCM-to-BRAM interface.

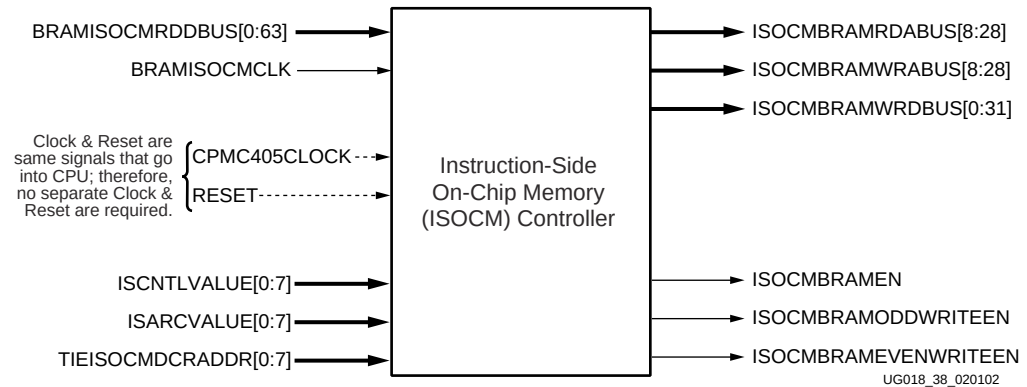


Figure 3-3: ISOCM Block

### Input Ports

#### BRAMISOCMCLK

This signal clocks the ISOCM controller. When in multi-cycle mode, BRAMISOCMCLK is a 1:N ratio of the processor clock. The Digital Clock Manager (DCM) should be used to generate the processor clock and the ISOCM clock. The BRAMISOCMCLK must be an integer multiple of the processor block clock CPMC405CLOCK.

#### BRAMISOCMRDDBUS[0:63]

64-bit read data from block RAMs, two instructions per cycle, to ISOCM.

### Output Ports

#### ISOCMBRAMRDABUS[8:28]

Read address from ISOCM to block RAM. Corresponds to CPU address bits [8:28].

#### ISOCMBRAMWRABUS[8:28]

**NOTE:** Optional. Used in dual-port BRAM interface designs only.

Write address from ISOCM to block RAMs. Initially set to value in ISINIT register.

#### ISOCMBRAMWRDBUS[0:31]

**NOTE:** Optional. Used in dual-port BRAM interface designs only.

32-bit Write data from ISOCM to block RAMs. Connect to both the even and odd write only ISBRAM data input ports. Initially set to value in ISFILL register.

#### ISOCMBRAMODDWRITEEN

**NOTE:** Optional. Used in dual-port BRAM interface designs only.



Write enable to qualify a valid write into a block RAM. This signal enables a write into block RAMs that carry odd instruction words, BRAMISOCMRDDBUS[32:63]. Connect this signal to both the Enable (EN) and Write (WR) inputs of a dual-port ISBRAM port for power savings. For single-port ISBRAM implementations, this signal can be left unconnected.

### ISOCMBRAMEVENWRITEEN

**NOTE:** Optional. Used in dual-port BRAM interface designs only.

Write enable to qualify a valid write into a block RAM. This signal enables a write into block RAMs that carry even instruction words, BRAMISOCMRDDBUS[0:31]. Connect this signal to both the Enable (EN) and Write (WR) inputs of a dual-port ISBRAM port for power savings. For single-port ISBRAM implementations, this signal can be left unconnected.

### ISOCMBRAMEN

Block RAM read enable from ISOCM to block RAMs. This signal is asserted for valid ISBRAM read cycles. For highest performance, the BRAM enable input (EN) can be locally tied to a logic 1 level. Power consumption can be reduced by connecting the BRAM enable input (EN) to this signal. Timing analysis is required to verify the design meets frequency of operation requirements if the enable is not tied to a logic 1 level.

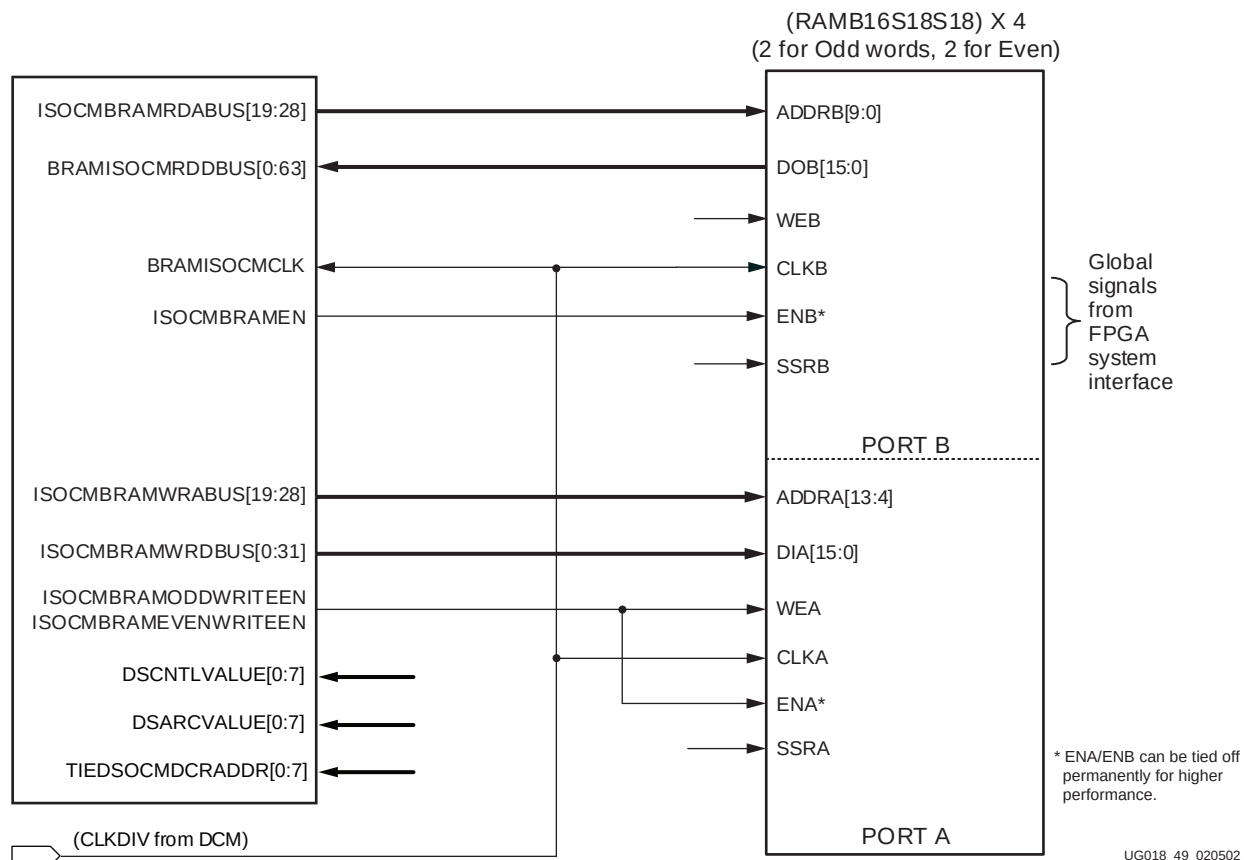


Figure 3-4: ISOCM to BRAM Interface: 8K Example

## ISOCM Attributes

Attributes are inputs to the OCM, from the FPGA, that must be connected to initialize registers at FPGA power up, or following a reset.

### ISCNTLVALUE[0:7]

Default value that needs to be loaded into ISCNTL register, at FPGA power up. See [Figure 3-8, page 1029](#), for register bit definitions.

### ISARCVAlUE[0:7]

Default value that needs to be loaded into ISARC register, at FPGA power up. See [Figure 3-8, page 1029](#), for register bit definitions.

### TIEISOCMDCRADDR[0:7]

Top 8 bits of DCR address space for ISOCM DCR registers. The DCR address space is 10 bits wide. The two least significant bits are predefined in ISOCM controller.

For example, if TIEISOCMDCRADDR = 00 0010 11

|       |                   |   |                      |   |       |
|-------|-------------------|---|----------------------|---|-------|
| then, | address of ISINIT | = | 00 0010 11 <u>00</u> | = | 0x02C |
|       | address of ISFILL | = | 00 0010 11 <u>01</u> | = | 0x02D |
|       | address of ISARC  | = | 00 0010 11 <u>10</u> | = | 0x02E |
|       | address of ISCNTL | = | 00 0010 11 <u>11</u> | = | 0x02F |

## Timing Specification

The single-cycle and multi-cycle operation modes are designed to guarantee a certain performance level by the OCM controller, assuming a certain processor clock frequency and size of BRAM. If more DSBRAM or ISBRAM is required, the designer may need to lower the clock frequency or add wait states to insure that the OCM operates correctly. The OCM clock cycle modes are selected through the DSOCMMCM and ISOCMMCM register bits in the DSCNTL and ISCNTL registers.

### Single-Cycle Mode

In single-cycle mode, the processor, OCM controller, and BRAM all run at the same clock speed. A D-side (Load/Store) or I-side (Fetch) operation completes in four CPU clock cycles. In the first cycle, the address is launched from the core to the controller. In second cycle, the core is presented with a “valid request” assertion and the controller makes the requested data available to the 405 core. In the third clock cycle, the DSBRAM accesses the memory location presented on the interface during the second clock cycle. In the fourth clock cycle, the core latches the data. See [Figure 3-5](#) for a sample operation in single-cycle mode.

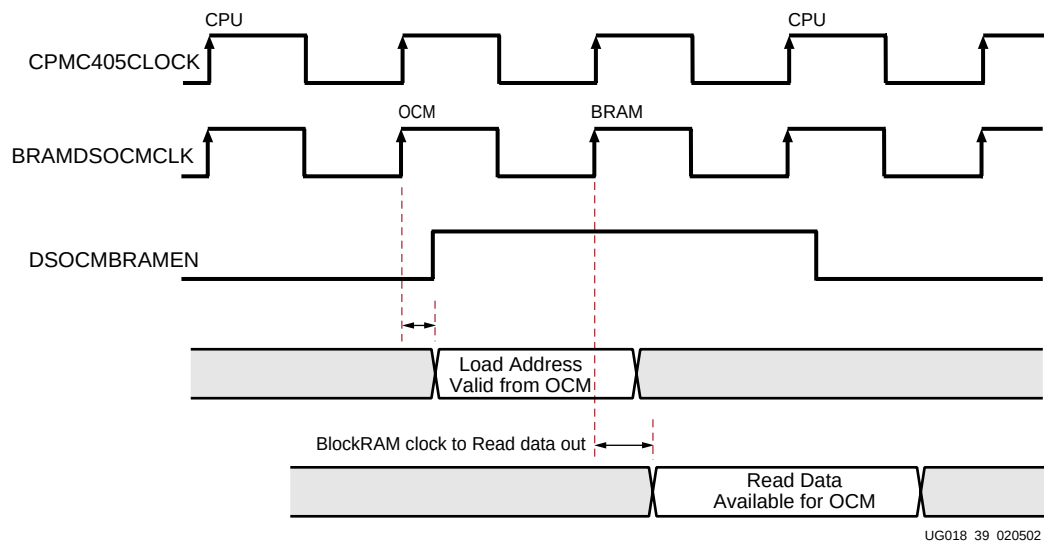


Figure 3-5: Single-Cycle Static Performance Analysis of DSOCM (Load Operation)

## Multi-Cycle Mode

Multi-cycle mode allows the processor to run at a higher clock speed than the OCM controller. Wait states are inserted between each transaction. The Digital Clock Manager must be used to generate the clocks for the processor and the OCM controllers.

See [Figure 3-6](#) for a comparative load operation transaction in multi-cycle mode.

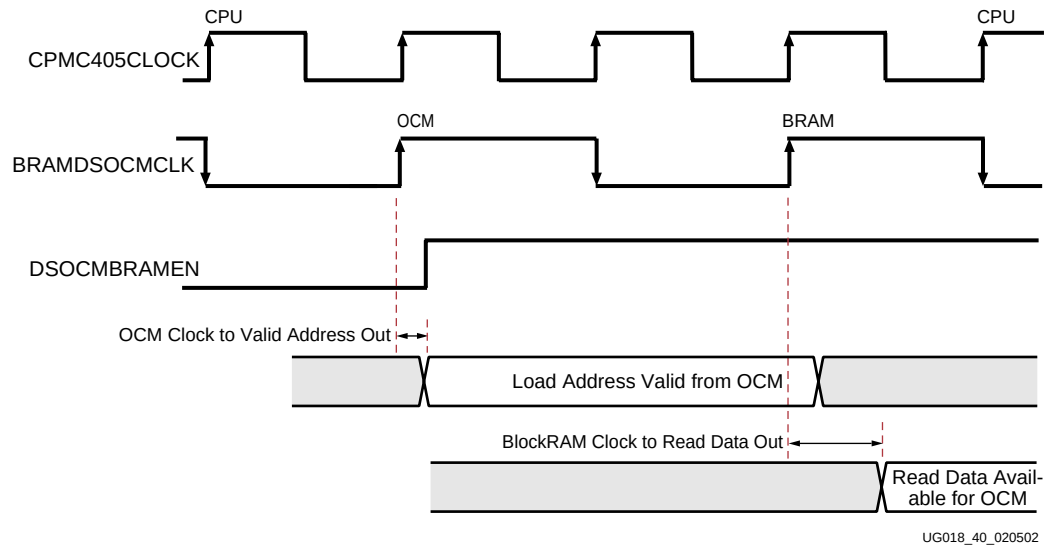


Figure 3-6: Multi-Cycle Static Performance Analysis of DSOCM (Load Operation)

# Programmer's Model

## DCR Registers

From a system software perspective, the programmer has visibility and access to DCR registers within each interface. Typically, **mtdcr** and **mfdcr** instructions can be used to write and read from these registers, respectively. All registers are read/write.

**Figure 3-7** lists all the DCR registers and the bit definitions of registers on the D-side OCM interface. **Figure 3-8** lists all the DCR registers and the bit definitions of registers on the I-side OCM interface.

### External Registers

Allocated within DCR address space  
(Programmer's Model)

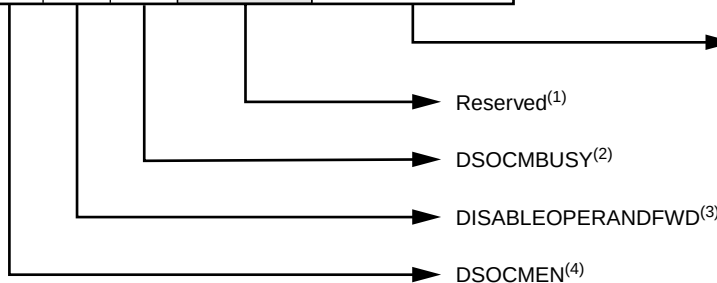
| DSARC (DSOCM Address Range Compare Register) |      |      |      |      |      |      |      |
|----------------------------------------------|------|------|------|------|------|------|------|
| 0                                            | 1    | 2    | 3    | 4    | 5    | 6    | 7    |
| A0/P                                         | A1/P | A2/P | A3/P | A4/P | A5/P | A6/P | A7/P |

**8 bits:** Address range compare for DSOCM memory space. They are also configurable via FPGA, through the DSARCVAlUE inputs to the processor block.

**Note:** The top 8 bits of the CPU address are compared with DSARC to provide a 16 MB logical address for space for DSOCM block.

| DSCNTL (DCR Control Register) |      |      |      |      |         |   |      |
|-------------------------------|------|------|------|------|---------|---|------|
| 0                             | 1    | 2    | 3    | 4    | 5       | 6 | 7    |
| D0/P                          | D1/P | D2/P | D3/P | D4/P | D5/P... |   | D7/P |

**8 bits:** Control Register for DSOCM. They are also configurable via FPGA, through the DSCNTLVALUE inputs to the processor block.



| DSOCMMCM[0:2] | CPMC405CLOCK: BRAMDSOCMCLK Ratio |
|---------------|----------------------------------|
| 000           | Not supported                    |
| 001           | 1:1                              |
| 010           | Not supported                    |
| 011           | 2:1                              |
| 100           | Not supported                    |
| 101           | 3:1                              |
| 110           | Not supported                    |
| 111           | 4:1                              |

$2n - 1$

where  $n$  = number of processor clocks in one OCM clock cycle. Must be an integer.

### Notes:

1. Reserved bits must be configured to 0.
2. See section "DSOCM Ports" in the text.
3. DISABLEOPERANDFWD:  
When DISABLEOPERANDFWD is asserted, load data from the DSOCM goes directly into a latch in the processor block. This causes an additional cycle (a total of two cycles) of latency between a load instruction which is followed by an instruction that requires the load data as an operand.  
When DISABLEOPERANDFWD is *not* asserted, load data from the DSOCM must pass through steering logic before arriving at a latch. This causes a single cycle of latency between a load instruction which is followed by an instruction that requires the load data as an operand.
4. DSOCMEN:  
Enables the DSOCM address decoder.

UG018\_46\_020502

Figure 3-7: DSOCM DCR Registers

### External Registers

Allocated within DCR address space  
(Programmer's Model)

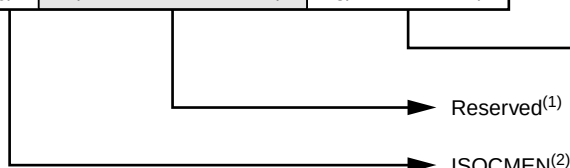
| ISARC (ISOCM Address Range Compare Register) |      |      |      |      |      |      |      |
|----------------------------------------------|------|------|------|------|------|------|------|
| 0                                            | 1    | 2    | 3    | 4    | 5    | 6    | 7    |
| A0/P                                         | A1/P | A2/P | A3/P | A4/P | A5/P | A6/P | A7/P |

**8 bits:** Address range compare for ISOCM memory space. They are also configurable via FPGA, through the ISARCVLUE inputs to the processor block.

**Note:** The top 8 bits of the CPU address are compared with ISARC to provide a 16 MB logical address for space for ISOCM block

| ISCNTL (ISOCM Control Register) |         |   |   |      |         |   |      |
|---------------------------------|---------|---|---|------|---------|---|------|
| 0                               | 1       | 2 | 3 | 4    | 5       | 6 | 7    |
| D0/P                            | D1/P... |   |   | D4/P | D5/P... |   | D7/P |

**8 bits:** Control Register for ISOCM. They are also configurable via FPGA, through the ISCNTLVUE inputs to the processor block.



#### Notes:

1. Reserved bits must be configured to 0.
2. ISOCMEN:  
Enables the DSOCM address decoder.

| ISOCMMCM[0:2] | CPMC405CLOCK:<br>BRAMISOCMCLK<br>Ratio |
|---------------|----------------------------------------|
| 000           | Not supported                          |
| 001           | 1:1                                    |
| 010           | Not supported                          |
| 011           | 2:1                                    |
| 100           | Not supported                          |
| 101           | 3:1                                    |
| 110           | Not supported                          |
| 111           | 4:1                                    |

$$2n - 1$$

where  $n$  = number of processor clocks in one OCM clock cycle.  
Must be an integer.

| ISINIT (ISOCM Initialization Address) |    |        |        |     |     |
|---------------------------------------|----|--------|--------|-----|-----|
| 0                                     | 1  | 2...   | ...18  | 19  | 20  |
| A8                                    | A9 | A10... | ...A26 | A27 | A28 |

**21 bits:** Initialization address to write into ISOCM memory space.  
The address is incremented by 1 for every write into ISFILL register, which is described below.

| ISFILL (ISOCM Fill Data Register) |    |       |        |     |     |
|-----------------------------------|----|-------|--------|-----|-----|
| 0                                 | 1  | 2...  | ...29  | 30  | 31  |
| D0                                | D1 | D2... | ...D29 | D30 | D31 |

**32 bits:** Data register for ISOCM, used to send instructions via DCR into ISOCM memory space.

UG018\_47\_020502

Figure 3-8: ISOCM DCR Registers

The I-OCM and D-OCM interfaces provide DCRs (DSARC & ISARC) which set the top 8 (base) address bits of the I-OCM and D-OCM (CPU address bits 0:7). These addresses can be used to independently place the I-OCM and D-OCM in any 16MB address range. The I-OCM and D-OCM hardware outputs a maximum of 22 address bits (data-side address bits [8:29] and instruction-side address bits [8:28]) to address block RAM.

The OCM clock cycle modes are selected through the MULTICYCLEMODE control bits (DSOCMMCM and ISOCMMCM) in the DSCNTL and ISCNTL registers. The OCM

memory space decoders are enabled when the DSOCMEN and ISOCMEN bits are asserted in the DSCNTL and ISCNTL registers.

## References

1. *Virtex-II Pro Platform FPGA Handbook*
2. *PPC405 Processor Block Manual*
3. *PPC405 User Manual*

## Application Notes

### Interfacing to Block RAM

Refer to the *Virtex-II Pro Platform FPGA Handbook* for a detailed description of the block RAM resources. The block RAMs are synchronous and provide a dual-port access to 16K bits of data, not including parity. A useful feature is the different configurations each block RAM can have. This is shown in [Table 3-1](#).

**Table 3-1: Block RAM Configurations in a Virtex-II FPGA**

| Word Width | Depth  | Address | Data             |
|------------|--------|---------|------------------|
| 1          | 16,384 | A<13:0> | D<0>             |
| 2          | 8192   | A<13:1> | D<1:0>           |
| 4          | 4096   | A<13:2> | D<3:0>           |
| 9          | 2048   | A<13:3> | D<7:0> + P<0>    |
| 18         | 1024   | A<13:4> | D<15:0> + P<1:0> |
| 36         | 512    | A<13:5> | D<31:0> + P<3:0> |

Block RAMs can be used as the source of data and instruction storage space for OCM. Since the block RAMs are not dedicated to the OCM, any size of block RAM desired can be used.

The **Application Example** section shows a reference OCM design with both ISBRAM and DSBRAM.

### Size vs. Performance

For a given processor clock frequency, the larger the size of block RAM, the greater the associated performance penalty. Past a certain threshold of BRAM, the OCM may not function correctly if the block RAM access time is greater than the microprocessor clock period. For designs with large amounts of block RAM, the designer may need to use multi-cycle mode and/or reduce the microprocessor clock frequency.

There may be applications associated with OCM that necessitate a dual-port implementation of block RAMs rather than the default single-port implementation. [Table 3-2](#) shows a sample block RAM implementation for the I-side and D-side OCM interfaces, illustrating the recommended method to connect ISBRAM and DSBRAM. The data and control signals will be routed differently depending upon the total amount of memory required for the application.

The instruction-side and data-side clock mode control bits (ISOCMMCM and DSOCMMCM) will change depending upon processor clock frequency, BRAM access time, signal loading, and signal routing delays. As the total amount of OCM increases, the performance decreases. The minimum DSBRAM is 8KB and the minimum ISBRAM is 4KB.

**Table 3-2: ISBRAM and DSBRAM Configuration Range Using the 14 Least-Significant Processor Block Address Outputs**

| Total Memory Used | 32-Bit Data OCM |                |                     | 64-Bit Instruction OCM |                |                     |
|-------------------|-----------------|----------------|---------------------|------------------------|----------------|---------------------|
|                   | BRAM Quantity   | BRAM Port Size | Clock Mode DSOCMMCM | BRAM Quantity          | BRAM Port Size | Clock Mode ISOCMMCM |
| 128KB             | No support      |                | TBD                 | 64                     | 16K x 1        | TBD                 |
| 64KB              | 32              | 16K x 1        | TBD                 | 32                     | 8K x 2         | TBD                 |
| 32KB              | 16              | 8K x 2         | TBD                 | 16                     | 4K x 4         | TBD                 |
| 16KB              | 8               | 4K x 4         | TBD                 | 8                      | 2K x 9         | TBD                 |
| 8KB               | 4               | 2K x 9         | TBD                 | 4                      | 1K x 16        | TBD                 |
| 4KB               | Note (1)        | 1K x 16        |                     | 2                      | 512 x 36       | TBD                 |
| 2KB               | Note (1)        | 512 x 36       |                     | Note (2)               |                |                     |

**Notes:**

1. The processor byte write function is not supported for this configuration.
2. The ISBRAM must be 64 bits wide.
3. For larger ISBRAM and DSBRAM configurations, additional address outputs can be used with increased access times.

## Application Example

This section contains a Verilog reference design containing an OCM interface to a 4KB ISBRAM and a dual port 8KB DSBRAM.

```
//*****
// File: ocm_example.v
//
// PowerPC 405 On Chip Memory Controller Application Example
// Features:
// 1. Digital Clock Manager for PPC/OCM/BRAM clock generation.
// 2. REFCLK input generates 2X and 4X clocks.
// 3. On Chip Memory Controller will use CLK2X.
// 4. DSBRAM and ISBRAM also use CLK2X.
// 5. PowerPC 405 will use CLK4X.
// 6. 4KB ISBRAM dual ported design for DCR write access.
// 7. 8KB DSBRAM dual ported design for FPGA R/W access.
//
// Application Notes:
// 1. User connects this module up to the Processor Block.
// 2. Processor block inputs not shown in this example:
// A. DSARCVALUE = 8'hF4
// B. DSCNTLVALUE = 8'h83 (2:1 PPC/OCM clock ratio)
// C. ISARCVALUE = 8'hFF
// D. ISCNTLVALUE = 8'h83 (2:1 PPC/OCM clock ratio)
// 3. Processor Block OCM outputs not used: DSOCMBUSY
//*****

module OCM (REFCLK, BRAMDSOCCLK, BRAMISOCCLK, CPMC405CLOCK, RST,
 DSOCMBRAMABUS, DSOCMBRAMEN, DSOCMBRAMWRDBUS, BRAMDSOCMRDDBUS,
 DSOCMBRAMBYTEWRITE, ADDR8, ENB, CLKB, DINB, DOUTB, WEB,
 ISOCMBRAMRDABUS, ISOCMBRAMWRABUS, ISOCMBRAMEN,
 ISOCMBRAMWRDBUS, ISOCMBRAMODDWRITEEN, ISOCMBRAMEVENWRITEEN,
 BRAMISOCMRDDBUS);
```



```
// Digital Clock Manager I/O:
input REFCLK;
output CPMC405CLOCK; //PowerPC405 clock
output RST;

//*****
// Data Side On Chip Memory I/O:
// NOTE: OCM interface uses big endian bit format.
//*****

input [8:29] DSOCMBRAMABUS; // A29 is LSB
input DSOCMBRAMEN; // BRAM enable
output BRAMDSOCCLK; // DSOCM and DSBRAM clock
input [0:31] DSOCMBRAMWRDBUS; // Bit 31 is the LSB!
output [0:31] BRAMDSOCMRDDBUS; // Bit 31 is the LSB!
input [0:3] DSOCMBRAMBYTEWRITE; // Bit 0 controls the MSB byte

//*****
// FPGA logic interface to "B" side of Dual Port BRAM:
// NOTE: FPGA interface uses little endian bit format.
//*****

input [10:0] ADDR_B; // FPGA address, A0 is LSB
input ENB; // FPGA enable
input CLKB; // FPGA clock
input [31:0] DINB; // FPGA side data input
output [31:0] DOUTB; // FPGA side data output
input [3:0] WEB; // FPGA write enables

//*****
// Instruction Side On Chip Memory I/O:
// NOTE: OCM interface uses big endian bit format.
//*****

input [8:28] ISOCMBRAMWRABUS; // BRAM Write Address
input [8:28] ISOCMBRAMRDABUS; // BRAM Read Address
input ISOCMBRAMEN; // BRAM enable
output BRAMISOCCLK; // ISOCM and ISBRAM clock
input [0:31] ISOCMBRAMWRDBUS; // Store data bus to BRAM
input ISOCMBRAMODDWRITEEN; // Odd word write enable
input ISOCMBRAMEVENWRITEEN; // Even word write enable
output [0:63] BRAMISOCMRDDBUS; // Load data bus from BRAM

// Instantiate OCM Application Example lower level modules here:

wire OCMCLK;
wire CPMC405CLOCK;

assign BRAMDSOCCLK = OCMCLK;
assign BRAMISOCCLK = OCMCLK;

ocmclk OCM1
(
 .REFCLK (REFCLK),
 .CLK (),
 .CLK2X (OCMCLK),
 .CLKMULT (CPMC405CLOCK),
 .RST (RST)
);

DS_bram_wrap OCM2
```

```

(
.DSOCMBRAMABUS (DSOCMBRAMABUS[19:29]),
.DSOCMBRAMEN (DSOCMBRAMEN),
.BRAMDSOCMCLK (OCMCLK),
.DSOCMBRAMWRDBUS (DSOCMBRAMWRDBUS),
.BRAMDSOCMRDDBUS (BRAMDSOCMRDDBUS),
.DSOCMBRAMBYTEWRITE (DSOCMBRAMBYTEWRITE),
.ADDRB (ADDR),
.ENB (ENB),
.CLKB (CLKB),
.DINB (DINB),
.DOUTB (DOUTB),
.WEB (WEB)
);

IS_bram_wrap OCM3
(
.ISOCMBRAMRDABUS (ISOCMBRAMRDABUS[20:28]),
.ISOCMBRAMWRABUS (ISOCMBRAMWRABUS[20:28]),
.ISOCMBRAMEN (ISOCMBRAMEN),
.BRAMISOCMCLK (OCMCLK),
.ISOCMBRAMWRDBUS (ISOCMBRAMWRDBUS),
.ISOCMBRAMODDWRITEEN (ISOCMBRAMODDWRITEEN),
.ISOCMBRAMEVENWRITEEN (ISOCMBRAMEVENWRITEEN),
.BRAMISOCMRDDBUS (BRAMISOCMRDDBUS[0:63])
);

endmodule // OCM

module ocmclk (REFCLK, CLK, CLK2X, CLKMULT, RST);

input REFCLK; // input clock
output CLK; // buffered 1X clock
output CLK2X; // buffered 2X clock
output CLKMULT; // buffered 4X clock
output RST; // logic RST

wire dcm_locked, refclk_in;
wire clk_i, clk2x_i, clkmult_i;
reg RST;
reg [2:0] startup_counter;

// Clock Generation, REFCLK = CLK, Multiply Clock Up to CPU Freq
IBUFG buf0 (.I(REFCLK), .O(refclk_in));

// Set to 4X REFCLK for CPU clock multiplier
defparam dcm1.CLKFX_MULTIPLY = 12'h004;

DCM dcm1 (.CLKFB (CLK),
.CLKIN (refclk_in),
.DSSEN (1'b0),
.PSCLK (1'b0),
.PSEN (1'b0),
.PSINCDEC (1'b0),
.RST (1'b0),
.CLK0 (clk_i),
.CLK90 (),
.CLK180 (),
.CLK270 (),

```

```

 .CLK2X (clk2x_i),
 .CLK2X180 (),
 .CLKDV (),
 .CLKFX (clkmult_i),
 .CLKFX180 (),
 .LOCKED (dcm_locked),
 .PSDONE (),
 .STATUS ());

BUFG buf1 (.I (clk_i), .0 (CLK));
BUFG buf2 (.I (clk2x_i), .0 (CLK2X));
BUFG buf3 (.I (clkmult_i), .0 (CLKMULT));

// Startup Reset Signal
always @ (posedge CLK)
 if (!dcm_locked)
 startup_counter <= 3'b0;
 else if (startup_counter != 3'b111)
 startup_counter <= startup_counter + 1;

always @ (posedge CLK or negedge dcm_locked)
 if (!dcm_locked)
 RST <= 1'b1;
 else
 RST <= (startup_counter != 3'b111);

endmodule //ocmclk

//*****
// Data Side BlockSelect RAM interface module.
// DSBRAMS are enabled once DSOCMEN is asserted.
//
//*****

module DS_bram_wrap (DSOCMBRAMABUS,
 DSOCMBRAMEN,
 BRAMDSOCCLK,
 DSOCMBRAMWRDBUS,
 BRAMDSOCMRDDBUS,
 DSOCMBRAMBYTEWRITE,
 ADDR0,
 ENB,
 CLKB,
 DINB,
 DOUTB,
 WEB
);

// DSOCM controller interface to "A" side of Dual Port BRAM.
// Processor side supports byte writes to data buffer.

input [10:0] DSOCMBRAMABUS; // A00 is the least significant
input DSOCMBRAMEN; // BRAM chip select
input BRAMDSOCCLK; // DSOCM clock with 180 deg phase shift
input [31:0] DSOCMBRAMWRDBUS; // Din0 is the LSB
output [31:0] BRAMDSOCMRDDBUS; // Dout0 is the LSB
input [3:0] DSOCMBRAMBYTEWRITE; // Bit 3 controls the MSB byte

// FPGA logic interface to "B" side of Dual Port BRAM:

```

```

input [10:0] ADDRb; // FPGA address
input ENb; // FPGA enable
input CLKb; // FPGA clock
input [31:0] DINb; // FPGA side data input
output [31:0] DOUTb; // FPGA side data output
input [3:0] WEB; // FPGA write enables

RAMB16_S9_S9 u3 (// PPC405 Byte 0 (MSB)
 .WEA (DSOCMBRAMBYTEWRITE[3]),
 .ENA (DSOCMBRAMEN),
 .SSRA (1'b0),
 .CLKA (BRAMDSOCMCLK),
 .ADDRA (DSOCMBRAMABUS[10:0]),
 .DIPA (1'b0),
 .DIA (DSOCMBRAMWRDBUS[31:24]),
 .DOA (BRAMDSOCMRDDBUS[31:24]),
 .DOPA (),

 // FPGA I/F Byte 3 (MSB)
 .WEB (WEB[3]),
 .ENB (ENb),
 .SSRB (1'b0),
 .CLKB (CLKb),
 .ADDRB (ADDRb[10:0]),
 .DIPB (1'b0),
 .DIB (DINb[31:24]),
 .DOB (DOUTb[31:24]),
 .DOPB ()
);

RAMB16_S9_S9 u2 (// PPC405 Byte 1
 .WEA (DSOCMBRAMBYTEWRITE[2]),
 .ENA (DSOCMBRAMEN),
 .SSRA (1'b0),
 .CLKA (BRAMDSOCMCLK),
 .ADDRA (DSOCMBRAMABUS[10:0]),
 .DIPA (1'b0),
 .DIA (DSOCMBRAMWRDBUS[23:16]),
 .DOA (BRAMDSOCMRDDBUS[23:16]),
 .DOPA (),

 // FPGA I/F Byte 2
 .WEB (WEB[2]),
 .ENB (ENb),
 .SSRB (1'b0),
 .CLKB (CLKb),
 .ADDRB (ADDRb[10:0]),
 .DIPB (1'b0),
 .DIB (DINb[23:16]),
 .DOB (DOUTb[23:16]),
 .DOPB ()
);

RAMB16_S9_S9 u1 (// PPC405 Byte 2
 .WEA (DSOCMBRAMBYTEWRITE[1]),
 .ENA (DSOCMBRAMEN),
 .SSRA (1'b0),
 .CLKA (BRAMDSOCMCLK),
 .ADDRA (DSOCMBRAMABUS[10:0]),
 .DIPA (1'b0),
 .DIA (DSOCMBRAMWRDBUS[15:8]),

```

```

 .DOA (BRAMDSOCMRDDBUS[15:8]),
 .DOPA (),

 // FPGA I/F Byte 1
 .WEB (WEB[1]),
 .ENB (ENB),
 .SSRB (1'b0),
 .CLKB (CLKB),
 .ADDRB (ADDR[10:0]),
 .DIPB (1'b0),
 .DIB (DINB[15:8]),
 .DOB (DOUTB[15:8]),
 .DOPB ()
);

 RAMB16_S9_S9 u0 (// PPC405 Byte 3 (LSB)
 .WEA (DSOCMBRAMBYTEWRITE[0]),
 .ENA (DSOCMBRAMEN),
 .SSRA (1'b0),
 .CLKA (BRAMDSOCMCLK),
 .ADDRA (DSOCMBRAMABUS[10:0]),
 .DIPA (1'b0),
 .DIA (DSOCMBRAMWRDBUS[7:0]),
 .DOA (BRAMDSOCMRDDBUS[7:0]),
 .DOPA (),

 // FPGA I/F Byte 0 (LSB)
 .WEB (WEB[0]),
 .ENB (ENB),
 .SSRB (1'b0),
 .CLKB (CLKB),
 .ADDRB (ADDR[10:0]),
 .DIPB (1'b0),
 .DIB (DINB[7:0]),
 .DOB (DOUTB[7:0]),
 .DOPB ()
);

endmodule //DS_bram_wrap

//*****
// Instruction Side BRAM interface module.
// ISBRAMS are enabled only when ISOCMBRAMEN is asserted.
//*****

module IS_bram_wrap
(
 ISOCMBRAMRDABUS,
 ISOCMBRAMWRABUS,
 ISOCMBRAMEN,
 BRAMISOCMCLK,
 ISOCMBRAMWRDBUS,
 ISOCMBRAMODDWRITEEN,
 ISOCMBRAMEVENWRITEEN,
 BRAMISOCMRDDBUS
);

 input [8:0] ISOCMBRAMWRABUS; // BRAM Write Address
 input [8:0] ISOCMBRAMRDABUS; // BRAM Read Address
 input ISOCMBRAMEN; // BRAM enable
 input BRAMISOCMCLK; // BRAM clock

```

```

input [31:0] ISOCMBRAMWRDBUS; // Store data bus to BRAM
input ISOCMBRAMODDWRITEEN; // BRAM Odd word write enable
input ISOCMBRAMEVENWRITEEN; // BRAM Even word write enable
output [0:63] BRAMISOCMRDDBUS; // Load data bus from BRAM

```

```

RAMB16_S36_S36 u2 (.DOA (BRAMISOCMRDDBUS[32:63]),
 .DOB (),
 .DOPA (),
 .DOPB (),
 .ADDRA (ISOCMBRAMRDABUS[8:0]),
 .CLKA (BRAMISOCMCLK),
 .DIA (ISOCMBRAMWRDBUS[31:0]),
 .DIPA (4'b0000),
 .ENA (ISOCMBRAMEN),
 .SSRA (1'b0),
 .WEA (1'b0),
 .ADDRB (ISOCMBRAMWRABUS[8:0]),
 .CLKB (BRAMISOCMCLK),
 .DIB (ISOCMBRAMWRDBUS[31:0]),
 .DIPB (4'b0000),
 .ENB (ISOCMBRAMEN),
 .SSRB (1'b0),
 .WEB (ISOCMBRAMODDWRITEEN)
);

RAMB16_S36_S36 u1 (.DOA (BRAMISOCMRDDBUS[0:31]),
 .DOB (),
 .DOPA (),
 .DOPB (),
 .ADDRA (ISOCMBRAMRDABUS[8:0]),
 .CLKA (BRAMISOCMCLK),
 .DIA (ISOCMBRAMWRDBUS[31:0]),
 .DIPA (4'b0000),
 .ENA (ISOCMBRAMEN),
 .SSRA (1'b0),
 .WEA (1'b0),
 .ADDRB (ISOCMBRAMWRABUS[8:0]),
 .CLKB (BRAMISOCMCLK),
 .DIB (ISOCMBRAMWRDBUS[31:0]),
 .DIPB (4'b0000),
 .ENB (ISOCMBRAMEN),
 .SSRB (1'b0),
 .WEB (ISOCMBRAMEVENWRITEEN)
);

endmodule //IS_bram_wrap

```



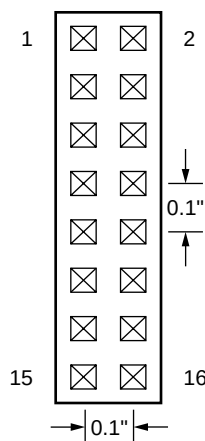
## *RISCWatch and RISCTrace Interfaces*

This appendix summarizes the interface requirements between the PPC405x3 and the RISCWatch and RISCTrace tools.

The requirement for separate JTAG and trace connectors is being replaced with a single Mictor connector to improve the electrical and mechanical characteristics of the interface. Pin assignments for the Mictor connector are included in the signal-mapping tables.

## RISCWatch Interface

The RISCWatch tool communicates with the PPC405x3 using the JTAG and debug interfaces. It requires a 16-pin, male 2x8 header connector located on the target development board. The layout of the connector is shown in [Figure A-1](#) and the signals are described in [Table A-1](#). A mapping of PPC405x3 to RISCWatch signals is provided in [Table A-2](#). At the board level, the connector should be placed as close as possible to the processor chip to ensure signal integrity. Position 14 is used as a connection key and does not contain a pin.



UG018 50 100901

**Figure A-1: JTAG-Connector Physical Layout**



Table A-1: JTAG Connector Signals for RISCWatch

| Pin | RISCWatch  |                                           | Description                |
|-----|------------|-------------------------------------------|----------------------------|
|     | I/O        | Signal Name                               |                            |
| 1   | Input      | TDO                                       | JTAG test-data out.        |
| 2   | No Connect | Reserved                                  |                            |
| 3   | Output     | TDI <sup>1</sup>                          | JTAG test-data in.         |
| 4   | Output     | $\overline{\text{TRST}}$                  | JTAG test reset.           |
| 5   | No Connect | Reserved                                  |                            |
| 6   | Output     | +Power <sup>2</sup>                       | Processor power OK         |
| 7   | Output     | TCK <sup>3</sup>                          | JTAG test clock.           |
| 8   | No Connect | Reserved                                  |                            |
| 9   | Output     | TMS                                       | JTAG test-mode select.     |
| 10  | No Connect | Reserved                                  |                            |
| 11  | Output     | $\overline{\text{HALT}}$                  | Processor debug halt mode. |
| 12  | No Connect | Reserved                                  |                            |
| 13  | No Connect | Reserved                                  |                            |
| 14  | KEY        | No pin should be placed at this position. |                            |
| 15  | No Connect | Reserved                                  |                            |
| 16  |            | GND                                       | Ground                     |

**Notes:**

1. A 10K $\Omega$  pull-up resistor should be connected to this signal to reduce chip-power consumption. The pull-up resistor is not required.
2. The +POWER signal, is provided by the board, and indicates whether the processor is operating. This signal does not supply *power* to the debug tools or to the processor. A series resistor (1K $\Omega$  or less) should be used to provide short-circuit current-limiting protection.
3. A 10K $\Omega$  pull-up resistor must be connected to these signals to ensure proper chip operation when these inputs are not used.

Table A-2: PPC405x3 to RISCWatch Signal Mapping

| PPC405x3                     |        | RISCWatch                |        | JTAG Connector Pin | Mictor Connector Pin |
|------------------------------|--------|--------------------------|--------|--------------------|----------------------|
| Signal                       | I/O    | Signal                   | I/O    |                    |                      |
| C405JTGTDO <sup>1</sup>      | Output | TDO                      | Input  | 1                  | 11                   |
| JTGC405TDI                   | Input  | TDI                      | Output | 3                  | 19                   |
| JTGC405TRSTNEG               | Input  | $\overline{\text{TRST}}$ | Output | 4                  | 21                   |
| JTGC405TCK                   | Input  | TCK                      | Output | 7                  | 15                   |
| JTGC405TMS                   | Input  | TMS                      | Output | 9                  | 17                   |
| DBG405DEBUGHALT <sup>2</sup> | Input  | $\overline{\text{HALT}}$ | Output | 11                 | 7                    |

**Notes:**

1. This signal must be driven by a tri-state device using C405JTGTDOEN as the enable signal.
1. This signal must be inverted between the PPC405x3 and the RISCWatch.

# RISCTrace Interface

The RISCTrace tool communicates with the PPC405x3 using the trace interface. It requires a 20-pin, male 2x10 header connector (3M 3592-6002 or equivalent) located on the target development board. The layout of the connector is shown in [Figure A-2](#) and the signals are described in [Table A-3](#). A mapping of PPC405x3 to RISCTrace signals is provided in [Table A-4](#). At the board level, the connector should be placed as close as possible to the processor chip to ensure signal integrity. An index at pin one and a key notch on the same side of the connector as the index are required.

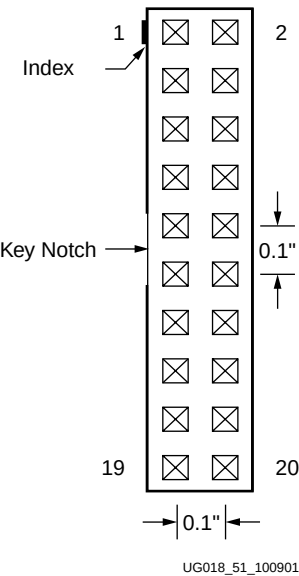


Figure A-2: Trace-Connector Physical Layout

Table A-3: Trace Connector Signals for RISCTrace

| Pin | RISCTrace  |             | Description       |
|-----|------------|-------------|-------------------|
|     | I/O        | Signal Name |                   |
| 1   | No Connect | Reserved    |                   |
| 2   | No Connect | Reserved    |                   |
| 3   | Output     | TrcClk      | Trace cycle.      |
| 4   | No Connect | Reserved    |                   |
| 5   | No Connect | Reserved    |                   |
| 6   | No Connect | Reserved    |                   |
| 7   | No Connect | Reserved    |                   |
| 8   | No Connect | Reserved    |                   |
| 9   | No Connect | Reserved    |                   |
| 10  | No Connect | Reserved    |                   |
| 11  | No Connect | Reserved    |                   |
| 12  | Output     | TS1O        | Execution status. |
| 13  | Output     | TS2O        | Execution status. |
| 14  | Output     | TS1E        | Execution status. |

Table A-3: Trace Connector Signals for RISCTrace (Continued)

| Pin | RISCTrace |             | Description       |
|-----|-----------|-------------|-------------------|
|     | I/O       | Signal Name |                   |
| 15  | Output    | TS2E        | Execution status. |
| 16  | Output    | TS3         | Trace status.     |
| 17  | Output    | TS4         | Trace status.     |
| 18  | Output    | TS5         | Trace status.     |
| 19  | Output    | TS6         | Trace status.     |
| 20  |           | GND         | Ground            |

Table A-4: PPC405x3 to RISCTrace Signal Mapping

| PPC405x3                      |        | RISCTrace |       | Trace Connector Pin | Mictor Connector Pin |
|-------------------------------|--------|-----------|-------|---------------------|----------------------|
| Signal                        | I/O    | Signal    | I/O   |                     |                      |
| C405TRCCYCLE                  | Output | TrcClk    | Input | 3                   | 6                    |
| C405TRCODDEXECUTIONSTATUS[0]  | Output | TS1O      | Input | 12                  | 24                   |
| C405TRCODDEXECUTIONSTATUS[1]  | Output | TS2O      | Input | 13                  | 26                   |
| C405TRCEVENEXECUTIONSTATUS[0] | Output | TS1E      | Input | 14                  | 28                   |
| C405TRCEVENEXECUTIONSTATUS[1] | Output | TS2E      | Input | 15                  | 30                   |
| C405TRCTRACESTATUS[0]         | Output | TS3       | Input | 16                  | 32                   |
| C405TRCTRACESTATUS[1]         | Output | TS4       | Input | 17                  | 34                   |
| C405TRCTRACESTATUS[2]         | Output | TS5       | Input | 18                  | 36                   |
| C405TRCTRACESTATUS[3]         | Output | TS6       | Input | 19                  | 38                   |

## Signal Summary

Table B-1 lists the PPC405x3 interface signals in alphabetical order. A cross reference is provided to each signal description. The signal naming conventions used are described on page 932.

Table B-1: PPC405x3 Interface Signals in Alphabetical Order

| Signal               | I/O Type | Interface        | If Unused                         | Function                                                                                        |
|----------------------|----------|------------------|-----------------------------------|-------------------------------------------------------------------------------------------------|
| C405CPMCORESLEEPREQ  | O        | CPM, page 935    | No Connect                        | Indicates the core is requesting to be put into sleep mode.                                     |
| C405CPMMSRCE         | O        | CPM, page 935    | No Connect                        | Indicates the value of MSR[CE].                                                                 |
| C405CPMMSREE         | O        | CPM, page 935    | No Connect                        | Indicates the value of MSR[EE].                                                                 |
| C405CPMTIMERIRQ      | O        | CPM, page 935    | No Connect                        | Indicates a timer-interrupt request occurred.                                                   |
| C405CPMTIMERRESETREQ | O        | CPM, page 935    | No Connect                        | Indicates a watchdog-timer reset request occurred.                                              |
| C405DBGMSRWE         | O        | Debug, page 1013 | No Connect                        | Indicates the value of MSR[WE].                                                                 |
| C405DBGSTOPACK       | O        | Debug, page 1013 | No Connect                        | Indicates the PPC405x3 is in debug halt mode.                                                   |
| C405DBGWBCOMPLETE    | O        | Debug, page 1012 | No Connect                        | Indicates the current instruction in the PPC405x3 writeback pipeline stage is completing.       |
| C405DBGWBFULL        | O        | Debug, page 1012 | No Connect                        | Indicates the PPC405x3 writeback pipeline stage is full.                                        |
| C405DBGWBIAR[0:29]   | O        | Debug, page 1012 | No Connect                        | The address of the current instruction in the PPC405x3 writeback pipeline stage.                |
| C405DCRABUS[0:9]     | O        | DCR, page 998    | No Connect                        | Specifies the address of the DCR access request.                                                |
| C405DCRDBUSOUT[0:31] | O        | DCR, page 998    | No Connect or attach to input bus | The 32-bit DCR write-data bus.                                                                  |
| C405DCRREAD          | O        | DCR, page 997    | No Connect                        | Indicates a DCR read request occurred.                                                          |
| C405DCRWRITE         | O        | DCR, page 997    | No Connect                        | Indicates a DCR write request occurred.                                                         |
| C405JTG CAPTUREDR    | O        | JTAG, page 1009  | No Connect                        | Indicates the TAP controller is in the capture-DR state.                                        |
| C405JTGEXTEST        | O        | JTAG, page 1009  | No Connect                        | Indicates the JTAG EXTEST instruction is selected.                                              |
| C405JTGPGMOUT        | O        | JTAG, page 1010  | No Connect                        | Indicates the state of a general purpose program bit in the JTAG debug control register (JDCR). |
| C405JTGSIFTDR        | O        | JTAG, page 1009  | No Connect                        | Indicates the TAP controller is in the shift-DR state.                                          |
| C405JTGTDO           | O        | JTAG, page 1009  | No Connect                        | JTAG TDO (test-data out).                                                                       |

Table B-1: PPC405x3 Interface Signals in Alphabetical Order (Continued)

| Signal                  | I/O Type | Interface                       | If Unused  | Function                                                                                                                |
|-------------------------|----------|---------------------------------|------------|-------------------------------------------------------------------------------------------------------------------------|
| C405JTGTDOEN            | O        | JTAG, <a href="#">page 1009</a> | No Connect | Indicates the JTAG TDO signal is enabled.                                                                               |
| C405JTGUPDATEDR         | O        | JTAG, <a href="#">page 1010</a> | No Connect | Indicates the TAP controller is in the update-DR state.                                                                 |
| C405PLBDCUABORT         | O        | DSPLB, <a href="#">page 973</a> | No Connect | Indicates the DCU is aborting an unacknowledged data-access request.                                                    |
| C405PLBDCUABUS[0:31]    | O        | DSPLB, <a href="#">page 970</a> | No Connect | Specifies the memory address of the data-access request.                                                                |
| C405PLBDCUBE[0:7]       | O        | DSPLB, <a href="#">page 972</a> | No Connect | Specifies which bytes are transferred during single-word transfers.                                                     |
| C405PLBDCUCACHEABLE     | O        | DSPLB, <a href="#">page 970</a> | No Connect | Indicates the value of the cacheability storage attribute for the target address.                                       |
| C405PLBDCUGUARDED       | O        | DSPLB, <a href="#">page 971</a> | No Connect | Indicates the value of the guarded storage attribute for the target address.                                            |
| C405PLBDCUPRIORITY[0:1] | O        | DSPLB, <a href="#">page 973</a> | No Connect | Indicates the priority of the data-access request.                                                                      |
| C405PLBDCUREQUEST       | O        | DSPLB, <a href="#">page 969</a> | No Connect | Indicates the DCU is making a data-access request.                                                                      |
| C405PLBDCURNW           | O        | DSPLB, <a href="#">page 970</a> | No Connect | Specifies whether the data-access request is a read or a write.                                                         |
| C405PLBDCUSIZE2         | O        | DSPLB, <a href="#">page 970</a> | No Connect | Specifies a single word or eight-word transfer size.                                                                    |
| C405PLBDCUU0ATTR        | O        | DSPLB, <a href="#">page 971</a> | No Connect | Indicates the value of the user-defined storage attribute for the target address.                                       |
| C405PLBDCUWRDBUS[0:63]  | O        | DSPLB, <a href="#">page 974</a> | No Connect | The DCU write-data bus used to transfer data from the DCU to the PLB slave.                                             |
| C405PLBDCUWRITETHRU     | O        | DSPLB, <a href="#">page 971</a> | No Connect | Indicates the value of the write-through storage attribute for the target address.                                      |
| C405PLBICUABORT         | O        | ISPLB, <a href="#">page 949</a> | No Connect | Indicates the ICU is aborting an unacknowledged fetch request.                                                          |
| C405PLBICUABUS[0:29]    | O        | ISPLB, <a href="#">page 947</a> | No Connect | Specifies the memory address of the instruction-fetch request. Bits 30:31 of the 32-bit address are assumed to be zero. |
| C405PLBICUCACHEABLE     | O        | ISPLB, <a href="#">page 948</a> | No Connect | Indicates the value of the cacheability storage attribute for the target address.                                       |
| C405PLBICUPRIORITY[0:1] | O        | ISPLB, <a href="#">page 949</a> | No Connect | Indicates the priority of the ICU fetch request.                                                                        |
| C405PLBICUREQUEST       | O        | ISPLB, <a href="#">page 947</a> | No Connect | Indicates the ICU is making an instruction-fetch request.                                                               |
| C405PLBICUSIZE[2:3]     | O        | ISPLB, <a href="#">page 947</a> | No Connect | Specifies a four word or eight word line-transfer size.                                                                 |
| C405PLBICUU0ATTR        | O        | ISPLB, <a href="#">page 948</a> | No Connect | Indicates the value of the user-defined storage attribute for the target address.                                       |
| C405RSTCHIPRESETREQ     | O        | Reset, <a href="#">page 941</a> | Required   | Indicates a chip-reset request occurred.                                                                                |
| C405RSTCORERESETREQ     | O        | Reset, <a href="#">page 940</a> | Required   | Indicates a core-reset request occurred.                                                                                |

**Table B-1: PPC405x3 Interface Signals in Alphabetical Order (Continued)**

| Signal                          | I/O Type | Interface                         | If Unused                           | Function                                                                                    |
|---------------------------------|----------|-----------------------------------|-------------------------------------|---------------------------------------------------------------------------------------------|
| C405RSTSYSRESETREQ              | O        | Reset, <a href="#">page 941</a>   | Required                            | Indicates a system-reset request occurred.                                                  |
| C405TRCCYCLE                    | O        | Trace, <a href="#">page 1015</a>  | No Connect                          | Specifies the trace cycle.                                                                  |
| C405TRCEVENEXECUTIONSTATUS[0:1] | O        | Trace, <a href="#">page 1016</a>  | No Connect                          | Specifies the execution status collected during the first of two processor cycles.          |
| C405TRCODDEXECUTIONSTATUS[0:1]  | O        | Trace, <a href="#">page 1016</a>  | No Connect                          | Specifies the execution status collected during the second of two processor cycles.         |
| C405TRCTRACESTATUS[0:3]         | O        | Trace, <a href="#">page 1016</a>  | No Connect                          | Specifies the trace status.                                                                 |
| C405TRCTRIGGEREVENTOUT          | O        | Trace, <a href="#">page 1015</a>  | Wrap to Trigger Event In            | Indicates a trigger event occurred.                                                         |
| C405TRCTRIGGEREVENTTYPE[0:10]   | O        | Trace, <a href="#">page 1015</a>  | No Connect                          | Specifies which debug event caused the trigger event.                                       |
| C405XXMACHINECHECK              | O        | Control, <a href="#">page 938</a> | No Connect                          | Indicates a machine-check error has been detected by the PPC405x3.                          |
| CPMC405CLOCK                    | I        | CPM, <a href="#">page 934</a>     | Required                            | PPC405x3 clock input (for all non-JTAG logic, including timers).                            |
| CPMC405CORECLKINACTIVE          | I        | CPM, <a href="#">page 934</a>     | 0                                   | Indicates the CPM logic disabled the clocks to the core.                                    |
| CPMC405CPUCLKEN                 | I        | CPM, <a href="#">page 934</a>     | 1                                   | Enables the core clock zone.                                                                |
| CPMC405JTAGCLKEN                | I        | CPM, <a href="#">page 934</a>     | 1                                   | Enables the JTAG clock zone.                                                                |
| CPMC405TIMERCLKEN               | I        | CPM, <a href="#">page 934</a>     | 1                                   | Enables the timer clock zone.                                                               |
| CPMC405TIMERTICK                | I        | CPM, <a href="#">page 935</a>     | 1                                   | Increments or decrements the PPC405x3 timers every time it is active with the CPMC405CLOCK. |
| DBG405DEBUGHALT                 | I        | Debug, <a href="#">page 1012</a>  | 0                                   | Indicates the external debug logic is placing the processor in debug halt mode.             |
| DBG405EXTBUSHOLDACK             | I        | Debug, <a href="#">page 1011</a>  | 0                                   | Indicates the bus controller has given control of the bus to an external master.            |
| DBG405UNCONDDEBUGEVENT          | I        | Debug, <a href="#">page 1012</a>  | 0                                   | Indicates the external debug logic is causing an unconditional debug event.                 |
| DCRC405ACK                      | I        | DCR, <a href="#">page 998</a>     | 0                                   | Indicates a DCR access has been completed by a peripheral.                                  |
| DCRC405DBUSIN[0:31]             | I        | DCR, <a href="#">page 998</a>     | 0x0000_0000 or attach to output bus | The 32-bit DCR read-data bus.                                                               |
| EICC405CRITINPUTIRQ             | I        | EIC, <a href="#">page 1005</a>    | 0                                   | Indicates an external critical interrupt occurred.                                          |
| EICC405EXTINPUTIRQ              | I        | EIC, <a href="#">page 1005</a>    | 0                                   | Indicates an external noncritical interrupt occurred.                                       |
| JTGC405BNDSCANTDO               | I        | JTAG, <a href="#">page 1009</a>   | 0                                   | JTAG boundary scan input from the previous boundary scan element TDO output.                |
| JTGC405TCK                      | I        | JTAG, <a href="#">page 1008</a>   | See IEEE 1149.1                     | JTAG TCK (test clock).                                                                      |
| JTGC405TDI                      | I        | JTAG, <a href="#">page 1008</a>   | 1                                   | JTAG TDI (test-data in).                                                                    |
| JTGC405TMS                      | I        | JTAG, <a href="#">page 1008</a>   | 1                                   | JTAG TMS (test-mode select).                                                                |
| JTGC405TRSTNEG                  | I        | Reset, <a href="#">page 942</a>   | Required                            | Performs a JTAG test reset (TRST).                                                          |

Table B-1: PPC405x3 Interface Signals in Alphabetical Order (Continued)

| Signal                  | I/O Type | Interface                       | If Unused             | Function                                                                                                           |
|-------------------------|----------|---------------------------------|-----------------------|--------------------------------------------------------------------------------------------------------------------|
| JTGC405TRSTNEG          | I        | JTAG, <a href="#">page 1009</a> | Required              | JTAG $\overline{\text{TRST}}$ (test reset).                                                                        |
| MCBCPUCLKEN             | I        | FPGA, <a href="#">page 1017</a> | 1                     | Indicates the PPC405x3 clock enable should follow GWE during a partial reconfiguration.                            |
| MCBJTAGEN               | I        | FPGA, <a href="#">page 1017</a> | 1                     | Indicates the JTAG clock enable should follow GWE during a partial reconfiguration.                                |
| MCBTIMEREN              | I        | FPGA, <a href="#">page 1017</a> | 1                     | Indicates the timer clock enable should follow GWE during a partial reconfiguration.                               |
| MCPPCRST                | I        | FPGA, <a href="#">page 1017</a> | 1                     | Indicates the PPC405x3 should be reset when GSR is asserted during a partial reconfiguration.                      |
| PLBC405DCUADDRACK       | I        | DSPLB, <a href="#">page 975</a> | 0                     | Indicates a PLB slave acknowledges the current data-access request.                                                |
| PLBC405DCUBUSY          | I        | DSPLB, <a href="#">page 978</a> | 0                     | Indicates the PLB slave is busy performing an operation requested by the DCU.                                      |
| PLBC405DCUERR           | I        | DSPLB, <a href="#">page 979</a> | 0                     | Indicates an error was detected by the PLB slave during the transfer of data to or from the DCU.                   |
| PLBC405DCURDDACK        | I        | DSPLB, <a href="#">page 977</a> | 0                     | Indicates the DCU read-data bus contains valid data for transfer to the DCU.                                       |
| PLBC405DCURDDBUS[0:63]  | I        | DSPLB, <a href="#">page 977</a> | 0x0000_0000_0000_0000 | The DCU read-data bus used to transfer data from the PLB slave to the DCU.                                         |
| PLBC405DCURDWDADDR[1:3] | I        | DSPLB, <a href="#">page 977</a> | 0b000                 | Indicates which word or doubleword of an eight-word line transfer is present on the DCU read-data bus.             |
| PLBC405DCUFSIZE1        | I        | DSPLB, <a href="#">page 976</a> | 0                     | Specifies the bus width (size) of the PLB slave that accepted the request.                                         |
| PLBC405DCUWRDACK        | I        | DSPLB, <a href="#">page 978</a> | 0                     | Indicates the data on the DCU write-data bus is being accepted by the PLB slave.                                   |
| PLBC405ICUADDRACK       | I        | ISPLB, <a href="#">page 949</a> | 0                     | Indicates a PLB slave acknowledges the current ICU fetch request.                                                  |
| PLBC405ICUBUSY          | I        | ISPLB, <a href="#">page 952</a> | 0                     | Indicates the PLB slave is busy performing an operation requested by the ICU.                                      |
| PLBC405ICUERR           | I        | ISPLB, <a href="#">page 953</a> | 0                     | Indicates an error was detected by the PLB slave during the transfer of instructions to the ICU.                   |
| PLBC405ICURDDACK        | I        | ISPLB, <a href="#">page 950</a> | 0                     | Indicates the ICU read-data bus contains valid instructions for transfer to the ICU.                               |
| PLBC405ICURDDBUS[0:63]  | I        | ISPLB, <a href="#">page 951</a> | 0x0000_0000_0000_0000 | The ICU read-data bus used to transfer instructions from the PLB slave to the ICU.                                 |
| PLBC405ICURDWDADDR[1:3] | I        | ISPLB, <a href="#">page 951</a> | 0b000                 | Indicates which word or doubleword of a four-word or eight-word line transfer is present on the ICU read-data bus. |
| PLBC405ICUFSIZE1        | I        | ISPLB, <a href="#">page 950</a> | 0                     | Specifies the bus width (size) of the PLB slave that accepted the request.                                         |
| PLBCLK                  | I        | FPGA, <a href="#">page 1018</a> | Required              | PLB clock.                                                                                                         |
| RSTC405RESETCHIP        | I        | Reset, <a href="#">page 941</a> | Required              | Indicates a chip-reset occurred.                                                                                   |
| RSTC405RESETCORE        | I        | Reset, <a href="#">page 941</a> | Required              | Resets the PPC405x3 core logic, data cache, instruction cache, and the on-chip memory controller (OCM).            |

**Table B-1: PPC405x3 Interface Signals in Alphabetical Order (Continued)**

| Signal                   | I/O Type | Interface                         | If Unused                 | Function                                                                                                        |
|--------------------------|----------|-----------------------------------|---------------------------|-----------------------------------------------------------------------------------------------------------------|
| RSTC405RESETSYS          | I        | Reset, <a href="#">page 942</a>   | Required                  | Indicates a system-reset occurred. Resets the logic in the PPC405x3 JTAG unit.                                  |
| TIEC405DETERMINISTICMULT | I        | Control, <a href="#">page 937</a> | Required                  | Specifies whether all multiply operations complete in a fixed number of cycles or have an early-out capability. |
| TIEC405DISOPERANDFWD     | I        | Control, <a href="#">page 938</a> | Required                  | Disables operand forwarding for load instructions.                                                              |
| TIEC405MMUEN             | I        | Control, <a href="#">page 937</a> | Required                  | Enables the memory-management unit (MMU)                                                                        |
| TRCC405TRACEDISABLE      | I        | Trace, <a href="#">page 1016</a>  | 0                         | Disables trace collection and broadcast.                                                                        |
| TRCC405TRIGGEREVENTIN    | I        | Trace, <a href="#">page 1016</a>  | Wrap to Trigger Event Out | Indicates a trigger event occurred and that trace status is to be generated.                                    |





# Index

## A

abort  
    data-side PLB 973, 993  
    instruction-side PLB 949, 964  
address acknowledge  
    data-side PLB 975  
    instruction-side PLB 949  
address bus  
    data-side PLB 970  
    DCR 998  
    instruction-side PLB 947  
address pipelining  
    cacheable fetch 958, 959  
    cacheable reads 982  
    data 967  
    fetch requests 944  
    non-cacheable fetch 961  
    reads and writes 983, 988  
addressing modes 920

## B

big endian, definition of 921  
boundary scan 1009  
bus-interface unit 954, 980  
busy  
    data-side PLB 978  
    instruction-side PLB 952  
bypass  
    data 966  
    instruction 944  
byte enables 972

## C

cacheability  
    data-side PLB 970  
    instruction-side PLB 948  
CCR0  
    fetch without allocate 944, 948  
    load without allocate 966  
    load word as line 966  
    non-cacheable request size 944, 951  
    store without allocate 966  
chip reset 939, 941  
    request 941  
clock  
    PLB 1018  
    PPC405 934  
clock and power management  
    See CPM interface.  
clock zone 933  
condition register  
    See CR.

core clock zone 933, 934  
core reset 939, 941  
    request 940  
core-configuration register  
    See CCR0.  
CPM interface 933  
    signals 933  
CPU control  
    interface 937  
CR 923  
critical interrupt request 1005

## D

data registers, JTAG 1006  
data-cache unit  
    See DCU.  
data-side PLB interface 965  
    See also read request.  
    See also write request.  
    abort 973  
    address acknowledge 975  
    address bus 970  
    busy 978  
    byte enables 972  
    cacheability 970  
    error 979  
    guarded storage 971  
    priority 973  
    read acknowledge 977  
    read not write 970  
    read-data bus 977  
    request 969  
    signals 967  
    slave size 976  
    timing diagrams 980  
    transfer order 977  
    transfer size 970  
    U0 attribute 971  
    write acknowledge 978  
    write-data bus 974  
    write-through 971

DCR interface 923, 994  
    address bus 998  
    chain implementation 994  
    description of 927  
    read request 997  
    read-data bus 998  
    request acknowledge 998  
    signals 996  
    write request 997  
    write-data bus 998

### DCU

    description of 926  
    fill buffer 966  
debug halt mode 1012  
debug interface 1011  
    bus hold acknowledge 1011

debug halt 1012  
debug halt acknowledge 1013  
signals 1011  
unconditional debug event 1012  
wait-state enable 1013  
writeback complete 1012  
writeback full 1012  
writeback instruction address 1012

debug modes 927

device-control register

    See DCR interface.

DSPLB

    See data-side PLB.

## E

EIC interface 1004  
    signals 1004  
error  
    data-side PLB 979  
    instruction-side PLB 953  
exceptions  
    critical 925, 1004  
    noncritical 925, 1004  
external interrupt controller  
    See EIC interface.

## F

fetch request 943  
    address pipelining 944  
    cacheable 944  
    non-cacheable request size 944  
    prefetching 944  
    without allocate 944  
FIT  
    description of 927  
    timer exception 935  
    update frequency 935  
fixed-interval timer  
    See FIT.

## G

general-purpose register  
    See GPR.  
global clock gating 933  
global local clock enables 933  
global set reset 1017  
global write enable  
    effect on core clock zone 1017  
    effect on JTAG clock zone 1017  
    effect on timer clock zone 1017  
GPR 922, 924  
guarded storage

data 967  
data-side PLB 971  
instruction 945

## I

### ICU

description of 926  
fill buffer 944  
line buffer 926

instruction register, JTAG 1006

instruction-cache unit

See ICU.

instruction-side PLB interface 943

See also fetch request.  
abort 949  
address acknowledge 949  
address bus 947  
busy 952  
cacheability 948  
error 953  
fetch request 943, 947  
priority 949  
read acknowledge 950  
read-data bus 951  
signals 945  
slave size 950  
timing diagrams 954  
transfer order 951  
transfer size 947  
U0 attribute 948

interfaces

CPM 933  
CPU control 937  
data-side PLB 965  
DCR 994  
debug 1011  
EIC 1004  
instruction-side PLB 943  
JTAG 1006  
trace 1014

ISPLB

See instruction-side PLB.

## J

### JTAG

test clock 1006  
test-access port 1006  
test-data in 1006  
test-data out 1006  
test-mode select 1006

JTAG clock zone 933, 934

JTAG interface 1006

boundary scan 1009  
capture-DR state 1009  
debug control 1010  
external test instruction 1009  
shift-DR state 1009  
signals 1007  
test clock 1008  
test reset 942, 1009  
test-data in 1008  
test-data out 1009

test-data out enable 1009  
test-mode select 1008  
update-DR state 1010

## L

little endian, definition of 921

## M

MAC 925

early out 937

machine check 938, 953, 979

machine-state register

See MSR.

memory-management unit

See MMU.

MMU 925

enable and disable 937

most recent reset 939

MSR 923

critical-interrupt enable 935, 1004  
external-interrupt enable 935, 1004  
wait-state enable 935, 1013

multiply accumulate

See MAC.

multiply, early out 937

## N

noncritical interrupt request 1005

## O

OEA

See PowerPC.

operand forwarding, disabling 938

## P

performance summary 928

PIT

description of 926  
timer exception 935  
update frequency 935

PLB

description of 927  
priority, data-side 973  
priority, instruction-side 949

PLB clock 1018

PLB slave

aborting requests 949, 974  
attaching to 32-bit slave 951, 972  
busy 952, 978  
detecting errors 953, 979

power-on reset 939

PowerPC

architecture 915

embedded-environment  
architecture 915  
OEA 916, 917  
UISA 916  
VEA 916

PPC405 923 to 928

caches 926  
central-processing unit 924  
clock 934  
debug resources 927  
exception-handling logic 925  
external interfaces 927  
memory-management unit 925  
performance 928  
software features 918  
timers 926

prefetch 944

privileged mode, definition of 920

processor block, definition of 915

processor local bus

See PLB.

processor reset

See core reset.

programmable-interval timer

See PIT.

## R

read acknowledge

data-side PLB 977  
instruction-side PLB 950

read not write 970

read request 965

address pipelining 967  
cacheable 966  
DCR 997  
unaligned operands 967  
without allocate 966

read-data bus

data-side PLB 977  
DCR 998  
instruction-side PLB 951

real mode, definition of 920

registers

supported by PPC405 921

request

chip reset 941  
core reset 940  
critical interrupt 1005  
data-side PLB 969  
instruction-side PLB 947  
noncritical interrupt 1005  
system reset 941

reset

chip 939, 941  
core or processor 939, 940, 941  
global set reset 1017  
interface requirements 939  
system 939, 941, 942  
watchdog time-out 935

## S

signal name prefixes 932  
 signal summary 1043  
 signals  
   CPM interface 933  
   CPU control interface 937  
   data-side PLB interface 967  
   DCR interface 996  
   debug interface 1011  
   EIC interface 1004  
   instruction-side PLB interface 945  
   JTAG interface 1007  
   naming conventions 932  
   reset interface 940  
   summary 1043  
   trace interface 1014  
 slave size  
   data-side PLB 976  
   instruction-side PLB 950  
 sleep mode 934  
   request 935  
   waking 933  
 special-purpose register  
   See SPR.  
 split data bus 965  
   overlapped operations 988, 990  
 SPR 923  
 storage attributes 925  
 system reset 939, 942  
   request 941

## T

TAP controller, JTAG 1006  
 timer clock zone 933, 934  
 timer exception 935  
 TLB 925  
 trace interface 1014  
   disable 1016  
   even execution status 1016  
   odd execution status 1016  
   signals 1014  
   trace cycle 1015  
   trace status 1016  
   trigger event 1015  
   trigger event in 1016  
   trigger event type 1015  
 transfer order  
   data-side PLB 977  
   instruction-side PLB 951  
 transfer size  
   data-side PLB 970  
   instruction-side PLB 947  
 translation look-aside buffer  
   See TLB.  
 trigger events 1014

## U

U0 attribute

data-side PLB 971  
 instruction-side PLB 948

## UISA

See PowerPC.

unaligned operands 967  
 unconditional debug event 1012  
 user mode, definition of 920

## V

VEA 917

See PowerPC.

virtual mode, definition of 920

## W

watchdog timer

See WDT.

## WDT

description of 927  
 reset request 935  
 timer exception 935  
 update frequency 935  
 write acknowledge 978  
 write request 965  
   address pipelining 967  
   DCR 997  
   non-cacheable 966  
   unaligned operands 967  
   without allocate 966  
 write-data bus  
   data-side PLB 974  
   DCR 998  
 write-through cacheability 971



# **Volume 3: Rocket I/O™ Transceiver User Guide**

***Virtex-II Pro™ Platform FPGA  
Developer's Kit***

March 2002 Release





The Xilinx logo shown above is a registered trademark of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

"Xilinx" and the Xilinx logo are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, XC5210 are registered Trademarks of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Bencher, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Bencher, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, Nano-Blaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, Rocket I/O, SelectI/O, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM      IBM Logo      PowerPC      PowerPC Logo      Blue Logic      CoreConnect      CodePack

All other trademarks are the property of their respective owners.

Xilinx does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 2002 Xilinx, Inc. All Rights Reserved.

# Introduction

---

**IMPORTANT NOTE:**

This document assumes use of ISE v4.2.x. If running ISE v4.1.x, the following modifications must be made:

1. Remove the port ENMCOMMAALIGN and replace its function by adding the attribute MCOMMA\_ALIGN.
2. Remove the port ENPCOMMAALIGN and replace its function by adding the attribute PCOMMA\_ALIGN.
3. Where a High is indicated for a removed port, set the corresponding attribute to TRUE; where a Low is indicated, set the corresponding attribute to FALSE.

## Rocket I/O Features

The Rocket I/O™ transceiver's flexible, programmable features allow a multi-gigabit serial transceiver to be easily integrated into any Virtex-II Pro design:

- Variable speed full-duplex transceiver, allowing 500 Mb/s to 3.125 Gb/s baud transfer rates
- Monolithic clock synthesis and clock recovery system, eliminating the need for external components
- Automatic lock-to-reference function
- Serial output differential swing can be programmed at five levels from 800 mV to 1600 mV (peak-peak), allowing compatibility with other serial system voltage levels.
- Four levels of programmable pre-emphasis
- AC and DC coupling
- Programmable on-chip termination of 50Ω or 75Ω (eliminating the need for external termination resistors)
- Serial and parallel TX to RX internal loopback modes for testing operability
- Programmable comma detection to allow for any protocol and detection of any 10-bit character.

## In This User Guide

The Rocket I/O Transceiver User Guide contains these sections:

- **Chapter 1, Introduction** — This chapter.
- **Chapter 2, Rocket I/O™ Transceiver Overview** — An overview of the transceiver's capabilities and how it works.



- **Chapter 3, Digital Design Considerations** — Ports and attributes for the six provided communications protocol primitives; VHDL/Verilog code examples for clocking and reset schemes; transceiver instantiation; 8B/10B encoding; CRC; channel bonding.
- **Chapter 4, Analog Design Considerations** — Rocket I/O serial overview; pre-emphasis; jitter; clock/data recovery; PCB design requirements.
- **Chapter 5, Simulation and Implementation** — Simulation models; implementation tools; debugging and diagnostics.
- **Appendix A, Rocket I/O™ Cell Models** — Verilog module declarations associated with each of the sixteen Rocket I/O communication standard implementations.

## Naming Conventions

Input and output ports of the Rocket I/O transceiver primitives are denoted in upper-case letters. Attributes of the Rocket I/O transceiver are denoted in upper-case letters with underscores. Trailing numbers in primitive names denote the byte width of the data path. These values are preset and not modifiable. When assumed to be the same frequency, RXUSRCLK and TXUSRCLK are referred to as USRCLK and can be used interchangeably. This also holds true for RXUSRCLK2, TXUSRCLK2, and USRCLK2.

### Comma Definition

A comma is a “K” character used by the transceiver to align the serial data on a byte/half-word boundary (depending on the protocol used), so that the serial data is correctly decoded into parallel data.

## For More Information

For a complete menu of online information resources available on the Xilinx website, visit <http://www.xilinx.com/virtex2pro/>.

For a comprehensive listing of available tutorials and resources on network technologies and communications protocols, visit <http://www.iol.unh.edu/training/>.

## Further Reading

The Virtex-II Pro™ Developer's Kit contains a wealth of valuable information that will assist you in your design efforts. The documentation contained within the eight volumes is organized to assist you in quickly finding relevant materials. To obtain the most recent revision of this documentation, please see [http://support.xilinx.com/xlnx/xil\\_tt\\_product.jsp?sProduct=Virtex-II+Pro](http://support.xilinx.com/xlnx/xil_tt_product.jsp?sProduct=Virtex-II+Pro).

## Documentation Provided by Xilinx

*Virtex-II Pro™ Platform FPGA Handbook*

*Virtex-II Pro™ Platform FPGA Developer's Kit Documentation:*

*Volume 1, Advance Product Specification*

*Volume 2, PowerPC® 405 Processor*

*Volume 3, Rocket I/O™ Transceiver User Guide*

*Volume 4, Design Flow*

*Volume 5, Software Development Tools*

*Volume 6, Reference Systems*

*Volume 7, Hardware IP Specifications*

*Volume 8, Software IP and Applications*

## IBM® CoreConnect™ Documentation

The Virtex-II Pro Developer's Kit integrates seamlessly with the IBM CoreConnect Toolkit. This toolkit is not included with the Developer's Kit, but is required if bus functional simulation is desired. The toolkit provides a number of features which enhance design productivity and allow you to get the most from the Developer's Kit. To obtain the toolkit, you must be a licensee of the IBM CoreConnect Bus Architecture. Licensing CoreConnect provides access to a wealth of documentation, Bus Functional Models, Hardware IP, and the toolkit.

Xilinx provides a Web-based licensing mechanism that allows you to obtain the CoreConnect toolkit from our website. To license CoreConnect, use an Internet browser to access [http://www.xilinx.com/ipcenter/processor\\_central/register\\_coreconnect.htm](http://www.xilinx.com/ipcenter/processor_central/register_coreconnect.htm). Once your request has been approved (typically within 24 hours), you will receive an e-mail granting access to a protected website. You may then download the toolkit.

If you prefer, you can also license CoreConnect directly from IBM.

If you would like further information on CoreConnect Bus Architecture, please see IBM's CoreConnect website at <http://www.ibm.com/chips/products/coreconnect>.

Once you have licensed the CoreConnect toolkit, and installed it with the Developer's Kit, the following documents will be available to you in the following locations:

### IBM CoreConnect Bus Architecture Specifications

*IBM CoreConnect Processor Local Bus (PLB) Architecture Specification*  
see \$CORECONNECT/published/corecon/64bitPlbBus.pdf

*IBM CoreConnect On-chip Peripheral Bus (OPB) Architecture Specification*  
see \$CORECONNECT/published/corecon/OpbBus.pdf

*IBM CoreConnect Device Control Register (DCR) Bus Architecture Specification*  
see \$CORECONNECT/published/corecon/DcrBus.pdf

### IBM CoreConnect Toolkit Documentation

*PLB Bus Functional Model Toolkit - User's Manual*  
see \$CORECONNECT/published/corecon/PlbToolkit.pdf

*OPB Bus Functional Model Toolkit - User's Manual*  
see \$CORECONNECT/published/corecon/OpbToolkit.pdf

*DCR Bus Functional Model Toolkit - User's Manual*  
see \$CORECONNECT/published/corecon/DcrToolkit.pdf

*CoreConnect Test Generator - User's Manual*  
see \$CORECONNECT/published/corecon/ctg.pdf

**Note:** \$CORECONNECT is an environment variable that is created when installing the Developer's Kit or CoreConnect Toolkit.

## Software Development Documentation

There are many sources of documentation available for those who wish to learn more about Software Development. It is recommended that a web search be conducted using a favorite search engine for keywords such as "PowerPC+Software+Development". Alternatively, a technical bookstore should be able to provide many valuable resources. The books listed below are a very small fraction of those available.

### Books About Programming in C

There are many good books about the C and C++ programming languages. A few of these are listed below:

Books about C

Kernigham Brian, Ritchie Dennis. 1988. *The C Programming Language*. Prentice Hall

Wang Paul. 1992. *An Introduction to ANSI C on Unix*. Wadsworth.

Kumar Ram, Agrawal Rakesh. 1992. *Programming in ANSI C*. West Publishing Company.

#### Books about C++

Stroustrup Bjarne. 1991. *The C++ Programming Language*. Addison-Wesley.

Lafore Robert. 1995. *Object-Oriented Programming in C++*. Waite Group Press.

#### Online Documents About Programming in C

The Internet offers plenty of documentation about how to program in C and C++. One of the best approaches to finding documentation online is to use a search engine (such as Google, <http://www.google.com>) and search on "introduction to programming in C."

## Rocket I/O™ Transceiver Overview

### Basic Architecture and Capabilities

The Rocket I/O transceiver is based on Mindspeed's SkyRail™ technology. [Figure 2-1, page 1060](#), depicts an overall block diagram of the transceiver. Up to 16 transceiver modules are available on a single Virtex-II Pro FPGA, depending on the part being used. [Table 2-1](#) shows the Rocket I/O cores available by device.

**Table 2-1: Rocket I/O Cores**

| Device  | Rocket I/O Cores |
|---------|------------------|
| XC2VP2  | 4                |
| XC2VP4  | 4                |
| XC2VP7  | 8                |
| XC2VP20 | 8                |
| XC2VP50 | 16               |

The transceiver module is designed to operate at any serial bit rate in the range of 500 Mb/s to 3.125 Gb/s per channel, including the specific bit rates used by the communications standards listed in [Table 2-2](#). The serial bit rate need not be configured in the transceiver, as the operating frequency is implied by the received data, the reference clock applied, and the SERDES\_10B attribute ([Table 2-3, page 1060](#)).

**Table 2-2: Communications Standards Supported by Rocket I/O Transceiver**

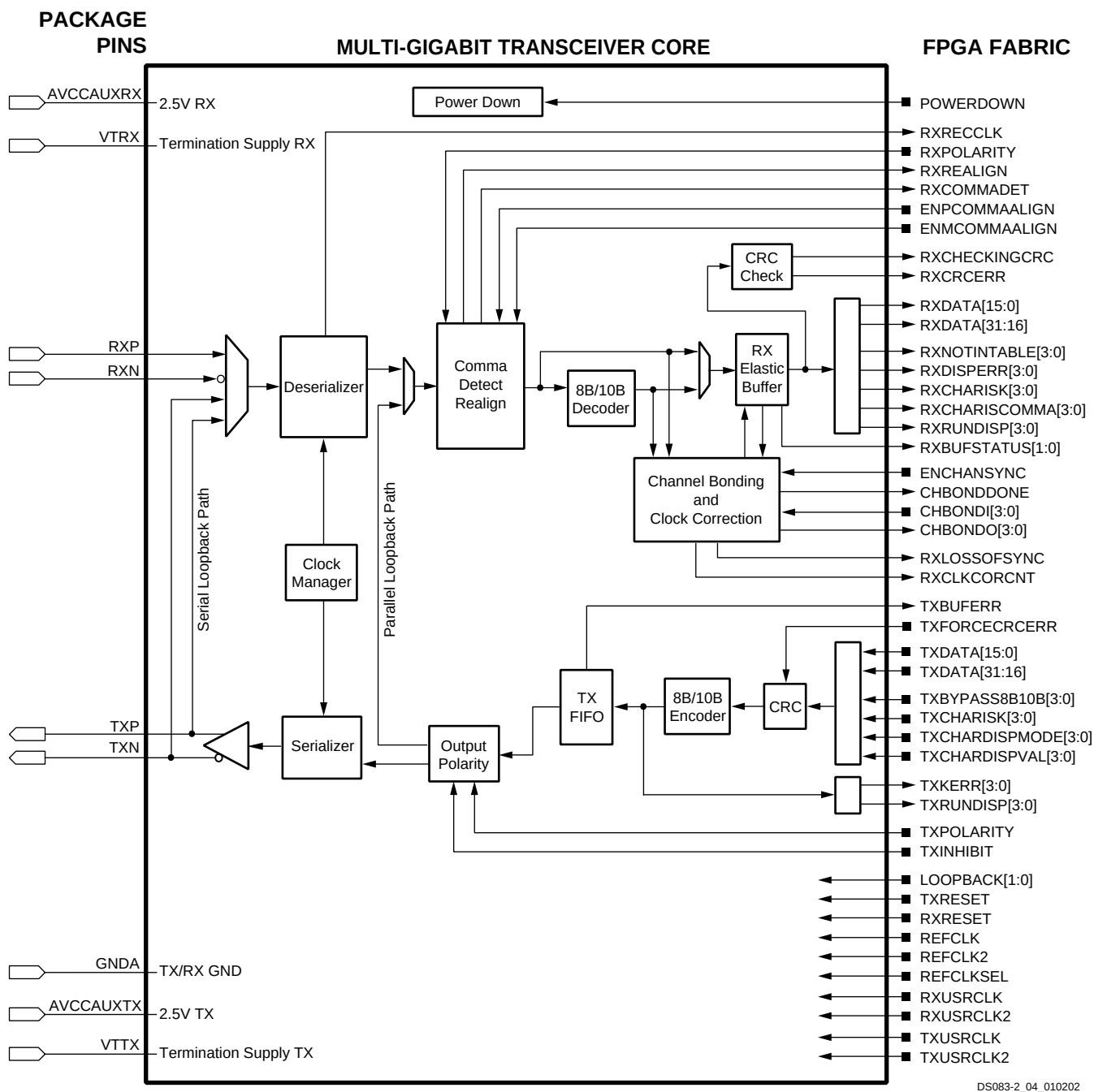
| Mode                     | Channels (Lanes) <sup>(1)</sup> | I/O Bit Rate (Gb/s) |
|--------------------------|---------------------------------|---------------------|
| Fibre Channel            | 1                               | 1.06                |
|                          |                                 | 2.12                |
| Gbit Ethernet            | 1                               | 1.25                |
| XAUI (10-Gbit Ethernet)  | 4                               | 3.125               |
| Infiniband               | 1, 4, 12                        | 2.5                 |
| Aurora (Xilinx protocol) | 1, 2, 3, 4, ...                 | 0.5 – 3.125         |
| Custom Mode              | 1, 2, 3, 4, ...                 | 0.5 – 3.125         |

**Notes:**

1. One channel is considered to be one transceiver.

Table 2-3: Serial Baud Rates and the SERDES\_10B Attribute

| SERDES_10B | Serial Baud Rate      |
|------------|-----------------------|
| False      | 800 Mb/s – 3.125 Gb/s |
| True       | 500 Mb/s – 1.0 Gb/s   |



DS083-2\_04\_010202

Figure 2-1: Rocket I/O Transceiver Block Diagram

**Table 2-4** lists the 16 gigabit transceiver primitives provided. These primitives carry attributes set to default values for the communications protocols listed in **Table 2-2**. Data widths of one, two, and four bytes are selectable for each protocol.

**Table 2-4: Supported Rocket I/O Transceiver Primitives**

| Primitives      | Description                        | Primitive       | Description                       |
|-----------------|------------------------------------|-----------------|-----------------------------------|
| GT_CUSTOM       | Fully customizable by user         | GT_XAUI_2       | 10-Gb Ethernet, 2-byte data path  |
| GT_FIBRE_CHAN_1 | Fibre Channel, 1-byte data path    | GT_XAUI_4       | 10-Gb Ethernet, 4-byte data path  |
| GT_FIBRE_CHAN_2 | Fibre Channel, 2-byte data path    | GT_INFINIBAND_1 | Infiniband, 1-byte data path      |
| GT_FIBRE_CHAN_4 | Fibre Channel, 4-byte data path    | GT_INFINIBAND_2 | Infiniband, 2-byte data path      |
| GT_ETHERNET_1   | Gigabit Ethernet, 1-byte data path | GT_INFINIBAND_4 | Infiniband, 4-byte data path      |
| GT_ETHERNET_2   | Gigabit Ethernet, 2-byte data path | GT_AURORA_1     | Xilinx protocol, 1-byte data path |
| GT_ETHERNET_4   | Gigabit Ethernet, 4-byte data path | GT_AURORA_2     | Xilinx protocol, 2-byte data path |
| GT_XAUI_1       | 10-Gb Ethernet, 1-byte data path   | GT_AURORA_4     | Xilinx protocol, 4-byte data path |

There are two ways to modify the Rocket I/O transceiver:

- Static properties can be set through attributes in the HDL code. Use of attributes are covered in detail in **Primitive Attributes**, [page 1073](#).
- Dynamic changes can be made by the ports of the primitives

The Rocket I/O transceiver consists of the Physical Media Attachment (PMA) and Physical Coding Sublayer (PCS). The PMA contains the serializer/deserializer (SERDES), TX and RX buffers, clock generator, and clock recovery circuitry. The PCS contains the 8B/10B encoder/decoder and the elastic buffer supporting channel bonding and clock correction. The PCS also handles Cyclic Redundancy Check (CRC). Refer again to **Figure 2-1**, showing the Rocket I/O transceiver top-level block diagram and FPGA interface signals.

## Clock Synthesizer

Synchronous serial data reception is facilitated by a clock/data recovery circuit. This circuit uses a fully monolithic Phase-Locked Loop (PLL), which does not require any external components. The clock/data recovery circuit extracts both phase and frequency from the incoming data stream. The recovered clock is presented on output RXRECCLK at 1/20 of the serial received data rate.

The gigabit transceiver multiplies the reference frequency provided on the reference clock input (REFCLK) by 20.

No fixed phase relationship is assumed between REFCLK, RXRECCLK, and/or any other clock that is not tied to either of these clocks. When the 4-byte or 1-byte receiver data path is used, RXUSRCLK and RXUSRCLK2 have different frequencies (1:2), and each edge of the slower clock is aligned to a falling edge of the faster clock. The same relationships apply to TXUSRCLK and TXUSRCLK2. See the section entitled **Clocking**, [page 1082](#), for details.

## Clock and Data Recovery

The clock/data recovery (CDR) circuits lock to the reference clock automatically if the data is not present. For proper operation, frequency variations of REFCLK, TXUSRCLK, RXUSRCLK, and the incoming stream (RXRECCLK) must not exceed  $\pm 100$  ppm.

It is critical to keep power supply noise low in order to minimize common and differential noise modes into the clock/data recovery circuitry. See **PCB Design Requirements**, page 1125, for more details.

## Transmitter

### FPGA Transmit Interface

The FPGA can send either one, two, or four characters of data to the transmitter. Each character can be either 8 bits or 10 bits wide. If 8-bit data is applied, the additional inputs become control signals for the 8B/10B encoder. When the 8B/10B encoder is bypassed, the 10-bit character order is:

```
TXCHARDISPMODE[0]
TXCHARDISPVAL[0]
TXDATA[7:0]
```

### 8B/10B Encoder

A bypassable 8B/10B encoder is included. The encoder uses the same 256 data characters and 12 control characters that are used for Gigabit Ethernet, XAUI, Fibre Channel, and InfiniBand.

The encoder accepts 8 bits of data along with a K-character signal for a total of 9 bits per character applied. If the K-character signal is High, the data is encoded into one of the 12 possible K-characters available in the 8B/10B code. If the K-character input is Low, the 8 bits are encoded as standard data. If the K-character input is High, and a user applies other than one of the 12 possible combinations, TXKERR indicates the error.

### Disparity Control

The 8B/10B encoder is initialized with a negative running disparity.

TXRUNDISP signals the transmitter's current running disparity.

Bits TXCHARDISPMODE and TXCHARDISPVAL control the generation of running disparity before each byte, as shown in **Table 2-5**.

**Table 2-5: Running Disparity Control**

| {txchardispmode,<br>txchardispval} | Function                                                              |
|------------------------------------|-----------------------------------------------------------------------|
| 00                                 | Maintain running disparity normally                                   |
| 01                                 | Invert normally generated running disparity before encoding this byte |
| 10                                 | Set negative running disparity before encoding this byte              |
| 11                                 | Set positive running disparity before encoding this byte              |

For example, the transceiver can generate the sequence

K28.5+ K28.5+ K28.5– K28.5–

or

K28.5– K28.5– K28.5+ K28.5+

by specifying inverted running disparity for the second and fourth bytes.

## Transmit FIFO

Proper operation of the circuit is only possible if the FPGA clock (TXUSRCLK) is frequency-locked to the reference clock (REFCLK). Phase variations up to one clock cycle are allowable. The FIFO has a depth of four. Overflow or underflow conditions are detected and signaled at the interface.

## Serializer

The multi-gigabit transceiver multiplies the reference frequency provided on the reference clock input (REFCLK) by 20. Data is converted from parallel to serial format and transmitted on the TXP and TXN differential outputs. Bit 0 is transmitted first and bit 19 is transmitted last.

The electrical polarity of TXP and TXN can be interchanged through the TXPOLARITY port. This option can either be programmed or controlled by an input at the FPGA core TX interface. This facilitates recovery from situations where printed circuit board traces have been reversed.

## Transmit Termination

On-chip termination is provided at the transmitter, eliminating the need for external termination. Programmable options exist for 50Ω (default) and 75Ω termination.

## Pre-emphasis Circuit and Swing Control

Four selectable levels of pre-emphasis, including default pre-emphasis, are available. Optimizing this setting allows the transceiver to drive up to 20 inches of FR4 at the maximum baud rate.

The programmable output swing control can adjust the differential output level between 400 mV and 800 mV (peak-to-peak) in four increments of 100 mV.

# Receiver

## Deserializer

The Rocket I/O transceiver core accepts serial differential data on its RXP and RXN inputs. The clock/data recovery circuit extracts clock phase and frequency from the incoming data stream and re-times incoming data to this clock. The recovered clock is presented on output RXRECCLK at 1/20 of the received serial data rate.

The receiver is capable of handling either transition-rich 8B/10B streams or scrambled streams, and can withstand a string of up to 75 non-transitioning bits without an error.

Word alignment is dependent on the state of comma detect bits. If comma detect is enabled, the transceiver recognizes up to two 10-bit preprogrammed characters. Upon detection of the character or characters, the comma detect output is driven High and the data is synchronously aligned. If a comma is detected and the data is aligned, no further alignment alteration takes place. If a comma is received and realignment is necessary, the data is realigned and an indication is given at the RX FPGA interface. The realignment indicator is a distinct output. The transceiver continuously monitors the data for the presence of the 10-bit character(s). Upon each occurrence of the 10-bit character, the data is checked for word alignment. If comma detect is disabled, the data is not aligned to any particular pattern. The programmable option allows a user to align data on comma+, comma-, both, or a unique user-defined and programmed sequence.

The electrical polarity of RXP and RXN can be interchanged through the RXPOLARITY port. This can be useful in the event that printed circuit board traces have been reversed.



## Receiver Termination

On-chip termination is provided at the receiver, eliminating the need for external termination. The receiver includes programmable on-chip termination circuitry for 50Ω (default) or 75Ω impedance.

## 8B/10B Decoder

An optional 8B/10B decoder is included. A programmable option allows the decoder to be bypassed. (See **HDL Code Examples: Transceiver Bypassing of 8B/10B Encoding**, [page 1113](#).) When the 8B/10B decoder is bypassed, the 10-bit character order is:

```
RXCHARISK[0]
RXRUNDISP[0]
RXDATA[7:0]
```

The decoder uses the same table that is used for Gigabit Ethernet, Fibre Channel and InfiniBand. In addition to decoding all data and K-characters, the decoder has several extra features. The decoder separately detects both “disparity errors” and “out-of-band” errors. A *disparity error* occurs when a 10-bit character is received that exists within the 8B/10B table, but has an incorrect disparity. An *out-of-band error* occurs when a 10-bit character is received that does not exist within the 8B/10B table. It is possible to obtain an out-of-band error without having a disparity error. The proper disparity is always computed for both legal and illegal characters. The current running disparity is available at the RXRUNDISP signal.

The 8B/10B decoder performs a unique operation if out-of-band data is detected. If out-of-band data is detected, the decoder signals the error and passes the illegal 10-bits through and places them on the outputs. This can be used for debugging purposes if desired.

The decoder also signals reception of one of the 12 valid K-characters. In addition, a programmable comma detect is included. The comma detect signal registers a comma on the receipt of any comma+, comma–, or both. Since the comma is defined as a 7-bit character, this includes several out-of-band characters. Another option allows the decoder to detect only the three defined commas (K28.1, K28.5, and K28.7) as comma+, comma–, or both. In total, there are six possible options, three for valid commas and three for “any comma”.

Note that all bytes (1, 2, or 4) at the RX FPGA interface each have their own individual 8B/10B indicators (K-character, disparity error, out-of-band error, current running disparity, and comma detect).

## Loopback

To facilitate testing without having the need to either apply patterns or measure data at GHz rates, two programmable loopback features are available.

One option, serial loopback, places the gigabit transceiver into a state where transmit data is directly fed back to the receiver. An important point to note is that the feedback path is at the output pads of the transmitter. This tests the entirety of the transmitter and receiver.

The second loopback path is a parallel path that checks the digital circuitry. When the parallel option is enabled, the serial loopback path is disabled. However, the transmitter outputs remain active and data is transmitted over a link. If TXINHIBIT is asserted, TXN is forced to 1 and TXP is forced to 0 until TXINHIBIT is de-asserted.

The two loopback options are shown in [Table 2-6](#).

Table 2-6: Loopback Options

| LOOPBACK[1:0] | Description                |
|---------------|----------------------------|
| LOOPBACK[1]   | External serial loopback   |
| LOOPBACK[0]   | Internal parallel loopback |

## Elastic and Transmitter Buffers

Both the transmitter and the receiver include buffers (FIFOs) in the data path. This section gives the reasons for including the buffers and outlines their operation.

### Receiver Buffer

The receiver buffer is required for two reasons:

- To accommodate the slight difference in frequency between the recovered clock RXRECCLK and the internal FPGA core clock RXUSRCLK (clock correction)
- To allow realignment of the input stream to ensure proper alignment of data being read through multiple transceivers (channel bonding)

The receiver uses an *elastic buffer*, where "elastic" refers to the ability to modify the read pointer for clock correction and channel bonding.

### Clock Correction

Clock RXRECCLK (the recovered clock) reflects the data rate of the incoming data. Clock RXUSRCLK defines the rate at which the FPGA core consumes the data. Ideally, these rates are identical. However, since the clocks typically have different sources, one of the clocks is faster than the other. The receiver buffer accommodates this difference between the clock rates. See [Figure 2-2](#).

Nominally, the buffer is always half full. This is shown in the top buffer, [Figure 2-2](#), where the shaded area represents buffered data not yet read. Received data is inserted via the write pointer under control of RXRECCLK. The FPGA core reads data via the read pointer under control of RXUSRCLK. The half full/half empty condition of the buffer gives a cushion for the differing clock rates. This operation continues indefinitely, regardless of whether or not "meaningful" data is being received. When there is no meaningful data to be received, the incoming data consists of IDLE characters or other padding.

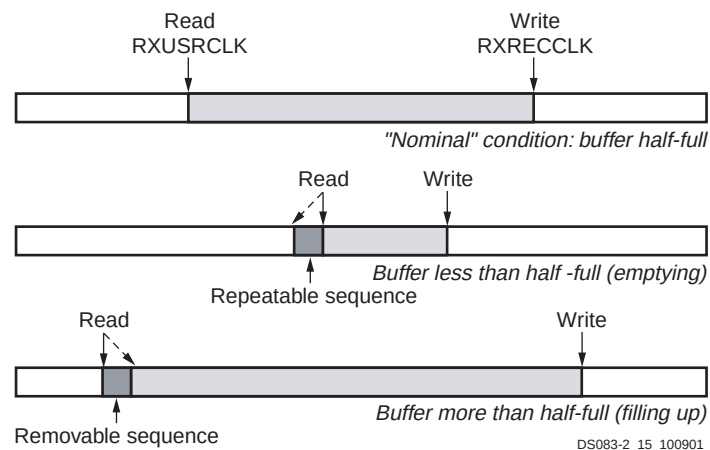


Figure 2-2: Clock Correction in Receiver

If RXUSRCLK is faster than RXRECCLK, the buffer becomes more empty over time. The clock correction logic corrects for this by decrementing the read pointer to reread a repeatable byte sequence. This is shown in the middle buffer, [Figure 2-2](#), where the solid read pointer decrements to the value represented by the dashed pointer. By decrementing the read pointer instead of incrementing it in the usual fashion, the buffer is partially refilled. The transceiver inserts a single repeatable byte sequence when necessary to refill a buffer. If the byte sequence length is greater than one, and if attribute CLK\_COR\_REPEAT\_WAIT is 0, then the transceiver can repeat the same sequence multiple times until the buffer is refilled to the half-full condition.

Similarly, if RXUSRCLK is slower than RXRECCLK, the buffer fills up over time. The clock correction logic corrects for this by incrementing the read pointer to skip over a removable byte sequence that need not appear in the final FPGA core byte stream. This is shown in the bottom buffer, **Figure 2-2**, where the solid read pointer increments to the value represented by the dashed pointer. This accelerates the emptying of the buffer, preventing its overflow. The transceiver design skips a single byte sequence, when necessary, to partially empty a buffer. If attribute CLK\_COR\_REPEAT\_WAIT is 0, the transceiver can also skip two consecutive removable byte sequences in one step, to further empty the buffer, when necessary.

These operations require the clock correction logic to recognize a byte sequence that can be freely repeated or omitted in the incoming data stream. This sequence is generally an IDLE sequence, or other sequence comprised of special values that occur in the gaps separating packets of meaningful data. These gaps are required to occur sufficiently often to facilitate the timely execution of clock correction.

## Channel Bonding

Some gigabit I/O standards such as Infiniband specify the use of multiple transceivers in parallel for even higher data rates. Words of data are split into bytes, with each byte sent over a separate channel (transceiver). See **Figure 2-3**.

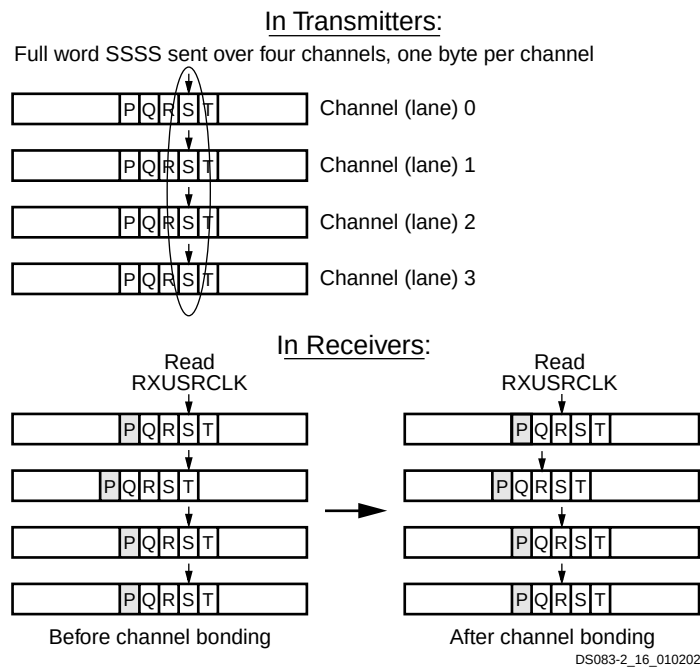


**Figure 2-3: Channel Bonding (Alignment)**

The top half of the figure shows the transmission of words split across four transceivers (channels or lanes). PPPP, QQQQ, RRRR, SSSS, and TTTT represent words sent over the four channels.

The bottom-left portion of the figure shows the initial situation in the FPGA's receivers at the other end of the four channels. Due to variations in transmission delay—especially if the channels are routed through repeaters—the FPGA core might not correctly assemble the bytes into complete words. The bottom-left illustration shows the incorrect assembly of data words PQPP, QRQQ, RSRR, etc.

To support correction of this misalignment, the data stream includes special byte sequences that define corresponding points in the several channels. In the bottom half of **Figure 2-3**, the shaded "P" bytes represent these special characters. Each receiver

recognizes the "P" channel bonding character, and remembers its location in the buffer. At some point, one transceiver designated as the master instructs all the transceivers to align to the channel bonding character "P" (or to some location relative to the channel bonding character). After this operation, the words transmitted to the FPGA core are properly aligned: RRRR, SSSS, TTTT, etc., as shown in the bottom-right portion of [Figure 2-3](#). To ensure that the channels remain properly aligned following the channel bonding operation, the master transceiver must also control the clock correction operations described in the previous section for all channel-bonded transceivers.

## Transmitter Buffer

The transmitter buffer's write pointer (TXUSRCLK) is frequency-locked to its read pointer (REFCLK). Therefore, clock correction and channel bonding are not required. The purpose of the transmitter's buffer is to accommodate a phase difference between TXUSRCLK and REFCLK. A simple FIFO suffices for this purpose. A FIFO depth of four permits reliable operation with simple detection of overflow or underflow, which might occur if the clocks are not frequency-locked.

## CRC

The Rocket I/O transceiver CRC logic supports the 32-bit invariant CRC calculation used by Infiniband, FibreChannel, and Gigabit Ethernet.

On the transmitter side, the CRC logic recognizes where the CRC bytes should be inserted and replaces four placeholder bytes at the tail of a data packet with the computed CRC. For Gigabit Ethernet and FibreChannel, transmitter CRC can adjust certain trailing bytes to generate the required running disparity at the end of the packet.

On the receiver side, the CRC logic verifies the received CRC value, supporting the same standards as above.

The CRC logic also supports a user mode, with a simple data packet structure beginning and ending with user-defined SOP and EOP characters.

There are limitations to the CRC support provided by the Rocket I/O transceiver core:

- It is for single-channel use only. Computation and byte-striping of CRC across multiple bonded channels is not supported. For that usage, the CRC logic can be implemented in the FPGA fabric.
- The Rocket I/O transceiver does not compute the 16-bit variant CRC used for Infiniband. Therefore, Rocket I/O CRC does not fulfill the Infiniband CRC requirement. Infiniband CRC can be computed in the FPGA fabric.

## Reset/Power Down

The receiver and transmitter have their own synchronous reset inputs. The transmitter reset recenters the transmission FIFO and resets all transmitter registers and the 8B/10B encoder. The receiver reset recenters the receiver elastic buffer and resets all receiver registers and the 8B/10B decoder. Neither reset signal has any effect on the PLLs.

Additional reset and power control descriptions are given in [Table 2-7](#) and [Table 2-8](#).

**Table 2-7: Reset and Power Control Descriptions**

| Ports     | Description                                                                                                                                                                                                        |
|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RXRESET   | Synchronous receive system reset recenters the receiver elastic buffer, and resets the 8B/10B decoder, comma detect, channel bonding, clock correction logic, and other receiver registers. The PLL is unaffected. |
| TXRESET   | Synchronous transmit system reset recenters the transmission FIFO, and resets the 8B/10B encoder and other transmission registers. The PLL is unaffected.                                                          |
| POWERDOWN | Shuts down the transceiver (both RX and TX sides) and sets TXP and TXN outputs to high-impedance state                                                                                                             |

**Table 2-8: Power Control Descriptions**

| POWERDOWN | Transceiver Status                   |
|-----------|--------------------------------------|
| 0         | Transceiver in operation             |
| 1         | Transceiver temporarily powered down |

## Digital Design Considerations

### List of Available Ports

The Rocket I/O transceiver primitives contain 50 ports, with the exception of the 46-port GT\_ETHERNET and GT\_FIBRE\_CHAN primitives. The differential serial data ports (RXN, RXP, TXN, and TXP) are connected directly to external pads; the remaining 46 ports are all accessible from the FPGA logic (42 ports for GT\_ETHERNET and GT\_FIBRE\_CHAN).

Table 3-1 contains the port descriptions of all primitives.

Table 3-1: GT\_CUSTOM<sup>(1)</sup>, GT\_AURORA, GT\_FIBRE\_CHAN<sup>(2)</sup>, GT\_ETHERNET<sup>(2)</sup>, GT\_INFINIBAND, and GT\_XAUI Primitive Ports

| Port                      | I/O | Port Size <sup>(3)</sup> | Definition                                                                                                                                                         |
|---------------------------|-----|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CHBONDDONE <sup>(2)</sup> | O   | 1                        | Indicates a receiver has successfully completed channel bonding when asserted High.                                                                                |
| CHBONDI <sup>(2)</sup>    | I   | 4                        | The channel bonding control that is used only by "slaves" which is driven by a transceiver's CHBONDO port.                                                         |
| CHBONDO <sup>(2)</sup>    | O   | 4                        | Channel bonding control that passes channel bonding and clock correction control to other transceivers.                                                            |
| CONFIGENABLE              | I   | 1                        | Reconfiguration enable input (unused)                                                                                                                              |
| CONFIGIN                  | I   | 1                        | Data input for reconfiguring transceiver (unused)                                                                                                                  |
| CONFIGOUT                 | O   | 1                        | Data output for configuration readback (unused)                                                                                                                    |
| ENCHANSYNC <sup>(2)</sup> | I   | 1                        | Comes from the core to the transceiver and enables the transceiver to perform channel bonding                                                                      |
| ENMCOMMAALIGN             | I   | 1                        | Selects realignment of incoming serial bitstream on minus-comma. High realigns serial bitstream byte boundary when minus-comma is detected.                        |
| ENPCOMMAALIGN             | I   | 1                        | Selects realignment of incoming serial bitstream on plus-comma. High realigns serial bitstream byte boundary when plus-comma is detected.                          |
| LOOPBACK                  | I   | 2                        | Selects the two loopback test modes. Bit 1 is for serial loopback and bit 0 is for internal parallel loopback.                                                     |
| POWERDOWN                 | I   | 1                        | Shuts down both the receiver and transmitter sides of the transceiver when asserted High. This decreases the power consumption while the transceiver is shut down. |

Table 3-1: GT\_CUSTOM<sup>(1)</sup>, GT\_AURORA, GT\_FIBRE\_CHAN<sup>(2)</sup>, GT\_ETHERNET<sup>(2)</sup>, GT\_INFINIBAND, and GT\_XAUI Primitive Ports (Continued)

| Port               | I/O | Port Size <sup>(3)</sup> | Definition                                                                                                                                                                                                                                                                                                                                                    |
|--------------------|-----|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| REFCLK             | I   | 1                        | High-quality reference clock driving transmission (reading TX FIFO, and multiplied for parallel/serial conversion) and clock recovery. REFCLK frequency is accurate to $\pm 100$ ppm. This clock originates off the device, is routed through fabric interconnect, and is selected by the REFCLKSEL.                                                          |
| REFCLK2            | I   | 1                        | An alternative to REFCLK. Can be selected by the REFCLKSEL.                                                                                                                                                                                                                                                                                                   |
| REFCLKSEL          | I   | 1                        | Selects the reference clock to use REFCLK or REFCLK2. Deasserted is REFCLK. Asserted is REFCLK2.                                                                                                                                                                                                                                                              |
| RXBUFSTATUS        | O   | 2                        | Receiver elastic buffer status. Bit 1 indicates if an overflow /underflow error has occurred when asserted High. Bit 0 indicates if the buffer is at least half-full when asserted High.                                                                                                                                                                      |
| RXCHARISCOMMA      | O   | 1, 2, 4                  | Similar to RXCHARISK except that the data is a comma.                                                                                                                                                                                                                                                                                                         |
| RXCHARISK          | O   | 1, 2, 4                  | If 8B/10B decoding is enabled, it indicates that the received data is a "K" character when asserted High. Included in Byte-mapping. If 8B/10B decoding bypassed, it becomes the 10th bit of the 10-bit encoded data.                                                                                                                                          |
| RXCHECKINGCRC      | O   | 1                        | CRC status for the receiver. Asserts High to indicate that the receiver has recognized the end of a data packet. Only meaningful if RX_CRC_USE = TRUE.                                                                                                                                                                                                        |
| RXCLKCORCNT        | O   | 3                        | Status that denotes occurrence of clock correction or channel bonding. This status is synchronized on the incoming RXDATA. See <b>Clock Correction Count</b> , page 1099.                                                                                                                                                                                     |
| RXCOMMADET         | O   | 1                        | Signals that a comma has been detected in the data stream.                                                                                                                                                                                                                                                                                                    |
| RXCRCERR           | O   | 1                        | Indicates if the CRC code is incorrect when asserted High. Only meaningful if RX_CRC_USE = TRUE.                                                                                                                                                                                                                                                              |
| RXDATA             | O   | 8,16,32                  | Up to four bytes of decoded (8B/10B encoding) or encoded (8B/10B bypassed) receive data.                                                                                                                                                                                                                                                                      |
| RXDISPERR          | O   | 1, 2, 4                  | If 8B/10B encoding is enabled it indicates whether a disparity error has occurred on the serial line. Included in Byte-mapping scheme.                                                                                                                                                                                                                        |
| RXLOSSOFSYNC       | O   | 2                        | Status related to byte-stream synchronization (RX_LOSS_OF_SYNC_FSM)<br>If RX_LOSS_OF_SYNC_FSM = TRUE, this outputs the state of the FSM.<br>Bit 1 signals a loss of sync.<br>Bit 0 indicates a resync state.<br>If RX_LOSS_OF_SYNC_FSM = FALSE, this indicates if received data is invalid (Bit 1) and if the channel bonding sequence is recognized (Bit 0). |
| RXN <sup>(4)</sup> | I   | 1                        | Serial differential port (FPGA external)                                                                                                                                                                                                                                                                                                                      |
| RXNOTINTABLE       | O   | 1, 2, 4                  | Status of encoded data when the data is not a valid character when asserted High. Applies to the byte-mapping scheme.                                                                                                                                                                                                                                         |
| RXP <sup>(4)</sup> | I   | 1                        | Serial differential port (FPGA external)                                                                                                                                                                                                                                                                                                                      |
| RXPOLARITY         | I   | 1                        | Similar to TXPOLARITY, but for RXN and RXP. When deasserted, assumes regular polarity. When asserted, reverses polarity.                                                                                                                                                                                                                                      |



Table 3-1: GT\_CUSTOM<sup>(1)</sup>, GT\_AURORA, GT\_FIBRE\_CHAN<sup>(2)</sup>, GT\_ETHERNET<sup>(2)</sup>, GT\_INFINIBAND, and GT\_XAUI Primitive Ports (Continued)

| Port           | I/O | Port Size <sup>(3)</sup> | Definition                                                                                                                                                                                                                                                                         |
|----------------|-----|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RXREALIGN      | O   | 1                        | Signal from the PMA denoting that the byte alignment with the serial data stream changed due to a comma detection. Asserted High when alignment occurs.                                                                                                                            |
| RXRECCLK       | O   | 1                        | Recovered clock that is divided by 20.                                                                                                                                                                                                                                             |
| RXRESET        | I   | 1                        | Synchronous RX system reset that "recenters" the receive elastic buffer. It also resets 8B/10B decoder, comma detect, channel bonding, clock correction logic, and other internal receive registers. It does not reset the receiver PLL.                                           |
| RXRUNDISP      | O   | 1, 2, 4                  | Signals the running disparity (negative/positive) in the received serial data. If 8B/10B encoding bypassed, it becomes the 9th bit of the 10-bit encoded data.                                                                                                                     |
| RXUSRCLK       | I   | 1                        | Clock from a DCM that is used for reading the RX elastic buffer. It also clocks CHBONDI and CHBONDO in and out of the transceiver. Typically, the same as TXUSRCLK.                                                                                                                |
| RXUSRCLK2      | I   | 1                        | Clock output from a DCM that clocks the receiver data and status between the transceiver and the FPGA core. Typically the same as TXUSRCLK2. The relationship between RXUSRCLK and RXUSRCLK2 depends on the width of the RXDATA.                                                   |
| TXBUFERR       | O   | 1                        | Provides status of the transmission FIFO. If asserted High, an overflow / underflow has occurred. When this bit becomes set, it can only be reset by asserting TXRESET.                                                                                                            |
| TXBYPASS8B10B  | I   | 1, 2, 4                  | This control signal determines whether the 8B/10B encoding is enabled or bypassed. If the signal is asserted High, the encoding is bypassed. This creates a 10-bit interface to the FPGA core. See the 8B/10B section for more details.                                            |
| TXCHARDISPMODE | I   | 1, 2, 4                  | If 8B/10B encoding is enabled, this bus determines what mode of disparity is to be sent. When 8B/10B is bypassed, this becomes the 10th bit of the 10-bit encoded TXDATA bus for each byte specified by the byte-mapping section.                                                  |
| TXCHARDISPVAL  | I   | 1, 2, 4                  | If 8B/10B encoding is enabled, this bus determines what type of disparity is to be sent. When 8B/10B is bypassed, this becomes the 9th bit of the 10-bit encoded TXDATA bus for each byte specified by the byte-mapping section.                                                   |
| TXCHARISK      | I   | 1, 2, 4                  | If 8B/10B encoding is enabled, this control bus determines if the transmitted data is a "K" character or a Data character. A logic High indicating a K-character.                                                                                                                  |
| TXDATA         | I   | 8,16,32                  | Transmit data that can be 1, 2, or 4 bytes wide, depending on the primitive used. TXDATA [7:0] is always the last byte transmitted. The position of the first byte depends on selected TX data path width.                                                                         |
| TXFORCECRCERR  | I   | 1                        | Specifies whether to insert error in computed CRC. When TXFORCECRCERR = TRUE, the transmitter corrupts the correctly computed CRC value by XORing with the bits specified in attribute TX_CRC_FORCE_VALUE. This input can be used to test detection of CRC errors at the receiver. |
| TXINHIBIT      | I   | 1                        | If a logic High, the TX differential pairs are forced to be a constant 1/0. TXN = 1, TXP = 0                                                                                                                                                                                       |



Table 3-1: GT\_CUSTOM<sup>(1)</sup>, GT\_AURORA, GT\_FIBRE\_CHAN<sup>(2)</sup>, GT\_ETHERNET<sup>(2)</sup>, GT\_INFINIBAND, and GT\_XAUI Primitive Ports (Continued)

| Port               | I/O | Port Size <sup>(3)</sup> | Definition                                                                                                                                                                                                            |
|--------------------|-----|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TXKERR             | O   | 1, 2, 4                  | If 8B/10B encoding is enabled, this signal indicates (asserted High) when the "K" character to be transmitted is not a valid "K" character. Bits correspond to the byte-mapping scheme.                               |
| TXN <sup>(4)</sup> | O   | 1                        | Transmit differential port (FPGA external)                                                                                                                                                                            |
| TXP <sup>(4)</sup> | O   | 1                        | Transmit differential port (FPGA external)                                                                                                                                                                            |
| TXPOLARITY         | I   | 1                        | Specifies whether or not to invert the final transmitter output. Able to reverse the polarity on the TXN and TXP lines. Deasserted sets regular polarity. Asserted reverses polarity.                                 |
| TXRESET            | I   | 1                        | Synchronous TX system reset that "recenters" the transmit elastic buffer. It also resets 8B/10B encoder and other internal transmission registers. It does not reset the transmission PLL.                            |
| TXRUNDISP          | O   | 1, 2, 4                  | Signals the running disparity after this byte is encoded. Zero equals negative disparity and positive disparity for a one.                                                                                            |
| TXUSRCLK           | I   | 1                        | Clock output from a DCM that is clocked with the REFCLK (or other reference clock). This clock is used for writing the TX buffer and is frequency-locked to the REFCLK.                                               |
| TXUSRCLK2          | I   | 1                        | Clock output from a DCM that clocks transmission data and status and reconfiguration data between the transceiver and the FPGA core. The ratio between the TXUSRCLK and TXUSRCLK2 depends on the width of the TXDATA. |

**Notes:**

1. The GT\_CUSTOM ports are always the maximum port size.
2. GT\_FIBRE\_CHAN and GT\_ETHERNET ports do not have the three CHBOND\*\* or ENCHANSYNC ports.
3. The port sizes change with relation to the primitive selected and also correlate to the byte mapping.
4. External ports only accessible from package pins.

## Primitive Attributes

The primitives also contain attributes set by default to specific values controlling each specific primitive's protocol parameters. Included are channel-bonding settings (for primitives supporting channel bonding), clock correction sequences, and CRC. [Table 3-2](#) shows a brief description of each attribute. [Table 3-3](#) and [Table 3-4](#) have the default values of each primitive.

Table 3-2: Rocket I/O Transceiver Attributes

| Attribute        | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ALIGN_COMMA_MSB  | <p>True/False controls the alignment of detected commas within the transceivers 2-byte wide data path.</p> <p><b>False:</b> Align commas within a 10-bit alignment range. As a result the comma is aligned to either RXDATA[15:8] byte or RXDATA [7:0] byte in the transceivers internal data path.</p> <p><b>True:</b> Aligns comma with 20-bit alignment range.</p> <p>As a result aligns on the RXDATA[15:8] byte.</p> <p><b>NOTE:</b> If protocols (like Gigabit Ethernet) are oriented in byte pairs with commas always in even (first) byte formation, this can be set to True. Otherwise, it should be set to False.</p>                                                                                                                                                                                                                                                                    |
| CHAN_BOND_LIMIT  | <p>Integer 1-31 that defines maximum number of bytes a slave receiver can read following a channel bonding sequence and still successfully align to that sequence.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| CHAN_BOND_MODE   | <p>STRING<br/>OFF, MASTER, SLAVE_1_HOP, SLAVE_2_HOPS</p> <p><b>OFF:</b> No channel bonding involving this transceiver.</p> <p><b>MASTER:</b> This transceiver is master for channel bonding. Its CHBONDO port directly drives CHBONDI ports on one or more SLAVE_1_HOP transceivers.</p> <p><b>SLAVE_1_HOP:</b> This transceiver is a slave for channel bonding. SLAVE_1_HOP's CHBONDI is directly driven by a MASTER transceiver CHBONDO port. SLAVE_1_HOP's CHBONDO port can directly drive CHBONDI ports on one or more SLAVE_2_HOPS transceivers.</p> <p><b>SLAVE_2_HOPS:</b> This transceiver is a slave for channel bonding. SLAVE_2_HOPS CHBONDI is directly driven by a SLAVE_1_HOP CHBONDO port.</p>                                                                                                                                                                                      |
| CHAN_BOND_OFFSET | <p>Integer 0-15 that defines offset (in bytes) from channel bonding sequence for realignment. It specifies the first elastic buffer read address that all channel-bonded transceivers have immediately after channel bonding.</p> <p>CHAN_BOND_WAIT specifies the number of bytes that the master transceiver passes to RXDATA, starting with the channel bonding sequence, before the transceiver executes channel bonding (alignment) across all channel-bonded transceivers.</p> <p>CHAN_BOND_OFFSET specifies the first elastic buffer read address that all channel-bonded transceivers have immediately after channel bonding (alignment), as a positive offset from the beginning of the matched channel bonding sequence in each transceiver.</p> <p>For optimal performance of the elastic buffer, CHAN_BOND_WAIT and CHAN_BOND_OFFSET should be set to the same value (typically 8).</p> |

Table 3-2: Rocket I/O Transceiver Attributes (Continued)

| Attribute                | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CHAN_BOND_ONE_SHOT       | <p>True/False that controls repeated execution of channel bonding.</p> <p><b>False:</b> Master transceiver initiates channel bonding whenever possible (whenever channel-bonding sequence is detected in the input) as long as input ENCHANSYNC is High and RXRESET is Low.</p> <p><b>True:</b> Master transceiver initiates channel bonding only the first time it is possible (channel bonding sequence is detected in input) following negated RXRESET and asserted ENCHANSYNC. After channel-bonding alignment is done, it does not occur again until RXRESET is asserted and negated, or until ENCHANSYNC is negated and reasserted.</p> <p>Slave transceivers should always have CHAN_BOND_ONE_SHOT set to False.</p> |
| CHAN_BOND_SEQ_*_*        | <p>11-bit vectors that define the channel bonding sequence. The usage of these vectors also depends on CHAN_BOND_SEQ_LEN and CHAN_BOND_SEQ_2_USE. See <b>Receiving Vitesse Channel Bonding Sequence</b>, page 1103, for format.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| CHAN_BOND_SEQ_2_USE      | <p>Controls use of second channel bonding sequence.</p> <p><b>False:</b> Channel bonding uses only one channel bonding sequence defined by CHAN_BOND_SEQ_1_1..4.</p> <p><b>True:</b> Channel bonding uses two channel bonding sequences defined by: CHAN_BOND_SEQ_1_1..4 and CHAN_BOND_SEQ_2_1..4 as further constrained by CHAN_BOND_SEQ_LEN.</p>                                                                                                                                                                                                                                                                                                                                                                          |
| CHAN_BOND_SEQ_LEN        | <p>Integer 1-4 defines length in bytes of channel bonding sequence. This defines the length of the sequence the transceiver matches to detect opportunities for channel bonding.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| CHAN_BOND_WAIT           | <p>Integer 1-15 that defines the length of wait (in bytes) after seeing channel bonding sequence before executing channel bonding.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| CLK_COR_INSERT_IDLE_FLAG | <p>True/False controls whether RXRUNDISP input status denotes running disparity or inserted-idle flag.</p> <p><b>False:</b> RXRUNDISP denotes running disparity when RXDATA is decoded data.</p> <p><b>True:</b> RXRUNDISP is raised for the first byte of each inserted (repeated) clock correction ("Idle") sequence (when RXDATA is decoded data).</p>                                                                                                                                                                                                                                                                                                                                                                   |
| CLK_COR_KEEP_IDLE        | <p>True/False controls whether or not the final byte stream must retain at least one clock correction sequence.</p> <p><b>False:</b> Transceiver can remove all clock correction sequences to further re-center the elastic buffer during clock correction.</p> <p><b>True:</b> In the final RXDATA stream, the transceiver must leave at least one clock correction sequence per continuous stream of clock correction sequences.</p>                                                                                                                                                                                                                                                                                      |
| CLK_COR_REPEAT_WAIT      | <p>Integer 0 - 31 controls frequency of repetition of clock correction operations. This attribute specifies the minimum number of RXUSRCLK cycles without clock correction that must occur between successive clock corrections. If this attribute is zero, no limit is placed on how frequently clock correction can occur.</p>                                                                                                                                                                                                                                                                                                                                                                                            |

Table 3-2: Rocket I/O Transceiver Attributes (Continued)

| Attribute         | Description                                                                                                                                                                                                                                                                                                                                         |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CLK_COR_SEQ_*_*   | 11-bit vectors that define the sequence for clock correction. The attribute used depends on the CLK_COR_SEQ_LEN and CLK_COR_SEQ_2_USE.                                                                                                                                                                                                              |
| CLK_COR_SEQ_2_USE | True/False Control use of second clock correction sequence.<br><b>False:</b> Clock correction uses only one clock correction sequence defined by CLK_COR_SEQ_1_1..4.<br><b>True:</b> Clock correction uses two clock correction sequences defined by:<br>CLK_COR_SEQ_1_1..4 and<br>CLK_COR_SEQ_2_1..4<br>as further constrained by CLK_COR_SEQ_LEN. |
| CLK_COR_SEQ_LEN   | Integer that defines the length of the sequence the transceiver matches to detect opportunities for clock correction. It also defines the size of the correction, since the transceiver executes clock correction by repeating or skipping entire clock correction sequences.                                                                       |
| CLK_CORRECT_USE   | True/False controls the use of clock correction logic.<br><b>False:</b> Permanently disable execution of clock correction (rate matching). Clock RXUSRCLK must be frequency-locked with RXRECCLK in this case.<br><b>True:</b> Enable clock correction (normal mode).                                                                               |
| COMMA_10B_MASK    | This 10-bit vector defines the mask that is ANDed with the incoming serial-bit stream before comparison against PCOMMA_10B_VALUE and MCOMMA_10B_VALUE.                                                                                                                                                                                              |
| CRC_END_OF_PKT    | K28_0, K28_1, K28_2, K28_3, K28_4, K28_5, K28_6, K28_7, K23_7, K27_7, K29_7, K30_7 End-of-packet (EOP) K-character for USER_MODE CRC. Must be one of the 12 legal K-character values.                                                                                                                                                               |
| CRC_FORMAT        | ETHERNET, INFINIBAND, FIBRE_CHAN, USER_MODE CRC algorithm selection. Modifiable only for GT_AURORA_n, GT_XAUI_n, and GT_CUSTOM. USER_MODE allows user definition of start-of-packet and end-of-packet K-characters.                                                                                                                                 |
| CRC_START_OF_PKT  | K28_0, K28_1, K28_2, K28_3, K28_4, K28_5, K28_6, K28_7, K23_7, K27_7, K29_7, K30_7 Start-of-packet (SOP) K-character for USER_MODE CRC. Must be one of the 12 legal K-character values.                                                                                                                                                             |
| DEC_MCOMMA_DETECT | True/False controls the raising of per-byte flag RXCHARISCOMMA on minus-comma.                                                                                                                                                                                                                                                                      |
| DEC_PCOMMA_DETECT | True/False controls the raising of per-byte flag RXCHARISCOMMA on plus-comma.                                                                                                                                                                                                                                                                       |

Table 3-2: Rocket I/O Transceiver Attributes (Continued)

| Attribute                       | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DEC_VALID_COMMA_ONLY            | <p>True/False controls the raising of RXCHARISCOMMA on an invalid comma.</p> <p><b>False:</b> Raise RXCHARISCOMMA on:</p> <p style="padding-left: 40px;">0011111xxx<br/>(if DEC_PCOMMA_DETECT is TRUE)</p> <p>and/or on:</p> <p style="padding-left: 40px;">1100000xxx<br/>(if DEC_MCOMMA_DETECT is TRUE)</p> <p>regardless of the settings of the xxx bits.</p> <p><b>True:</b> Raise RXCHARISCOMMA only on valid characters that are in the 8B/10B translation.</p> |
| MCOMMA_10B_VALUE                | <p>This 10-bit vector defines minus-comma for the purpose of raising RXCOMMADET and realigning the serial bit stream byte boundary. This definition does not affect 8B/10B encoding or decoding. Also see COMMA_10B_MASK.</p>                                                                                                                                                                                                                                         |
| MCOMMA_DETECT                   | <p>True/False indicates whether to raise or not raise the RXCOMMADET when minus-comma is detected.</p>                                                                                                                                                                                                                                                                                                                                                                |
| PCOMMA_10B_VALUE                | <p>This 10-bit vector defines plus-comma for the purpose of raising RXCOMMADET and realigning the serial bit stream byte boundary. This definition does not affect 8B/10B encoding or decoding. Also see COMMA_10B_MASK.</p>                                                                                                                                                                                                                                          |
| PCOMMA_DETECT                   | <p>True/False indicates whether to raise or not raise the RXCOMMADET when plus-comma is detected.</p>                                                                                                                                                                                                                                                                                                                                                                 |
| RX_BUFFER_USE                   | <p>Always set to True.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| RX_CRC_USE,<br>TX_CRC_USE       | <p>True/False determines if CRC is used or not.</p>                                                                                                                                                                                                                                                                                                                                                                                                                   |
| RX_DATA_WIDTH,<br>TX_DATA_WIDTH | <p>Integer (1, 2, or 4). Relates to the data width of the FPGA fabric interface.</p>                                                                                                                                                                                                                                                                                                                                                                                  |
| RX_DECODE_USE                   | <p>This determines if the 8B/10B decoding is bypassed. False denotes that it is bypassed.</p>                                                                                                                                                                                                                                                                                                                                                                         |
| RX_LOS_INVALID_INCR             | <p>Power of two in a range of 1 to 128 that denotes the number of valid characters required to "cancel out" appearance of one invalid character for loss of sync determination.</p>                                                                                                                                                                                                                                                                                   |
| RX_LOS_THRESHOLD                | <p>Power of two in a range of 4 to 512. When divided by RX_LOS_INVALID_INCR, denotes the number of invalid characters required to cause FSM transition to "sync lost" state.</p>                                                                                                                                                                                                                                                                                      |
| RX_LOSS_OF_SYNC_FSM             | <p>True/False denotes the nature of RXLOSSOFSYNC output.</p> <p><b>True:</b> RXLOSSOFSYNC outputs the state of the FSM bit.<br/>See <a href="#">RXLOSSOFSYNC</a>, page 1070, for details.</p>                                                                                                                                                                                                                                                                         |
| SERDES_10B                      | <p>Denotes whether the reference clock runs at 1/20 or 1/10 the serial bit rate. True denotes 1/10 and False denotes 1/20. False supports a serial bitstream range of 800 Mb/s to 3.125 Gb/s. True supports a range of 500 Mb/s to 1.0 Gb/s.</p>                                                                                                                                                                                                                      |

Table 3-2: Rocket I/O Transceiver Attributes (Continued)

| Attribute          | Description                                                                                                                                                                                                                                                                  |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TERMINATION_IMP    | Integer (50 or 75). Termination impedance of either 50Ω or 75Ω. Refers to both the RX and TX.                                                                                                                                                                                |
| TX_BUFFER_USE      | Always set to True.                                                                                                                                                                                                                                                          |
| TX_CRC_FORCE_VALUE | 8-bit vector. Value to corrupt TX CRC computation when input TXFORCECRCERR is high. This value is XORed with the correctly computed CRC value, corrupting the CRC if TX_CRC_FORCE_VALUE is nonzero. This can be used to test CRC error detection in the receiver downstream. |
| TX_DIFF_CTRL       | An integer value (400, 500, 600, 700, or 800) representing 400 mV, 500 mV, 600 mV, 700 mV, or 800 mV of voltage difference between the differential lines. Twice this value is the peak-peak voltage.                                                                        |
| TX_PREEMPHASIS     | An integer value (0-3) that sets the output driver pre-emphasis to improve output waveform shaping for various load conditions. Larger value denotes stronger pre-emphasis. See pre-emphasis values in <a href="#">Table 4-2, page 1120</a> .                                |

## Modifiable Primitives

As shown in Table 3-3 and Table 3-4, only certain attributes are modifiable for any primitive. These attributes help to define the protocol used by the primitive. Only the GT\_CUSTOM primitive allows the user to modify all of the attributes to a protocol not supported by another transceiver primitive. This allows for complete flexibility. The other primitives allow modification of the analog attributes of the serial data lines and several channel-bonding values.

Table 3-3: Default Attribute Values: GT\_AURORA, GT\_CUSTOM, GT\_ETHERNET

| Attribute                | Default GT_AURORA          | Default GT_CUSTOM <sup>(1)</sup> | Default GT_ETHERNET  |
|--------------------------|----------------------------|----------------------------------|----------------------|
| ALIGN_COMMA_MSB          | False                      | False                            | False                |
| CHAN_BOND_LIMIT          | 16                         | 16                               | 1                    |
| CHAN_BOND_MODE           | OFF <sup>(2)</sup>         | OFF                              | OFF                  |
| CHAN_BOND_OFFSET         | 8                          | 8                                | 0                    |
| CHAN_BOND_ONE_SHOT       | False <sup>(2)</sup>       | False                            | True                 |
| CHAN_BOND_SEQ_1_1        | 00101111100                | 00000000000                      | 00000000000          |
| CHAN_BOND_SEQ_1_2        | 00000000000                | 00000000000                      | 00000000000          |
| CHAN_BOND_SEQ_1_3        | 00000000000                | 00000000000                      | 00000000000          |
| CHAN_BOND_SEQ_1_4        | 00000000000                | 00000000000                      | 00000000000          |
| CHAN_BOND_SEQ_2_1        | 00000000000                | 00000000000                      | 00000000000          |
| CHAN_BOND_SEQ_2_2        | 00000000000                | 00000000000                      | 00000000000          |
| CHAN_BOND_SEQ_2_3        | 00000000000                | 00000000000                      | 00000000000          |
| CHAN_BOND_SEQ_2_4        | 00000000000                | 00000000000                      | 00000000000          |
| CHAN_BOND_SEQ_2_USE      | False                      | False                            | False                |
| CHAN_BOND_SEQ_LEN        | 1                          | 1                                | 1                    |
| CHAN_BOND_WAIT           | 8                          | 8                                | 7                    |
| CLK_COR_INSERT_IDLE_FLAG | False <sup>(2)</sup>       | False                            | False <sup>(2)</sup> |
| CLK_COR_KEEP_IDLE        | False <sup>(2)</sup>       | False                            | False <sup>(2)</sup> |
| CLK_COR_REPEAT_WAIT      | 1 <sup>(2)</sup>           | 1                                | 1 <sup>(2)</sup>     |
| CLK_COR_SEQ_1_1          | 00100011100                | 00000000000                      | 00110111100          |
| CLK_COR_SEQ_1_2          | 00100011100 <sup>(4)</sup> | 00000000000                      | 00001010000          |
| CLK_COR_SEQ_1_3          | 00100011100 <sup>(5)</sup> | 00000000000                      | 00000000000          |
| CLK_COR_SEQ_1_4          | 00100011100 <sup>(5)</sup> | 00000000000                      | 00000000000          |
| CLK_COR_SEQ_2_1          | 00000000000                | 00000000000                      | 00000000000          |
| CLK_COR_SEQ_2_2          | 00000000000                | 00000000000                      | 00000000000          |
| CLK_COR_SEQ_2_3          | 00000000000                | 00000000000                      | 00000000000          |
| CLK_COR_SEQ_2_4          | 00000000000                | 00000000000                      | 00000000000          |
| CLK_COR_SEQ_2_USE        | False                      | False                            | False                |

Table 3-3: Default Attribute Values: GT\_AURORA, GT\_CUSTOM, GT\_ETHERNET (Continued)

| Attribute            | Default<br>GT_AURORA    | Default<br>GT_CUSTOM <sup>(1)</sup> | Default<br>GT_ETHERNET  |
|----------------------|-------------------------|-------------------------------------|-------------------------|
| CLK_COR_SEQ_LEN      | N <sup>(3)</sup>        | 1                                   | 2                       |
| CLK_CORRECT_USE      | True                    | True                                | True                    |
| COMMA_10B_MASK       | 1111111111              | 1111111000                          | 1111111000              |
| CRC_END_OF_PKT       | K29_7                   | K29_7                               | K29_7                   |
| CRC_FORMAT           | USER_MODE               | USER_MODE                           | ETHERNET                |
| CRC_START_OF_PKT     | K27_7                   | K27_7                               | K27_7                   |
| DEC_MCOMMA_DETECT    | True                    | True                                | True                    |
| DEC_PCOMMA_DETECT    | True                    | True                                | True                    |
| DEC_VALID_COMMA_ONLY | True                    | True                                | True                    |
| MCOMMA_10B_VALUE     | 1100000101              | 1100000000                          | 1100000000              |
| MCOMMA_DETECT        | True                    | True                                | True                    |
| PCOMMA_10B_VALUE     | 0011111010              | 0011111000                          | 0011111000              |
| PCOMMA_DETECT        | True                    | True                                | True                    |
| RX_BUFFER_USE        | True                    | True                                | True                    |
| RX_CRC_USE           | False <sup>(2)</sup>    | False                               | False <sup>(2)</sup>    |
| RX_DATA_WIDTH        | N <sup>(3)</sup>        | 2                                   | N <sup>(3)</sup>        |
| RX_DECODE_USE        | True                    | True                                | True                    |
| RX_LOS_INVALID_INCR  | 1 <sup>(2)</sup>        | 1                                   | 1 <sup>(2)</sup>        |
| RX_LOS_THRESHOLD     | 4 <sup>(2)</sup>        | 4                                   | 4 <sup>(2)</sup>        |
| RX_LOSS_OF_SYNC_FSM  | True <sup>(2)</sup>     | True                                | True <sup>(2)</sup>     |
| SERDES_10B           | False <sup>(2)</sup>    | False                               | False <sup>(2)</sup>    |
| TERMINATION_IMP      | 50 <sup>(2)</sup>       | 50                                  | 50 <sup>(2)</sup>       |
| TX_BUFFER_USE        | True                    | True                                | True                    |
| TX_CRC_FORCE_VALUE   | 11010110 <sup>(2)</sup> | 11010110                            | 11010110 <sup>(2)</sup> |
| TX_CRC_USE           | False <sup>(2)</sup>    | False                               | False <sup>(2)</sup>    |
| TX_DATA_WIDTH        | N <sup>(3)</sup>        | 2                                   | N <sup>(3)</sup>        |
| TX_DIFF_CTRL         | 500 <sup>(2)</sup>      | 500                                 | 500 <sup>(2)</sup>      |
| TX_PREEMPHASIS       | 0 <sup>(2)</sup>        | 0                                   | 0 <sup>(2)</sup>        |

**Notes:**

1. All GT\_CUSTOM attributes are modifiable.
2. Modifiable attribute for specific primitives.
3. Depends on primitive used: either 1, 2, or 4.
4. Attribute value only when RX\_DATA\_WIDTH is 2 or 4. When RX\_DATA\_WIDTH is 1, attribute value is 0.
5. Attribute value only when RX\_DATA\_WIDTH is 4. When RX\_DATA\_WIDTH is 1 or 2, attribute value is 0.



Table 3-4: Default Attribute Values: GT\_FIBRE\_CHAN, GT\_INFINIBAND, and GT\_XAUI

| Attribute                | Default<br>GT_FIBRE_CHAN | Default<br>GT_INFINIBAND         | Default<br>GT_XAUI   |
|--------------------------|--------------------------|----------------------------------|----------------------|
| ALIGN_COMMA_MSB          | False                    | False                            | False                |
| CHAN_BOND_LIMIT          | 1                        | 16                               | 16                   |
| CHAN_BOND_MODE           | OFF                      | OFF <sup>(1)</sup>               | OFF <sup>(1)</sup>   |
| CHAN_BOND_OFFSET         | 0                        | 8                                | 8                    |
| CHAN_BOND_ONE_SHOT       | True                     | False <sup>(1)</sup>             | False <sup>(1)</sup> |
| CHAN_BOND_SEQ_1_1        | 000000000000             | 001101111100                     | 001011111100         |
| CHAN_BOND_SEQ_1_2        | 000000000000             | Lane ID (Modify with<br>Lane ID) | 000000000000         |
| CHAN_BOND_SEQ_1_3        | 000000000000             | 00001001010                      | 000000000000         |
| CHAN_BOND_SEQ_1_4        | 000000000000             | 00001001010                      | 000000000000         |
| CHAN_BOND_SEQ_2_1        | 000000000000             | 001101111100                     | 000000000000         |
| CHAN_BOND_SEQ_2_2        | 000000000000             | Lane ID (Modify with<br>Lane ID) | 000000000000         |
| CHAN_BOND_SEQ_2_3        | 000000000000             | 00001000101                      | 000000000000         |
| CHAN_BOND_SEQ_2_4        | 000000000000             | 00001000101                      | 000000000000         |
| CHAN_BOND_SEQ_2_USE      | False                    | True                             | False                |
| CHAN_BOND_SEQ_LEN        | 1                        | 4                                | 1                    |
| CHAN_BOND_WAIT           | 7                        | 8                                | 8                    |
| CLK_COR_INSERT_IDLE_FLAG | False <sup>(1)</sup>     | False <sup>(1)</sup>             | False <sup>(1)</sup> |
| CLK_COR_KEEP_IDLE        | False <sup>(1)</sup>     | False <sup>(1)</sup>             | False <sup>(1)</sup> |
| CLK_COR_REPEAT_WAIT      | 2 <sup>(1)</sup>         | 1 <sup>(1)</sup>                 | 1 <sup>(1)</sup>     |
| CLK_COR_SEQ_1_1          | 001101111100             | 001000111100                     | 001000111100         |
| CLK_COR_SEQ_1_2          | 00010010101              | 000000000000                     | 000000000000         |
| CLK_COR_SEQ_1_3          | 00010110101              | 000000000000                     | 000000000000         |
| CLK_COR_SEQ_1_4          | 00010110101              | 000000000000                     | 000000000000         |
| CLK_COR_SEQ_2_1          | 000000000000             | 000000000000                     | 000000000000         |
| CLK_COR_SEQ_2_2          | 000000000000             | 000000000000                     | 000000000000         |
| CLK_COR_SEQ_2_3          | 000000000000             | 000000000000                     | 000000000000         |
| CLK_COR_SEQ_2_4          | 000000000000             | 000000000000                     | 000000000000         |
| CLK_COR_SEQ_2_USE        | False                    | False                            | False                |
| CLK_COR_SEQ_LEN          | 4                        | 1                                | 1                    |
| CLK_CORRECT_USE          | True                     | True                             | True                 |
| COMMA_10B_MASK           | 1111111000               | 1111111000                       | 1111111000           |
| CRC_END_OF_PKT           | K29_7                    | Note (3)                         | K29_7 <sup>(1)</sup> |

Table 3-4: Default Attribute Values: GT\_FIBRE\_CHAN, GT\_INFINIBAND, and GT\_XAUI (Continued)

| Attribute                | Default<br>GT_FIBRE_CHAN | Default<br>GT_INFINIBAND    | Default<br>GT_XAUI       |
|--------------------------|--------------------------|-----------------------------|--------------------------|
| CRC_FORMAT               | FIBRE_CHAN               | INFINIBAND                  | USER_MODE <sup>(1)</sup> |
| CRC_START_OF_PKT         | K27_7                    | Note (3)                    | K27_7 <sup>(1)</sup>     |
| DEC_MCOMMA_DETECT        | True                     | True                        | True                     |
| DEC_PCOMMA_DETECT        | True                     | True                        | True                     |
| DEC_VALID_COMMA_ONLY     | True                     | True                        | True                     |
| Lane ID(INFINIBAND ONLY) | NA                       | 000000000000 <sup>(1)</sup> | NA                       |
| MCOMMA_10B_VALUE         | 1100000000               | 1100000000                  | 1100000000               |
| MCOMMA_DETECT            | True                     | True                        | True                     |
| PCOMMA_10B_VALUE         | 0011111000               | 0011111000                  | 0011111000               |
| PCOMMA_DETECT            | True                     | True                        | True                     |
| RX_BUFFER_USE            | True                     | True                        | True                     |
| RX_CRC_USE               | False <sup>(1)</sup>     | False <sup>(1)</sup>        | False <sup>(1)</sup>     |
| RX_DATA_WIDTH            | N <sup>(2)</sup>         | N <sup>(2)</sup>            | N <sup>(2)</sup>         |
| RX_DECODE_USE            | True                     | True                        | True                     |
| RX_LOS_INVALID_INCR      | 1 <sup>(1)</sup>         | 1 <sup>(1)</sup>            | 1 <sup>(1)</sup>         |
| RX_LOS_THRESHOLD         | 4 <sup>(1)</sup>         | 4 <sup>(1)</sup>            | 4 <sup>(1)</sup>         |
| RX_LOSS_OF_SYNC_FSM      | True <sup>(1)</sup>      | True <sup>(1)</sup>         | True <sup>(1)</sup>      |
| SERDES_10B               | False <sup>(1)</sup>     | False <sup>(1)</sup>        | False <sup>(1)</sup>     |
| TERMINATION_IMP          | 50 <sup>(1)</sup>        | 50 <sup>(1)</sup>           | 50 <sup>(1)</sup>        |
| TX_BUFFER_USE            | True                     | True                        | True                     |
| TX_CRC_FORCE_VALUE       | 11010110 <sup>(1)</sup>  | 11010110 <sup>(1)</sup>     | 11010110 <sup>(1)</sup>  |
| TX_CRC_USE               | False <sup>(1)</sup>     | False <sup>(1)</sup>        | False <sup>(1)</sup>     |
| TX_DATA_WIDTH            | N <sup>(2)</sup>         | N <sup>(2)</sup>            | N <sup>(2)</sup>         |
| TX_DIFF_CTRL             | 500 <sup>(1)</sup>       | 500 <sup>(1)</sup>          | 500 <sup>(1)</sup>       |
| TX_PREEMPHASIS           | 0 <sup>(1)</sup>         | 0 <sup>(1)</sup>            | 0 <sup>(1)</sup>         |

**Notes:**

1. Modifiable attribute for specific primitives.
2. Depends on primitive used: either 1, 2, or 4.
3. CRC\_EOP and CRC\_SOP are not applicable for this primitive.

## Byte Mapping

Most of the 4-bit wide status and control buses correlate to a specific byte of the TXDATA or RXDATA. This scheme is shown in Table 3-5. This creates a way to tie all the signals together regardless of the data path width needed for the GT\_CUSTOM. All other primitives with specific data width paths and all byte-mapped ports are affected by this situation. For example, a 1-byte wide data path has only 1-bit control and status bits (TXKERR[0]) correlating to the data bits TXDATA[7:0]. Note 3 in Table 3-1 shows the ports that use byte mapping.

Table 3-5: Control/Status Bus Association to Data Bus Byte Paths

| Control/Status Bit | Data Bits |
|--------------------|-----------|
| [0]                | [7:0]     |
| [1]                | [15:8]    |
| [2]                | [23:16]   |
| [3]                | [31:24]   |

## Clocking

### Clock Signals

There are five clock inputs into each Rocket I/O transceiver instantiation (Table 3-6). REFCLK is a clock generated from an external source. REFCLK is connected to the REFCLK of the Rocket I/O transceiver. It also clocks a Digital Clock Manager (DCM) to generate all of the other clocks for the gigabit transceiver. Typically, TXUSRCLK = RXUSRCLK and TXUSRCLK2 = RXUSRCLK2. The transceiver uses one or two clocks generated by the DCM. As an example, the USRCLK and USRCLK2 clocks run at the same speed if the 2-byte data path is used. The USRCLK must always be frequency-locked to the reference clock, REFCLK of the Rocket I/O transceiver.

**NOTE:** The REFCLK must be at least 40 MHz with a duty cycle between 45% and 55%, and should have a frequency stability of 100 ppm or better, with jitter as low as possible. Module 3 of the Virtex-II Pro data sheet gives further details.

Table 3-6: Clock Ports

| Clock     | I/Os   | Description                                                                                                                                                                                                                             |
|-----------|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RXRECCLK  | Output | Recovered clock (from serial data stream) divided by 20                                                                                                                                                                                 |
| REFCLK    | Input  | Reference clock used to read the TX FIFO and multiplied by 20 for parallel-to-serial conversion (20X)                                                                                                                                   |
| REFCLK2   | Input  | Reference clock used to read the TX FIFO and multiplied by 20 for parallel-to-serial conversion (20X)                                                                                                                                   |
| REFCLKSEL | Input  | Selects which reference clock is used. 0 selects REFCLK; 1 selects REFCLK2.                                                                                                                                                             |
| RXUSRCLK  | Input  | Clock from FPGA used for reading the RX Elastic Buffer. Clock signals CHBONDI and CHBONDO into and out of the transceiver. This clock is typically the same as TXUSRCLK.                                                                |
| TXUSRCLK  | Input  | Clock from FPGA used for writing the TX Buffer. This clock must be frequency locked to REFCLK for proper operation.                                                                                                                     |
| RXUSRCLK2 | Input  | Clock from FPGA used to clock RX data and status between the transceiver and FPGA fabric. The relationship between RXUSRCLK2 and RXUSRCLK depends on the width of the receiver data path. RXUSRCLK2 is typically the same as TXUSRCLK2. |
| TXUSRCLK2 | Input  | Clock from FPGA used to clock TX data and status between the transceiver and FPGA fabric. The relationship between TXUSRCLK2 and TXUSRCLK depends on the width of the transmission data path.                                           |

## Clock Ratio

USRCLK2 clocks the data buffers. The ability to send parallel data to the transceiver at three different widths requires the user to change the frequency of USRCLK2. This creates a frequency ratio between USRCLK and USRCLK2. The falling edges of the clocks must align. Finally, for a 4-byte data path, the 1-byte data path creates a clocking scheme where USRCLK2 is phase-shifted 180° and at twice the rate of USRCLK.

**Table 3-7: Data Width Clock Ratios**

| Data Width | Frequency Ratio of USRCLK/USRCLK2 |
|------------|-----------------------------------|
| 1 byte     | 1:2 <sup>(1)</sup>                |
| 2 byte     | 1:1                               |
| 4 byte     | 2:1 <sup>(1)</sup>                |

**Notes:**

- Each edge of slower clock must align with falling edge of faster clock

## Digital Clock Manager (DCM) Examples

With at least three different clocking schemes possible on the transceiver, a DCM is the best way to create these schemes.

**Table 3-8** shows typical DCM connections for several transceiver clocks. REFCLK is the input reference clock for the DCM. The other clocks are generated by the DCM. The DCM establishes a desired phase relationship between RXUSRCLK, TXUSRCLK, etc. in the FPGA core and REFCLK at the pad.

**Table 3-8: DCM Outputs for Different DATA\_WIDTHs**

| SERDES_10B | TX_DATA_WIDTH<br>RX_DATA_WIDTH | REFCLK | TXUSRCLK<br>RXUSRCLK   | TXUSRCLK2<br>RXUSRCLK2 |
|------------|--------------------------------|--------|------------------------|------------------------|
| False      | 1                              | CLKIN  | CLK0                   | CLK2X180               |
| False      | 2                              | CLKIN  | CLK0                   | CLK0                   |
| False      | 4                              | CLKIN  | CLK180 <sup>(1)</sup>  | CLKDV (divide by 2)    |
| True       | 1                              | CLKIN  | CLKDV (divide by 2)    | CLK180 <sup>(1)</sup>  |
| True       | 2                              | CLKIN  | CLKDV (divide by 2)    | CLKDV (divide by 2)    |
| True       | 4                              | CLKIN  | CLKFX180 (divide by 2) | CLKDV (divide by 4)    |

**Notes:**

- Since CLK0 is needed for feedback, it can be used instead of CLK180 to clock USRCLK or USRCLK2 of the transceiver with the use of the transceiver's local inverter, saving a global buffer (BUFG).

## Example 1: Two-Byte Clock

The following HDL codes are examples of a simple clock scheme using 2-byte data with both USRCLK and USRCLK2 at the same frequency. USRCLK\_M is the input for both USRCLK and USRCLK2.

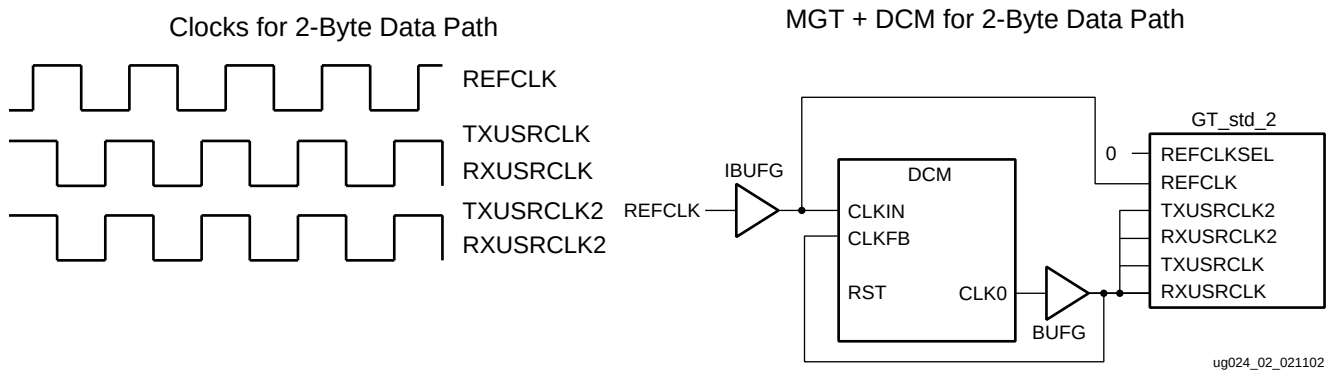


Figure 3-1: Two-Byte Clock

### VHDL Template

```
-- Module: TWO_BYTE_CLK
-- Description: VHDL submodule
-- DCM for 2-byte GT
--
-- Device: Virtex-II Pro Family

library IEEE;
use IEEE.std_logic_1164.all;
--
-- pragma translate_off
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
-- pragma translate_on
--
entity TWO_BYTE_CLK is
 port (
 REFCLKIN : in std_logic;
 RST : in std_logic;
 USRCLK_M : out std_logic;
 REFCLK : out std_logic;
 LOCK : out std_logic;
);
end TWO_BYTE_CLK;
--
architecture TWO_BYTE_CLK_arch of TWO_BYTE_CLK is
 --
 -- Components Declarations:
 component BUFG
 port (
 I : in std_logic;
 O : out std_logic
);
 end component;
 --
 component IBUFG
 port (
 I : in std_logic;
 O : out std_logic
);
 end component;
```

```

);
end component;
--
component DCM
 port (
 CLKIN : in std_logic;
 CLKFB : in std_logic;
 DSSEN : in std_logic;
 PSINCDEC : in std_logic;
 PSEN : in std_logic;
 PSCLK : in std_logic;
 RST : in std_logic;
 CLK0 : out std_logic;
 CLK90 : out std_logic;
 CLK180 : out std_logic;
 CLK270 : out std_logic;
 CLK2X : out std_logic;
 CLK2X180 : out std_logic;
 CLKDV : out std_logic;
 CLKFX : out std_logic;
 CLKFX180 : out std_logic;
 LOCKED : out std_logic;
 PSDONE : out std_logic;
 STATUS : out std_logic_vector (7 downto 0);
);
end component;
--
-- Signal Declarations:
--
signal GND : std_logic;
signal CLK0_W : std_logic;
signal CLK1X_W : std_logic;

begin

GND <= '0';
--
CLK1X <= CLK1X_W;
--
-- DCM Instantiation
U_DCM: DCM
 port map (
 CLKIN => REFCLK,
 CLKFB => USRCLK_M,
 DSSEN => GND,
 PSINCDEC => GND,
 PSEN => GND,
 PSCLK => GND,
 RST => RST,
 CLK0 => CLK0_W,
 LOCKED => LOCK
);
--
-- BUFG Instantiation
U_BUFG: IBUFG
 port map (
 I => REFCLKIN,
 O => REFCLK
);

```

```

U2_BUFG: BUFG
 port map (
 I => CLK0_W,
 O => USRCLK_M
);

```

```

end TWO_BYTE_CLK_arch;

```

### Verilog Template

```

//Module: TWO_BYTE_CLK
//Description: Verilog Submodule
// DCM for 2-byte GT
//
// Device: Virtex-II Pro Family

module TWO_BYTE_CLK (
 REFCLKIN,
 REFCLK,
 USRCLK_M,
 DCM_LOCKED
);

 input REFCLKIN;
 output REFCLK;
 output USRCLK_M;
 output DCM_LOCKED;

 wire REFCLKIN;
 wire REFCLK;
 wire USRCLK_M;
 wire DCM_LOCKED;
 wire REFCLKINBUF;
 wire clk_i;

 DCM dcm1 (
 .CLKFB (USRCLK_M),
 .CLKIN (REFCLKINBUF), .DSSEN(1'b0),
 .PCLK (1'b0),
 .PSEN (1'b0),
 .PSINCDEC (1'b0),
 .RST (1'b0),
 .CLK0 (clk_i),
 .CLK90 (),
 .CLK180 (),
 .CLK270 (),
 .CLK2X (),
 .CLK2X180 (),
 .CLKDV (),
 .CLKFX (),
 .CLKFX180 (),
 .LOCKED (DCM_LOCKED),
 .PSDONE (),
 .STATUS ()
);

 BUFG buf1 (
 .I (clk_i),
 .O (USRCLK_M)
);

 IBUFG buf2(
 .I (REFCLKIN),

```

```
.0 (REFCLKINBUF));
```

endmodule

### Example 2: Four-Byte Clock

If a 4-byte or 1-byte data path is chosen, the ratio between USRCLK and USRCLK2 changes. The time it take for the SERDES to serialize the parallel data requires the change in ratios.

The DCM example ([Figure 3-2](#)) is detailed for a 4-byte data path. If 3.125 Gb/s is required, the REFCLK is 156 MHz and the USRCLK2\_M only runs at 78 MHz including the clocking for any interface logic. Both USRCLK and USRCLK2 are aligned on the falling edge since the USRCLK\_M is 180° out of phase when using local inverters with the transceiver.

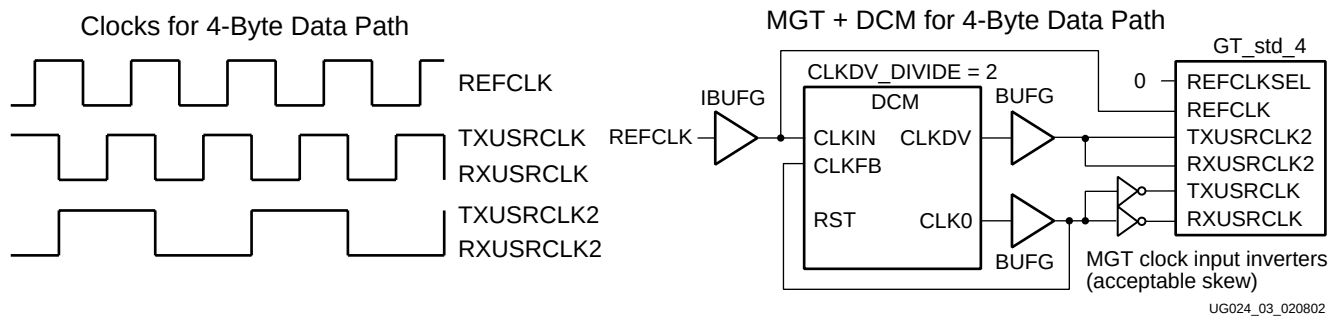


**Figure 3-2: Four-Byte Clock**

## VHDL Template

```
-- Module: FOUR_BYTE_CLK
-- Description: VHDL submodule
-- DCM for 4-byte GT
--
-- Device: Virtex-II Pro Family

library IEEE;
use IEEE.std_logic_1164.all;
--
-- pragma translate_off
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
-- pragma translate_on
--
entity FOUR_BYTE_CLK is
port (
 REFCLKIN : in std_logic;
 RST : in std_logic;
 USRCLK_M : out std_logic;
 USRCLK2_M : out std_logic;
 REFCLK : out std_logic;
 LOCK : out std_logic
);
end FOUR_BYTE_CLK;
--
architecture FOUR_BYTE_CLK_arch of FOUR_BYTE_CLK is
--
-- Components Declarations:
component BUFG
port (
 I : in std_logic;
```



```

 0 : out std_logic
);
end component;
--
component IBUFG
 port (
 I : in std_logic;
 O : out std_logic
);
end component;
--
component DCM
 port (
 CLKIN : in std_logic;
 CLKFB : in std_logic;
 DSSSEN : in std_logic;
 PSINCDEC : in std_logic;
 PSEN : in std_logic;
 PSCLK : in std_logic;
 RST : in std_logic;
 CLK0 : out std_logic;
 CLK90 : out std_logic;
 CLK180 : out std_logic;
 CLK270 : out std_logic;
 CLK2X : out std_logic;
 CLK2X180 : out std_logic;
 CLKDV : out std_logic;
 CLKFX : out std_logic;
 CLKFX180 : out std_logic;
 LOCKED : out std_logic;
 PSDONE : out std_logic;
 STATUS : out std_logic_vector (7 downto 0)
);
end component;
--
-- Signal Declarations:
--
signal GND : std_logic;
signal CLK0_W : std_logic;
signal CLKDV_W : std_logic;

begin

GND <= '0';
-- DCM Instantiation
U_DCM: DCM
 port map (
 CLKIN => REFCLK, CLKFB => CLK0_W,
 DSSSEN => GND,
 PSINCDEC => GND,
 PSEN => GND,
 PSCLK => GND,
 RST => RST,
 CLK0 => CLK0_W,
 CLKDV => CLKDV_W,
 LOCKED => LOCK
);

-- BUFG Instantiation
U_BUFG: IBUFG
 port map (
 I => REFCLKIN,

```

```

 0 => REFCLK);

U2_BUF: BUFG
 port map (
 I => CLK0_W,
 O => USRCLK_M
);

U3_BUF: BUFG
 port map (
 I => CLKDV_W,
 O => USRCLK2_M
);

end FOUR_BYTE_CLK_arch;

```

### Verilog Template

```

// Module: FOUR_BYTE_CLK
// Description: Verilog Submodule
// DCM for 4-byte GT
//
// Device: Virtex-II Pro Family

module FOUR_BYTE_CLK(
 REFCLKIN,
 REFCLK,
 USRCLK_M,
 USRCLK2_M,
 DCM_LOCKED
);

input REFCLKIN;
output REFCLK;
output USRCLK_M;
output USRCLK2_M;
output DCM_LOCKED;

wire REFCLKIN;
wire REFCLK;
wire USRCLK_M;
wire USRCLK2_M;
wire DCM_LOCKED;
wire REFCLKINBUF;
wire clkdv2;
wire clk_i;

 DCM dcm1 (
 .CLKFB (USRCLK_M),
 .CLKIN (REFCLKINBUF) , .DSSEN (
1'b0),
 .PSCLK (1'b0),
 .PSEN (1'b0),
 .PSINCDEC (1'b0),
 .RST (1'b0),
 .CLK0 (clk_i),
 .CLK90 (),
 .CLK180 (),
 .CLK270 (),
 .CLK2X (),
 .CLK2X180 (),
 .CLKDV (clkdv2),

```

```

 .CLKFX (),
 .CLKFX180 (),
 .LOCKED (DCM_LOCKED),
 .PSDONE (),
 .STATUS ()
);

 BUFG buf1 (
 .I (clkdv2),
 .O (USRCLK2_M)
);

 BUFG buf2 (
 .I (clk_i),
 .O (USRCLK_M)
);

 IBUFG buf3(
 .I (REFCLKIN),
 .O (REFCLKINBUF));
endmodule

```

### Example 3: One-Byte Clock

This is the 1-byte wide data path clocking scheme example. USRCLK2\_M is twice as fast as USRCLK\_M. It is also phase-shifted 180° for falling edge alignment.

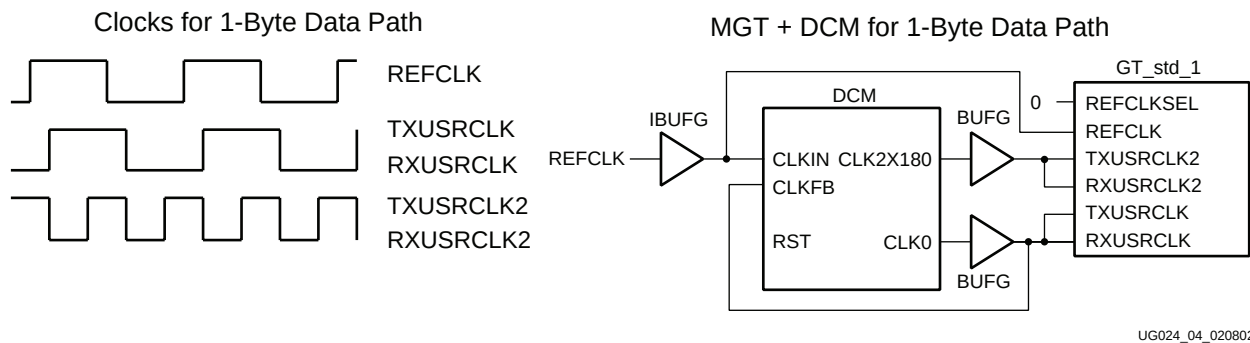


**Figure 3-3: One-Byte Clock**

## VHDL Template

```
-- Module: ONE_BYTE_CLK
-- Description: VHDL submodule
-- DCM for 1-byte GT
--
-- Device: Virtex-II Pro Family
```

```
library IEEE;
use IEEE.std_logic_1164.all;
--
-- pragma translate_off
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
-- pragma translate_on
--
entity ONE_BYTE_CLK is
 port (
 REFCLKIN : in std_logic;
```

```

 RST : in std_logic;
 USRCLK_M : out std_logic;
 USRCLK2_M : out std_logic;
 REFCLK : out std_logic;
 LOCK : out std_logic
);
end ONE_BYTE_CLK;
--
architecture ONE_BYTE_CLK_arch of ONE_BYTE_CLK is
--
-- Components Declarations:
component BUFG
 port (
 I : in std_logic;
 O : out std_logic
);
end component;
--
component IBUFG
 port (
 I : in std_logic;
 O : out std_logic
);
end component;
--
component DCM
 port (
 CLKIN : in std_logic;
 CLKFB : in std_logic;
 DSSEN : in std_logic;
 PSINCDEC : in std_logic;
 PSEN : in std_logic;
 PSCLK : in std_logic;
 RST : in std_logic;
 CLK0 : out std_logic;
 CLK90 : out std_logic;
 CLK180 : out std_logic;
 CLK270 : out std_logic;
 CLK2X : out std_logic;
 CLK2X180 : out std_logic;
 CLKDV : out std_logic;
 CLKFX : out std_logic;
 CLKFX180 : out std_logic;
 LOCKED : out std_logic;
 PSDONE : out std_logic;
 STATUS : out std_logic_vector (7 downto 0)
);
end component;
--
-- Signal Declarations:
--
signal GND : std_logic;
signal CLK0_W : std_logic;
signal CLK1X_W : std_logic;
signal CLK2X180_W : std_logic;

begin

GND <= '0';
--
CLK1X <= CLK1X_W;
--

```

```
-- DCM Instantiation
U_DCM: DCM
 port map (
 CLKIN => REFCLK,

 CLKFB => USRCLK_M,
 DSSEN => GND,
 PSINCDEC => GND,
 PSEN => GND,
 PSCLK => GND,
 RST => RST,
 CLK0 => CLK0_W,
 CLK2X180 => CLK2X180_W,
 LOCKED => LOCK
);

-- BUFG Instantiation
U_BUFG: IBUFG
 port map (
 I => REFCLKIN,
 O => REFCLK

);

U2_BUFG: BUFG
 port map (
 I => CLK0_W,
 O => USRCLK_M
);

U4_BUFG: BUFG
 port map (
 I => CLK2X180_W,
 O => USRCLK2
);

end ONE_BYTE_CLK_arch;
```

### Verilog Template

```
// Module: ONE_BYTE_CLK
// Description: Verilog Submodule
// DCM for 1-byte GT
//
// Device: Virtex-II Pro Family

module ONE_BYTE_CLK (
 REFCLKIN,
 REFCLK,
 USRCLK_M,
 USRCLK2_M,
 DCM_LOCKED
);

 input REFCLKIN;
 output REFCLK;
 output USRCLK_M;
 output USRCLK2_M;
 output DCM_LOCKED;

 wire REFCLKIN;
 wire REFCLK;
 wire USRCLK_M;
```

```

wire USRCLK2_M;
wire DCM_LOCKED;
wire REFCLKINBUF;
wire clk_i;
wire clk_2x_180;

DCM dcm1 (
 .CLKFB (USRCLK2_M),
 .CLKIN (REFCLKINBUF),

 .DSSEN (1'b0),
 .PCLK (1'b0),
 .PSEN (1'b0),
 .PSINCDEC (1'b0),
 .RST (1'b0),
 .CLK0 (clk_i),
 .CLK90 (),
 .CLK180 (),
 .CLK270 (),
 .CLK2X (),
 .CLK2X180 (clk_2x_180),
 .CLKDV (),
 .CLKFX (),
 .CLKFX180 (),
 .LOCKED (DCM_LOCKED),
 .PSDONE (),
 .STATUS ()
);

BUFG buf1 (
 .I (clk_2x_180),
 .O (USRCLK2_M)
);

BUFG buf2 (
 .I (clk_i),
 .O (USRCLK2_M)
);

IBUFGbuf3 (
 .I (REFCLKIN),
 .O (REFCLKINBUF)
);

endmodule

```

## Multiplexed Clocking Scheme

Following configuration of the FPGA, some applications might need to change the frequency of its REFCLK depending on the protocol used. [Figure 3-4](#) shows how the design can use two different reference clocks connected to two different DCMs. The clocks are then multiplexed before input into the Rocket I/O transceiver.

User logic can be designed to determine during autonegotiation if the reference clock used for the transceiver is incorrect. If so, the transceiver must then be reset and another reference clock selected.

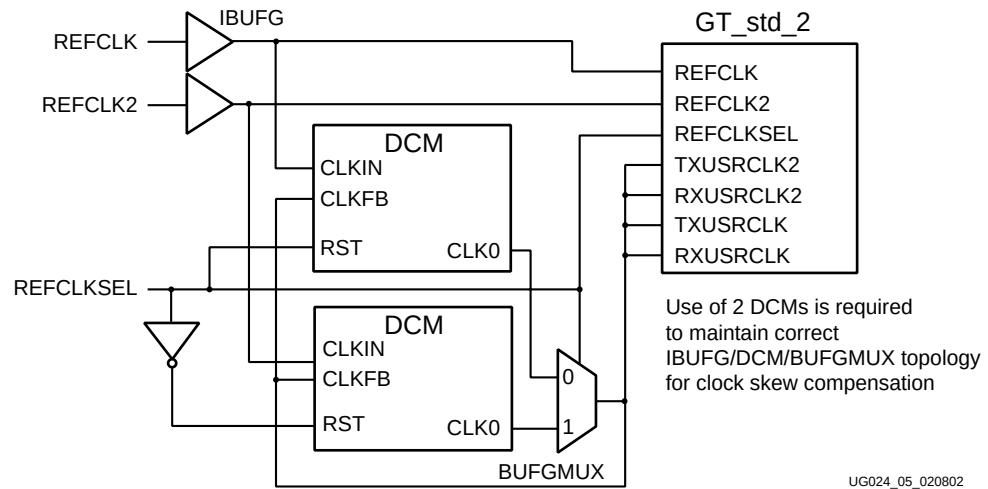


Figure 3-4: Multiplexed REFCLK

## Clock Dependency

All signals used by the FPGA fabric to interact between user logic and the transceiver depend on an edge of USRCLK2. These signals all have setup and hold times with respect to this clock. For specific timing values, see Module 3 of the Virtex-II Pro data sheet.

The timing relationships are illustrated.

[Table 3-9, Parameters Relative to the RX User Clock \(RXUSRCLK\), page 1094](#)

[Table 3-10, Parameters Relative to the RX User Clock2 \(RXUSRCLK2\), page 1094](#)

[Table 3-11, Parameters Relative to the TX User Clock2 \(TXUSRCLK2\), page 1095](#)

[Table 3-12, Miscellaneous Clock Parameters, page 1096](#)

Table 3-9: Parameters Relative to the RX User Clock (RXUSRCLK)

| Parameter                       | Function                      | Signals      |
|---------------------------------|-------------------------------|--------------|
| <b>Setup/Hold:</b>              |                               |              |
| $T_{GCKK\_CHBI}/T_{GCKC\_CHBI}$ | Control inputs                | CHBONDI[3:0] |
| <b>Clock to Out:</b>            |                               |              |
| $T_{GCKCO\_CHBO}$               | Control outputs               | CHBONDO[3:0] |
| <b>Clock:</b>                   |                               |              |
| $T_{RXPWH}$                     | Clock pulse width, High state | RXUSRCLK     |
| $T_{RXPWL}$                     | Clock pulse width, Low state  | RXUSRCLK     |

Table 3-10: Parameters Relative to the RX User Clock2 (RXUSRCLK2)

| Parameter                       | Function      | Signals    |
|---------------------------------|---------------|------------|
| <b>Setup/Hold:</b>              |               |            |
| $T_{GCKK\_RRST}/T_{GCKC\_RRST}$ | Control input | RXRESET    |
| $T_{GCKK\_RPOL}/T_{GCKC\_RPOL}$ | Control input | RXPOLARITY |
| $T_{GCKK\_ECSY}/T_{GCKC\_ECSY}$ | Control input | ENCHANSYNC |
| <b>Clock to Out:</b>            |               |            |

Table 3-10: Parameters Relative to the RX User Clock2 (RXUSRCLK2) (Continued)

| Parameter                  | Function                      | Signals            |
|----------------------------|-------------------------------|--------------------|
| T <sub>GCKST</sub> _RNIT   | Status outputs                | RXNOTINTABLE[3:0]  |
| T <sub>GCKST</sub> _RDERR  | Status outputs                | RXDISPERR[3:0]     |
| T <sub>GCKST</sub> _RCMCH  | Status outputs                | RXCHARISCOMMA[3:0] |
| T <sub>GCKST</sub> _ALIGN  | Status output                 | RXREALIGN          |
| T <sub>GCKST</sub> _CMDT   | Status output                 | RXCOMMADET         |
| T <sub>GCKST</sub> _RLOS   | Status outputs                | RXLOSSOFSYNC[1:0]  |
| T <sub>GCKST</sub> _RCCCNT | Status outputs                | RXCLKCORCNT[2:0]   |
| T <sub>GCKST</sub> _RBSTA  | Status outputs                | RXBUFSTATUS[1:0]   |
| T <sub>GCKST</sub> _RCCRC  | Status output                 | RXCHECKINGCRC      |
| T <sub>GCKST</sub> _RCRCE  | Status output                 | RXCRCERR           |
| T <sub>GCKST</sub> _CHBD   | Status output                 | CHBONDDONE         |
| T <sub>GCKST</sub> _RKCH   | Status outputs                | RXCHARISK[3:0]     |
| T <sub>GCKST</sub> _RRDIS  | Status outputs                | RXRUNDISP[3:0]     |
| T <sub>GCKDO</sub> _RDAT   | Data outputs                  | RXDATA[31:0]       |
| <b>Clock:</b>              |                               |                    |
| T <sub>RX2PWH</sub>        | Clock pulse width, High state | RXUSRCLK2          |
| T <sub>RX2PWH</sub>        | Clock pulse width, Low state  | RXUSRCLK2          |

Table 3-11: Parameters Relative to the TX User Clock2 (TXUSRCLK2)

| Parameter                                         | Function       | Signals             |
|---------------------------------------------------|----------------|---------------------|
| <b>Setup/Hold:</b>                                |                |                     |
| T <sub>GCKK</sub> _CFGEN/T <sub>GCKC</sub> _CFGEN | Control inputs | CONFIGENABLE        |
| T <sub>GCKK</sub> _TBYP/T <sub>GCKC</sub> _TBYP   | Control inputs | TXBYPASS8B10B[3:0]  |
| T <sub>GCKK</sub> _TCRCE/T <sub>GCKC</sub> _TCRCE | Control inputs | TXFORCECRCERR       |
| T <sub>GCKK</sub> _TPOL/T <sub>GCKC</sub> _TPOL   | Control inputs | TXPOLARITY          |
| T <sub>GCKK</sub> _TINH/T <sub>GCKC</sub> _TINH   | Control inputs | TXINHIBIT           |
| T <sub>GCKK</sub> _LBK/T <sub>GCKC</sub> _LBK     | Control inputs | LOOPBACK[1:0]       |
| T <sub>GCKK</sub> _TRST/T <sub>GCKC</sub> _TRST   | Control inputs | TXRESET             |
| T <sub>GCKK</sub> _TKCH/T <sub>GCKC</sub> _TKCH   | Control inputs | TXCHARISK[3:0]      |
| T <sub>GCKK</sub> _TCDM/T <sub>GCKC</sub> _TCDM   | Control inputs | TXCHARDISPMODE[3:0] |
| T <sub>GCKK</sub> _TCDV/T <sub>GCKC</sub> _TCDV   | Control inputs | TXCHARDISPVAL[3:0]  |
| T <sub>GDCK</sub> _CFGIN/T <sub>GCKD</sub> _CFGIN | Data inputs    | CONFIGIN            |
| T <sub>GDCK</sub> _TDAT/T <sub>GCKD</sub> _TDAT   | Data inputs    | TXDATA[31:0]        |
| <b>Clock to Out:</b>                              |                |                     |
| T <sub>GCKST</sub> _TBERR                         | Status outputs | TXBUFERR            |
| T <sub>GCKST</sub> _TKERR                         | Status outputs | TXKERR[3:0]         |



Table 3-11: Parameters Relative to the TX User Clock2 (TXUSRCLK2) (Continued)

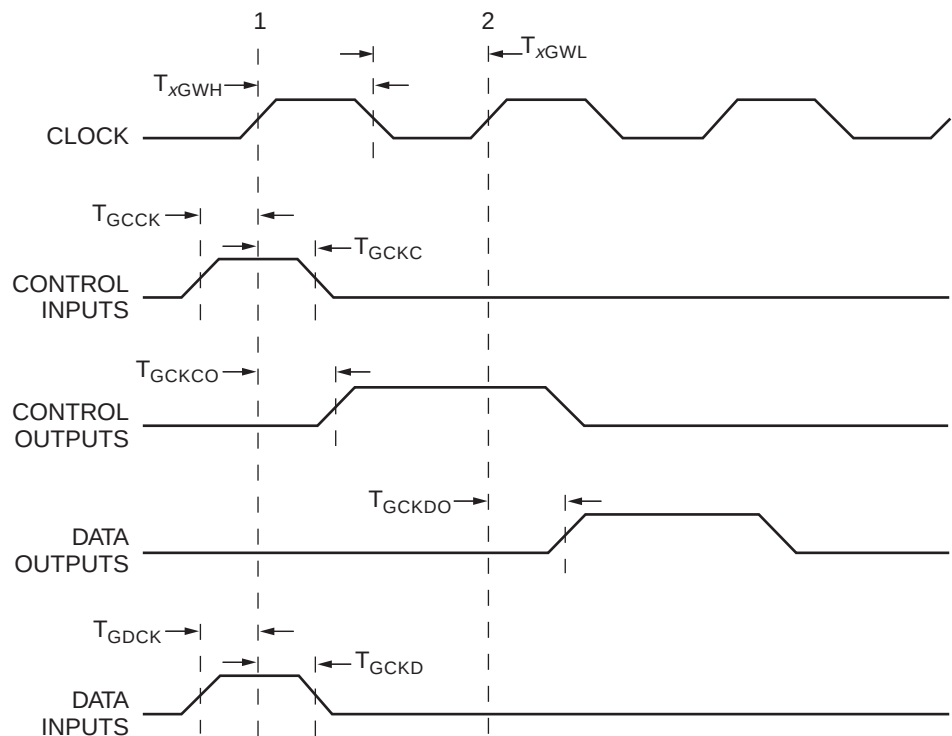
| Parameter           | Function                      | Signals        |
|---------------------|-------------------------------|----------------|
| $T_{GCKDO\_TRDIS}$  | Data outputs                  | TXRUNDISP[3:0] |
| $T_{GCKDO\_CFGOUT}$ | Data outputs                  | CONFIGOUT      |
| <b>Clock:</b>       |                               |                |
| $T_{TX2PWH}$        | Clock pulse width, High state | TXUSRCLK2      |
| $T_{TX2PWL}$        | Clock pulse width, Low state  | TXUSRCLK2      |

Table 3-12: Miscellaneous Clock Parameters

| Parameter     | Function                      | Signals                 |
|---------------|-------------------------------|-------------------------|
| <b>Clock:</b> |                               |                         |
| $T_{REFPWH}$  | Clock pulse width, High state | REFCLK <sup>(1)</sup>   |
| $T_{REFPWL}$  | Clock pulse width, Low state  | REFCLK <sup>(1)</sup>   |
| $T_{TXPWH}$   | Clock pulse width, High state | TXUSRCLK <sup>(2)</sup> |
| $T_{TXPWL}$   | Clock pulse width, Low state  | TXUSRCLK <sup>(2)</sup> |

**Notes:**

1. REFCLK is not synchronous to any Rocket I/O signals.
2. TXUSRCLK is not synchronous to any Rocket I/O signals.



UG012\_106\_02\_100101

Figure 3-5: Rocket I/O Timing Relative to Clock Edge

## Resets

There are two reset signals, one each for the transmit and receive sections of each gigabit transceiver. After the DCM locked signal is asserted, the resets can be asserted. The resets must be asserted for two USRCLK2 cycles to ensure correct initialization of the FIFOs. Although both the transmit and receive resets can be attached to the same signal, separate signals are preferred. This allows the elastic buffer to be cleared in case of an over/underflow without affecting the ongoing TX transmission. The following example is an implementation to reset all three data-width transceivers.

### VHDL Template

```
-- Module: gt_reset
-- Description: VHDL submodule
-- reset for GT
--
-- Device: Virtex-II Pro Family

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.Numeric_STD.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--
-- pragma translate_off
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
-- pragma translate_on
--
entity gt_reset is
port (
 USRCLK2_M : in std_logic;
 LOCK : in std_logic;
 REFCLK : out std_logic;
 DCM_LOCKED: in std_logic;
 RST : out std_logic);
end gt_reset;
--
architecture RTL of gt_reset is
--
 signal startup_count : std_logic_vector (7 downto 0);
begin
 process (USRCLK2_M, DCM_LOCKED)

 begin
 if (USRCLK2_M' event and USRCLK2_M = '1') then
 if(DCM_LOCKED = '0') then
 startup_count <= "00000000";
 elsif (DCM_LOCKED = '1') then
 startup_count <= startup_count + "00000001";
 end if;
 end if;

 if (USRCLK2_M' event and USRCLK2_M = '1') then
 if(DCM_LOCKED = '0') then
 RST <= '1';
 elsif (startup_count = "00000010") then
 RST <= '0';
 end if;
 end if;
 end process;
end process;
```

```
end RTL;
```

### Verilog Template

```
// Module: gt_reset
// Description: Verilog Submodule
// reset for 4-byte GT
//
// Device: Virtex-II Pro Family

module gt_reset(
 USRCLK2_M,
 DCM_LOCKED,
 RST
);

input USRCLK2_M;
input DCM_LOCKED;
input RST;

wire USRCLK2_M;
wire DCM_LOCKED;
reg RST;
reg [7:0] startup_counter;

always @ (posedge USRCLK2_M)
 if (!DCM_LOCKED)
 startup_counter <= 8'h0;
 else if (startup_counter != 8'h02)
 startup_counter <= startup_counter + 1;

always @ (posedge USRCLK2_M or negedge DCM_LOCKED)
 if (!DCM_LOCKED)
 RST <= 1'b1;
 else
 RST <= (startup_counter != 8'h02);

endmodule
```

## Rocket I/O Transceiver Instantiations

For the different clocking schemes, several things must change, including the clock frequency for USRCLK and USRCLK2 discussed in the DCM section above. The data and control ports for GT\_CUSTOM must also reflect this change in data width by concatenating zeros onto inputs and wires for outputs for Verilog designs, and by setting outputs to open and concatenating zeros on unused input bits for VHDL designs.

### HDL Code Examples

Availability of downloadable GT\_CUSTOM code examples with 1-, 2-, and 4-byte data widths is planned for a later date.

## PLL Operation and Clock Recovery

The clock correction sequence is a special sequence to accommodate frequency differences between the received data (as reflected in RXRECCLK) and RXUSRCLK. Most of the primitives have these defaulted to the respective protocols. Only the GT\_CUSTOM allows this sequence to be set to any specific protocol. The sequence contains 11 bits including the 10 bits of serial data. The 11th bit has two different formats. The typical usage is:

0, disparity error required, char is K, 8-bit data value (after 8B/10B decoding)

1, 10 bit data value (without 8B/10B decoding)

Table 3-13 is an example of data 11-bit attribute setting, the character value, CHARISK value, and the parallel data interface, and how each corresponds with the other.

Table 3-13: Clock Correction Sequence / Data Correlation for 16-Bit Data Port

| Attribute Setting             | Character | CHARISK | TXDATA (hex) |
|-------------------------------|-----------|---------|--------------|
| CLK_COR_SEQ_1_1 = 00110111100 | K28.5     | 1       | BC           |
| CLK_COR_SEQ_1_2 = 00010010101 | D21.4     | 0       | 95           |
| CLK_COR_SEQ_1_3 = 00010110101 | D21.5     | 0       | B5           |
| CLK_COR_SEQ_1_4 = 00010110101 | D21.5     | 0       | B5           |

The GT\_CUSTOM transceiver examples use the previous sequence for clock correction.

## Clock Correction Count

The clock correction count signal (RXCLKCORCNT) is a three-bit signal. It signals if the clock correction has occurred and whether the elastic buffer realigned the data by skipping or repeating data in the buffer. It also signals if channel bonding has occurred (Table 3-14).

Table 3-14: RXCLKCORCNT Definition

| RXCLKCORCNT[2:0] | Significance                                                              |
|------------------|---------------------------------------------------------------------------|
| 000              | No channel bonding or clock correction occurred for current RXDATA        |
| 001              | Elastic buffer skipped one clock correction sequence for current RXDATA   |
| 010              | Elastic buffer skipped two clock correction sequence for current RXDATA   |
| 011              | Elastic buffer skipped three clock correction sequence for current RXDATA |
| 100              | Elastic buffer skipped four clock correction sequence for current RXDATA  |
| 101              | Elastic buffer executed channel bonding for current RXDATA                |
| 110              | Elastic buffer repeated two clock correction sequences for current RXDATA |
| 111              | Elastic buffer repeated one clock correction sequences for current RXDATA |

## RX\_LOSS\_OF\_SYNC\_FSM

The transceivers FSM is driven by RXRECLK and uses status from the data stream prior to the elastic buffer. This is intended to give early warning of possible problems well before corrupt data appears on RXDATA. There are three states.

### SYNC\_ACQUIRED (RXLOSSOFFSYNC = 00)

In this state, a counter is decremented by 1 (but not past 0) for a valid received symbol and incremented by RX\_LOS\_INVALID\_INCR for an invalid symbol. If the count reaches or exceeds RX\_LOS\_THRESHOLD, the FSM moves to state LOSS\_OF\_SYNC. Otherwise, if a

channel bonding (alignment) sequence has just been written into the elastic buffer, or if a comma realignment has just occurred, the FSM moves to state RESYNC. Otherwise, the FSM remains in state SYNC\_ACQUIRED.

### RESYNC (RXLOSSOFSYNC = 01)

The FSM waits in this state for four RXRECCLK cycles and then goes to state SYNC\_ACQUIRED, unless an invalid symbol is received, in which case the FSM goes to state LOSS\_OF\_SYNC.

### LOSS\_OF\_SYNC (RXLOSSOFSYNC = 10)

The FSM remains in this state until a comma is received, at which time it goes to state RESYNC. The FSM state appears on RXLOSSOFSYNC if RX\_LOSS\_OF\_SYNC\_FSM is TRUE. Otherwise, RXLOSSOFSYNC[1] is high if an invalid byte (not in table, or with disparity error) is received, and RXLOSSOFSYNC[0] is High when a channel bonding (alignment) sequence has just been written into the elastic buffer.

## 8B/10B Operation

The Rocket I/O transceiver has the ability to encode eight bits into a 10-bit serial stream using standard 8B/10B encoding. This guarantees a DC-balanced, edge-rich serial stream, facilitating DC- or AC-coupling and clock recovery. If the 8B/10B encoding is disabled, the data is sent through in 10-bit blocks. The 8B/10B-bypass signal is set to 1111 and the RX\_DECODE\_USE attribute must be set to FALSE. If the 8B/10B encoding is needed, the bypass signal must be set to 0000 and the RX\_DECODE\_USE must be set to TRUE. If this pair is not matched, the data is not received correctly. Figure 3-6 shows the encoding/decoding blocks of the transceiver and how the data passes through these blocks. Table 3-15, page 1101, shows the significance of 8B/10B ports that change purpose depending on whether 8B/10B is bypassed or enabled.

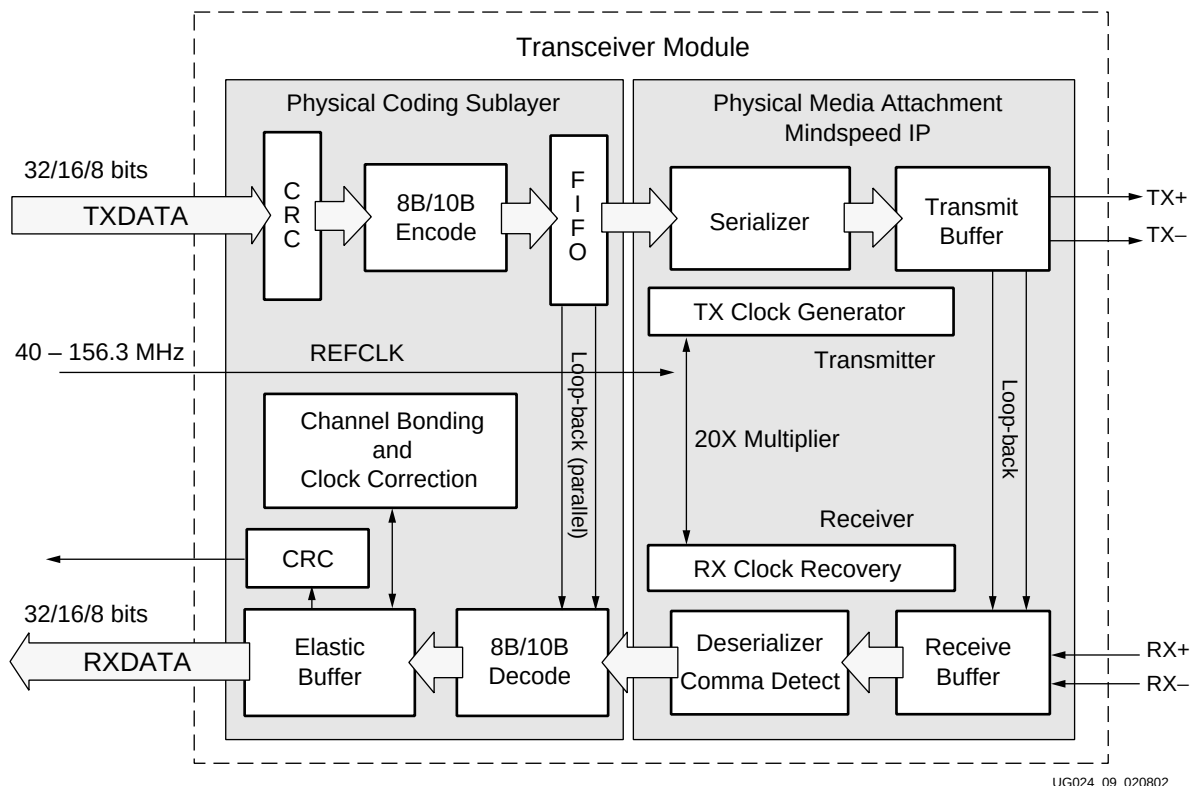


Figure 3-6: 8B/10B Data Flow

Table 3-15: 8B/10B Bypassed Signal Significance

|                |   | 8B/10B Enabled                          | 8B/10B Bypassed                                                                                         |
|----------------|---|-----------------------------------------|---------------------------------------------------------------------------------------------------------|
| TXBYPASS8B10B  | 1 | N/A                                     | Bypassed 8B/10B encoding                                                                                |
|                | 0 | Enabled 8B/10B                          | N/A                                                                                                     |
| TXCHARISK      | 1 | Transmitted byte is a K-character       | Unused                                                                                                  |
|                | 0 | Transmitted byte is a data character    | Unused                                                                                                  |
| TXCHARDISPMODE |   | Disparity generation control            | Part of 10-bit encoded byte: TXCHARDISPMODE[1], TXCHARDISPVAL[1], TXDATA[15:8], similar for other bytes |
| TXCHARDISPVAL  |   | Disparity generation control            | Part of 10-bit encoded byte: TXCHARDISPMODE[1], TXCHARDISPVAL[1], TXDATA[15:8], similar for other bytes |
| RXCHARISK      |   | Indicates if character is a K-character | Part of 10-bit encoded byte: RXCHARISK[1], RXRUNDISP[1], RXDATA[15:8], similar for other bytes          |
| RXRUNDISP      |   | Indicates running disparity             | Part of 10-bit encoded byte: RXCHARISK[1], RXRUNDISP[1], RXDATA[15:8], similar for other bytes          |

While the 8B/10B is enabled, the disparity of the serial transmission can be controlled with TXCHARDISPMODE and TXCHARDISPVAL. This is explained in [Table 3-16](#). During the bypassing of the 8B/10B encoding, these ports become part of the 10-bit encoded data that the transceiver must transmit. See [Figure 3-7](#) and [Figure 3-8](#) for TX and RX data maps during 8B/10B bypass.

Table 3-16: Running Disparity Modes with 8B/10B Enabled

| {TXCHARDISPMODE, TXCHARDISPVAL} | Function                                                                   |
|---------------------------------|----------------------------------------------------------------------------|
| 00                              | Maintain running disparity normally.                                       |
| 01                              | Invert the normally generated running disparity before encoding this byte. |
| 10                              | Set negative running disparity before encoding this byte.                  |
| 11                              | Set positive running disparity before encoding this byte.                  |

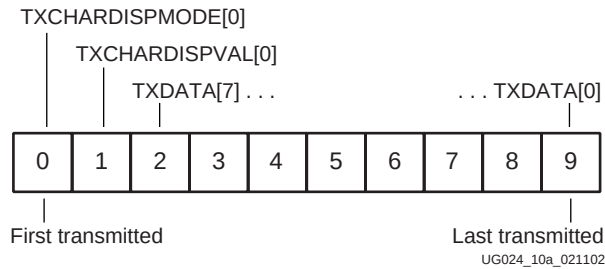


Figure 3-7: 10-Bit TX Data Map with 8B/10B Bypassed

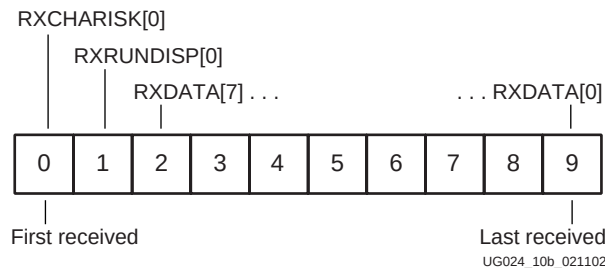


Figure 3-8: 10-Bit RX Data Map with 8B/10B Bypassed

## Vitesse Disparity Example

To support other protocols, the transceiver can affect the disparity mode of the serial data transmitted. For example, Vitesse channel-to-channel alignment protocol sends out:

K28.5+ K28.5+ K28.5- K28.5-

or

K28.5- K28.5- K28.5+ K28.5+

Instead of:

K28.5+ K28.5- K28.5+ K28.5-

or

K28.5- K28.5+ K28.5- K28.5+

The logic must assert TXCHARDISPVAL to cause the serial data to send out two negative running disparity characters.

## Transmitting Vitesse Channel Bonding Sequence

```

TXBYPASS8B10B
| TXCHARISK
| | TXCHARDISPMODE
| | | TXCHARDISPVAL
| | | | TXDATA
| | | | |
0 1 0 0 10111100 K28.5+ (or K28.5-)
0 1 0 1 10111100 K28.5+ (or K28.5-)
0 1 0 0 10111100 K28.5- (or K28.5+)
0 1 0 1 10111100 K28.5- (or K28.5+)

```

The Rocket I/O core receives this data but must have the CHAN\_BOND\_SEQ set with the disp\_err bit set High for the cases when TXCHARDISPVAL is set High during data transmission.

## Receiving Vitesse Channel Bonding Sequence

On the RX side, the definition of the channel bonding sequence uses the `disp_err` bit to specify the flipped disparity.

```

 10-bit literal value
 | disp_err
 | | char_is_k
 | | | 8-bit_byte_value
 | | | |
CHAN_BOND_SEQ_1_1 = 0 0 1 10111100 matches K28.5+ (or K28.5-)
CHAN_BOND_SEQ_1_2 = 0 1 1 10111100 matches K28.5+ (or K28.5-)
CHAN_BOND_SEQ_1_3 = 0 0 1 10111100 matches K28.5- (or K28.5+)
CHAN_BOND_SEQ_1_4 = 0 1 1 10111100 matches K28.5- (or K28.5+)
CHAN_BOND_SEQ_LEN = 4
CHAN_BOND_SEQ_2_USE = False

```

## Status Signals

Whether the 8B/10B encoding is enabled or disabled, there are several status signals for error indication. If an invalid K-character is sent to the transceiver, the `TXKERR` transitions High. This can produce several receive errors. These receive errors are `RXNOTINTABLE` or `RUNDISPERR`. `RUNDISPERR` transitions High if the incorrect disparity is received. `RXNOTINTABLE` determines if the incoming character is a valid character. These signals are meant to detect errors in the transmission data from incorrect framing. The **CRC Operation** section covers transmission data error detection caused by noise.

## 8B/10B Encoding

8B/10B encoding includes a set of Data characters and K-characters. Eight-bit values are coded into 10-bit values keeping the serial line DC balanced. K-characters are special Data characters designated with a `CHARISK`. K-characters are used for specific informative designations. [Table 3-17](#) and [Table 3-18](#) show the Data and K tables of valid characters.

Table 3-17: Valid Data Characters

| Data Byte Name | Bits HGF EDCBA | Current RD – abcdei fghj | Current RD + abcdei fghj |
|----------------|----------------|--------------------------|--------------------------|
| D0.0           | 000 00000      | 100111 0100              | 011000 1011              |
| D1.0           | 000 00001      | 011101 0100              | 100010 1011              |
| D2.0           | 000 00010      | 101101 0100              | 010010 1011              |
| D3.0           | 000 00011      | 110001 1011              | 110001 0100              |
| D4.0           | 000 00100      | 110101 0100              | 001010 1011              |
| D5.0           | 000 00101      | 101001 1011              | 101011 0100              |
| D6.0           | 000 00110      | 011001 1011              | 011001 0100              |
| D7.0           | 000 00111      | 111000 1011              | 000111 0100              |
| D8.0           | 000 01000      | 111001 0100              | 000110 1011              |
| D9.0           | 000 01001      | 100101 1011              | 011010 0100              |
| D10.0          | 000 01010      | 010101 1011              | 010101 0100              |
| D11.0          | 000 01011      | 110100 1011              | 110100 0100              |



Table 3-17: Valid Data Characters (Continued)

| Data Byte Name | Bits<br>HGF EDCBA | Current RD –<br>abcdei fghj | Current RD +<br>abcdei fghj |
|----------------|-------------------|-----------------------------|-----------------------------|
| D12.0          | 000 01100         | 001101 1011                 | 001101 0100                 |
| D13.0          | 000 01101         | 101100 1011                 | 101100 0100                 |
| D14.0          | 000 01110         | 011100 1011                 | 011100 0100                 |
| D15.0          | 000 01111         | 010111 0100                 | 101000 1011                 |
| D16.0          | 000 10000         | 011011 0100                 | 100100 1011                 |
| D17.0          | 000 10001         | 100011 1011                 | 100011 0100                 |
| D18.0          | 000 10010         | 010011 1011                 | 010011 0100                 |
| D19.0          | 000 10011         | 110010 1011                 | 110010 0100                 |
| D20.0          | 000 10100         | 001011 1011                 | 001011 0100                 |
| D21.0          | 000 10101         | 101010 1011                 | 101010 0100                 |
| D22.0          | 000 10110         | 011010 1011                 | 011010 0100                 |
| D23.0          | 000 10111         | 111010 0100                 | 000101 1011                 |
| D24.0          | 000 11000         | 110011 0100                 | 001100 1011                 |
| D25.0          | 000 11001         | 100110 1011                 | 100110 0100                 |
| D26.0          | 000 11010         | 010110 1011                 | 010110 0100                 |
| D27.0          | 000 11011         | 110110 0100                 | 001001 1011                 |
| D28.0          | 000 11100         | 001110 1011                 | 001110 0100                 |
| D29.0          | 000 11101         | 101110 0100                 | 010001 1011                 |
| D30.0          | 000 11110         | 011110 0100                 | 100001 1011                 |
| D31.0          | 000 11111         | 101011 0100                 | 010100 1011                 |
| D0.1           | 001 00000         | 100111 1001                 | 011000 1001                 |
| D1.1           | 001 00001         | 011101 1001                 | 100010 1001                 |
| D2.1           | 001 00010         | 101101 1001                 | 010010 1001                 |
| D3.1           | 001 00011         | 110001 1001                 | 110001 1001                 |
| D4.1           | 001 00100         | 110101 1001                 | 001010 1001                 |
| D5.1           | 001 00101         | 101001 1001                 | 101011 1001                 |
| D6.1           | 001 00110         | 011001 1001                 | 011001 1001                 |
| D7.1           | 001 00111         | 111000 1001                 | 000111 1001                 |
| D8.1           | 001 01000         | 111001 1001                 | 000110 1001                 |
| D9.1           | 001 01001         | 100101 1001                 | 011010 1001                 |
| D10.1          | 001 01010         | 010101 1001                 | 010101 1001                 |
| D11.1          | 001 01011         | 110100 1001                 | 110100 1001                 |
| D12.1          | 001 01100         | 001101 1001                 | 001101 1001                 |

Table 3-17: Valid Data Characters (Continued)

| Data Byte Name | Bits<br>HGF EDCBA | Current RD –<br>abcdei fghj | Current RD +<br>abcdei fghj |
|----------------|-------------------|-----------------------------|-----------------------------|
| D13.1          | 001 01101         | 101100 1001                 | 101100 1001                 |
| D14.1          | 001 01110         | 011100 1001                 | 011100 1001                 |
| D15.1          | 001 01111         | 010111 1001                 | 101000 1001                 |
| D16.1          | 001 10000         | 011011 1001                 | 100100 1001                 |
| D17.1          | 001 10001         | 100011 1001                 | 100011 1001                 |
| D18.1          | 001 10010         | 010011 1001                 | 010011 1001                 |
| D19.1          | 001 10011         | 110010 1001                 | 110010 1001                 |
| D20.1          | 001 10100         | 001011 1001                 | 001011 1001                 |
| D21.1          | 001 10101         | 101010 1001                 | 101010 1001                 |
| D22.1          | 001 10110         | 011010 1001                 | 011010 1001                 |
| D23.1          | 001 10111         | 111010 1001                 | 000101 1001                 |
| D24.1          | 001 11000         | 110011 1001                 | 001100 1001                 |
| D25.1          | 001 11001         | 100110 1001                 | 100110 1001                 |
| D26.1          | 001 11010         | 010010 1001                 | 010110 1001                 |
| D27.1          | 001 11011         | 110110 1001                 | 001001 1001                 |
| D28.1          | 001 11100         | 001110 1001                 | 001110 1001                 |
| D29.1          | 001 11101         | 101110 1001                 | 010001 1001                 |
| D30.1          | 001 11110         | 011110 1001                 | 100001 1001                 |
| D31.1          | 001 11111         | 101011 1001                 | 010100 1001                 |
| D0.2           | 010 00000         | 100111 0101                 | 011000 0101                 |
| D1.2           | 010 00001         | 011101 0101                 | 100010 0101                 |
| D2.2           | 010 00010         | 101101 0101                 | 010010 0101                 |
| D3.2           | 010 00011         | 110001 0101                 | 110001 0101                 |
| D4.2           | 010 00100         | 110101 0101                 | 001010 0101                 |
| D5.2           | 010 00101         | 101001 0101                 | 101011 0101                 |
| D6.2           | 010 00110         | 011001 0101                 | 011001 0101                 |
| D7.2           | 010 00111         | 111000 0101                 | 000111 0101                 |
| D8.2           | 010 01000         | 111001 0101                 | 000110 0101                 |
| D9.2           | 010 01001         | 100101 0101                 | 011010 0101                 |
| D10.2          | 010 01010         | 010101 0101                 | 010101 0101                 |
| D11.2          | 010 01011         | 110100 0101                 | 110100 0101                 |
| D12.2          | 010 01100         | 001101 0101                 | 001101 0101                 |
| D13.2          | 010 01101         | 101100 0101                 | 101100 0101                 |

Table 3-17: Valid Data Characters (Continued)

| Data Byte Name | Bits<br>HGF EDCBA | Current RD –<br>abcdei fghj | Current RD +<br>abcdei fghj |
|----------------|-------------------|-----------------------------|-----------------------------|
| D14.2          | 010 01110         | 011100 0101                 | 011100 0101                 |
| D15.2          | 010 01111         | 010111 0101                 | 101000 0101                 |
| D16.2          | 010 10000         | 011011 0101                 | 100100 0101                 |
| D17.2          | 010 10001         | 100011 0101                 | 100011 0101                 |
| D18.2          | 010 01010         | 010011 0101                 | 010011 0101                 |
| D19.2          | 010 10011         | 110010 0101                 | 110010 0101                 |
| D20.2          | 010 10100         | 001011 0101                 | 001011 0101                 |
| D21.2          | 010 10101         | 101010 0101                 | 101010 0101                 |
| D22.2          | 010 10110         | 011010 0101                 | 011010 0101                 |
| D23.2          | 010 10111         | 111010 0101                 | 000101 0101                 |
| D24.2          | 010 11000         | 110011 0101                 | 001100 0101                 |
| D25.2          | 010 11001         | 100110 0101                 | 100110 0101                 |
| D26.2          | 010 11010         | 010010 0101                 | 010110 0101                 |
| D27.2          | 010 11011         | 110110 0101                 | 001001 0101                 |
| D28.2          | 010 11100         | 001110 0101                 | 001110 0101                 |
| D29.2          | 010 11101         | 101110 0101                 | 010001 0101                 |
| D30.2          | 010 11110         | 011110 0101                 | 100001 0101                 |
| D31.2          | 010 11111         | 101011 0101                 | 010100 0101                 |
| D0.3           | 000 00000         | 100111 0011                 | 011000 1100                 |
| D1.3           | 011 00001         | 011101 0011                 | 100010 1100                 |
| D2.3           | 011 00010         | 101101 0011                 | 010010 1100                 |
| D3.3           | 011 00011         | 110001 1100                 | 110001 0011                 |
| D4.3           | 011 00100         | 110101 0011                 | 001010 1100                 |
| D5.3           | 011 00101         | 101001 1100                 | 101011 0011                 |
| D6.3           | 011 00110         | 011001 1100                 | 011001 0011                 |
| D7.3           | 011 00111         | 111000 1100                 | 000111 0011                 |
| D8.3           | 011 01000         | 111001 0011                 | 000110 1100                 |
| D9.3           | 011 01001         | 100101 1100                 | 011010 0011                 |
| D10.3          | 011 01010         | 010101 1100                 | 010101 0011                 |
| D11.3          | 011 01011         | 110100 1100                 | 110100 0011                 |
| D12.3          | 011 01100         | 001101 1100                 | 001101 0011                 |
| D13.3          | 011 01101         | 101100 1100                 | 101100 0011                 |
| D14.3          | 011 01110         | 011100 1100                 | 011100 0011                 |

Table 3-17: Valid Data Characters (Continued)

| Data Byte Name | Bits<br>HGF EDCBA | Current RD –<br>abcdei fghj | Current RD +<br>abcdei fghj |
|----------------|-------------------|-----------------------------|-----------------------------|
| D15.3          | 011 01111         | 010111 0011                 | 101000 1100                 |
| D16.3          | 011 10000         | 011011 0011                 | 100100 1100                 |
| D17.3          | 011 10001         | 100011 1100                 | 100011 0011                 |
| D18.3          | 011 10010         | 010011 1100                 | 010011 0011                 |
| D19.3          | 011 10011         | 110010 1100                 | 110010 0011                 |
| D20.3          | 011 10100         | 001011 1100                 | 001011 0011                 |
| D21.3          | 011 10101         | 101010 1100                 | 101010 0011                 |
| D22.3          | 011 10110         | 011010 1100                 | 011010 0011                 |
| D23.3          | 011 10111         | 111010 0011                 | 000101 1100                 |
| D24.3          | 011 11000         | 110011 0011                 | 001100 1100                 |
| D25.3          | 011 11001         | 100110 1100                 | 100110 0011                 |
| D26.3          | 011 11010         | 010110 1100                 | 010110 0011                 |
| D27.3          | 011 11011         | 110110 0011                 | 001001 1100                 |
| D28.3          | 011 11100         | 001110 1100                 | 001110 0011                 |
| D29.3          | 011 11101         | 101110 0011                 | 010001 1100                 |
| D30.3          | 011 11110         | 011110 0011                 | 100001 1100                 |
| D31.3          | 011 11111         | 101011 0011                 | 010100 1100                 |
| D0.4           | 100 00000         | 100111 0010                 | 011000 1101                 |
| D1.4           | 100 00001         | 011101 0010                 | 100010 1101                 |
| D2.4           | 100 00010         | 101101 0010                 | 010010 1101                 |
| D3.4           | 100 00011         | 110001 1101                 | 110001 0010                 |
| D4.4           | 100 00100         | 110101 0010                 | 001010 1101                 |
| D5.4           | 100 00101         | 101001 1101                 | 101011 0010                 |
| D6.4           | 100 00110         | 011001 1101                 | 011001 0010                 |
| D7.4           | 100 00111         | 111000 1101                 | 000111 0010                 |
| D8.4           | 100 01000         | 111001 0010                 | 000110 1101                 |
| D9.4           | 100 01001         | 100101 1101                 | 011010 0010                 |
| D10.4          | 100 01010         | 010101 1101                 | 010101 0010                 |
| D11.4          | 100 01011         | 110100 1101                 | 110100 0010                 |
| D12.4          | 100 01100         | 001101 1101                 | 001101 0010                 |
| D13.4          | 100 01101         | 101100 1101                 | 101100 0010                 |
| D14.4          | 100 01110         | 011100 1101                 | 011100 0010                 |
| D15.4          | 100 01111         | 010111 0010                 | 101000 1101                 |

Table 3-17: Valid Data Characters (Continued)

| Data Byte Name | Bits<br>HGF EDCBA | Current RD –<br>abcdei fghj | Current RD +<br>abcdei fghj |
|----------------|-------------------|-----------------------------|-----------------------------|
| D16.4          | 100 10000         | 011011 0010                 | 100100 1101                 |
| D17.4          | 100 10001         | 100011 1101                 | 100011 0010                 |
| D18.4          | 100 10010         | 010011 1101                 | 010011 0010                 |
| D19.4          | 100 10011         | 110010 1101                 | 110010 0010                 |
| D20.4          | 100 10100         | 001011 1101                 | 001011 0010                 |
| D21.4          | 100 10101         | 101010 1101                 | 101010 0010                 |
| D22.4          | 100 10110         | 011010 1101                 | 011010 0010                 |
| D23.4          | 100 10111         | 111010 0010                 | 000101 1101                 |
| D24.4          | 100 11000         | 110011 0010                 | 001100 1101                 |
| D25.4          | 100 11001         | 100110 1101                 | 100110 0010                 |
| D26.4          | 100 11010         | 010010 1101                 | 010110 0010                 |
| D27.4          | 100 11011         | 110110 0010                 | 001001 1101                 |
| D28.4          | 100 11100         | 001110 1101                 | 001110 0010                 |
| D29.4          | 100 11101         | 101110 0010                 | 010001 1101                 |
| D30.4          | 100 11110         | 011110 0010                 | 100001 1101                 |
| D31.4          | 100 11111         | 101011 0010                 | 010100 1101                 |
| D0.5           | 101 00000         | 100111 1010                 | 011000 1010                 |
| D1.5           | 101 00001         | 011101 1010                 | 100010 1010                 |
| D2.5           | 101 00010         | 101101 1010                 | 010010 1010                 |
| D3.5           | 101 00011         | 110001 1010                 | 110001 1010                 |
| D4.5           | 101 00100         | 110101 101                  | 001010 1010                 |
| D5.5           | 101 00101         | 101001 1010                 | 101011 1010                 |
| D6.5           | 101 00110         | 011001 1010                 | 011001 1010                 |
| D7.5           | 101 00111         | 111000 1010                 | 000111 1010                 |
| D8.5           | 101 01000         | 111001 1010                 | 000110 1010                 |
| D9.5           | 101 01001         | 100101 1010                 | 011010 1010                 |
| D10.5          | 101 01010         | 010101 1010                 | 010101 1010                 |
| D11.5          | 101 01011         | 110100 1010                 | 110100 1010                 |
| D12.5          | 101 01100         | 001101 1010                 | 001101 1010                 |
| D13.5          | 101 01101         | 101100 1010                 | 101100 1010                 |
| D14.5          | 101 01110         | 011100 1010                 | 011100 1010                 |
| D15.5          | 101 01111         | 010111 1010                 | 101000 1010                 |
| D16.5          | 101 10000         | 011011 1010                 | 100100 1010                 |

Table 3-17: Valid Data Characters (Continued)

| Data Byte Name | Bits<br>HGF EDCBA | Current RD –<br>abcdei fghj | Current RD +<br>abcdei fghj |
|----------------|-------------------|-----------------------------|-----------------------------|
| D17.5          | 101 10001         | 100011 1010                 | 100011 1010                 |
| D18.5          | 101 01010         | 010011 1010                 | 010011 1010                 |
| D19.5          | 101 10011         | 110010 1010                 | 110010 1010                 |
| D20.5          | 101 10100         | 001011 1010                 | 001011 1010                 |
| D21.5          | 101 10101         | 101010 1010                 | 101010 1010                 |
| D22.5          | 101 10110         | 011010 1010                 | 011010 1010                 |
| D23.5          | 101 10111         | 111010 1010                 | 000101 1010                 |
| D24.5          | 101 11000         | 110011 1010                 | 001100 1010                 |
| D25.5          | 101 11001         | 100110 1010                 | 100110 1010                 |
| D26.5          | 101 11010         | 010010 1010                 | 010110 1010                 |
| D27.5          | 101 11011         | 110110 1010                 | 001001 1010                 |
| D28.5          | 101 11100         | 001110 1010                 | 001110 1010                 |
| D29.5          | 101 11101         | 101110 1010                 | 010001 1010                 |
| D30.5          | 101 11110         | 011110 1010                 | 100001 1010                 |
| D31.5          | 101 11111         | 101011 1010                 | 010100 1010                 |
| D0.6           | 110 00000         | 100111 0110                 | 011000 0110                 |
| D1.6           | 110 00001         | 011101 0110                 | 100010 0110                 |
| D2.6           | 110 00010         | 101101 0110                 | 010010 0110                 |
| D3.6           | 110 00011         | 110001 0110                 | 110001 0110                 |
| D4.6           | 110 00100         | 110101 0110                 | 001010 0110                 |
| D5.6           | 110 00101         | 101001 0110                 | 101011 0110                 |
| D6.6           | 110 00110         | 011001 0110                 | 011001 0110                 |
| D7.6           | 110 00111         | 111000 0110                 | 000111 0110                 |
| D8.6           | 110 01000         | 111001 0110                 | 000110 0110                 |
| D9.6           | 110 01001         | 100101 0110                 | 011010 0110                 |
| D10.6          | 110 01010         | 010101 0110                 | 010101 0110                 |
| D11.6          | 110 01011         | 110100 0110                 | 110100 0110                 |
| D12.6          | 110 01100         | 001101 0110                 | 001101 0110                 |
| D13.6          | 110 01101         | 101100 0110                 | 101100 0110                 |
| D14.6          | 110 01110         | 011100 0110                 | 011100 0110                 |
| D15.6          | 110 01111         | 010111 0110                 | 101000 0110                 |
| D16.6          | 110 10000         | 011011 0110                 | 100100 0110                 |
| D17.6          | 110 10001         | 100011 0110                 | 100011 0110                 |

Table 3-17: Valid Data Characters (Continued)

| Data Byte Name | Bits<br>HGF EDCBA | Current RD –<br>abcdei fghj | Current RD +<br>abcdei fghj |
|----------------|-------------------|-----------------------------|-----------------------------|
| D18.6          | 110 01010         | 010011 0110                 | 010011 0110                 |
| D19.6          | 110 10011         | 110010 0110                 | 110010 0110                 |
| D20.6          | 110 10100         | 001011 0110                 | 001011 0110                 |
| D21.6          | 110 10101         | 101010 0110                 | 101010 0110                 |
| D22.6          | 110 10110         | 011010 0110                 | 011010 0110                 |
| D23.6          | 110 10111         | 111010 0110                 | 000101 0110                 |
| D24.6          | 110 11000         | 110011 0110                 | 001100 0110                 |
| D25.6          | 110 11001         | 100110 0110                 | 100110 0110                 |
| D26.6          | 110 11010         | 010010 0110                 | 010110 0110                 |
| D27.6          | 110 11011         | 110110 0110                 | 001001 0110                 |
| D28.6          | 110 11100         | 001110 0110                 | 001110 0110                 |
| D29.6          | 110 11101         | 101110 0110                 | 010001 0110                 |
| D30.6          | 110 11110         | 011110 0110                 | 100001 0110                 |
| D31.6          | 110 11111         | 101011 0110                 | 010100 0110                 |
| D0.7           | 111 00000         | 100111 0001                 | 011000 1110                 |
| D1.7           | 111 00001         | 011101 0001                 | 100010 1110                 |
| D2.7           | 111 00010         | 101101 0001                 | 010010 1110                 |
| D3.7           | 111 00011         | 110001 1110                 | 110001 0001                 |
| D4.7           | 111 00100         | 110101 0001                 | 001010 1110                 |
| D5.7           | 111 00101         | 101001 1110                 | 101011 0001                 |
| D6.7           | 111 00110         | 011001 1110                 | 011001 0001                 |
| D7.7           | 111 00111         | 111000 1110                 | 000111 0001                 |
| D8.7           | 111 01000         | 111001 0001                 | 000110 1110                 |
| D9.7           | 111 01001         | 100101 1110                 | 011010 0001                 |
| D10.7          | 111 01010         | 010101 1110                 | 010101 0001                 |
| D11.7          | 111 01011         | 110100 1110                 | 110100 1000                 |
| D12.7          | 111 01100         | 001101 1110                 | 001101 0001                 |
| D13.7          | 111 01101         | 101100 1110                 | 101100 1000                 |
| D14.7          | 111 01110         | 011100 1110                 | 011100 1000                 |
| D15.7          | 111 01111         | 010111 0001                 | 101000 1110                 |
| D16.7          | 111 10000         | 011011 0001                 | 100100 1110                 |
| D17.7          | 111 10001         | 100011 0111                 | 100011 0001                 |
| D18.7          | 111 10010         | 010011 0111                 | 010011 0001                 |

Table 3-17: Valid Data Characters (Continued)

| Data Byte Name | Bits<br>HGF EDCBA | Current RD –<br>abcdei fghj | Current RD +<br>abcdei fghj |
|----------------|-------------------|-----------------------------|-----------------------------|
| D19.7          | 111 10011         | 110010 1110                 | 110010 0001                 |
| D20.7          | 111 10100         | 001011 0111                 | 001011 0001                 |
| D21.7          | 111 10101         | 101010 1110                 | 101010 0001                 |
| D22.7          | 111 10110         | 011010 1110                 | 011010 0001                 |
| D23.7          | 111 10111         | 111010 0001                 | 000101 1110                 |
| D24.7          | 111 11000         | 110011 0001                 | 001100 1110                 |
| D25.7          | 111 11001         | 100110 1110                 | 100110 0001                 |
| D26.7          | 111 11010         | 010110 1110                 | 010110 0001                 |
| D27.7          | 111 11011         | 110110 0001                 | 001001 1110                 |
| D28.7          | 111 11100         | 001110 1110                 | 001110 0001                 |
| D29.7          | 111 11101         | 101110 0001                 | 010001 1110                 |
| D30.7          | 111 11110         | 011110 0001                 | 100001 1110                 |
| D31.7          | 111 11111         | 101011 0001                 | 010100 1110                 |



Table 3-18: Valid Control “K” Characters

| Special Code Name | Bits HGF EDCBA | Current RD – abcdei fghj | Current RD + abcdei fghj |
|-------------------|----------------|--------------------------|--------------------------|
| K28.0             | 000 11100      | 001111 0100              | 110000 1011              |
| K28.1             | 001 11100      | 001111 1001              | 110000 0110              |
| K28.2             | 010 11100      | 001111 0101              | 110000 1010              |
| K28.3             | 011 11100      | 001111 0011              | 110000 1100              |
| K28.4             | 100 11100      | 001111 0010              | 110000 1101              |
| K28.5             | 101 11100      | 001111 1010              | 110000 0101              |
| K28.6             | 110 11100      | 001111 0110              | 110000 1001              |
| K28.7             | 111 11100      | 001111 1000              | 110000 0111              |
| K23.7             | 111 10111      | 111010 1000              | 000101 0111              |
| K27.7             | 111 11011      | 110110 1000              | 001001 0111              |
| K29.7             | 111 11101      | 101110 1000              | 010001 0111              |
| K30.7             | 111 11110      | 011110 1000              | 100001 0111              |

## 8B/10B Serial Output Format

The 8B/10B encoding translates a 8-bit parallel data byte to be transmitted into a 10-bit serial data stream. This conversion and data alignment are shown in Figure 3-9. The serial port transmits the least significant bit of the 10-bit data “a” first and proceeds to “j”. This allows data to be read and matched to the form shown in Table 3-17.

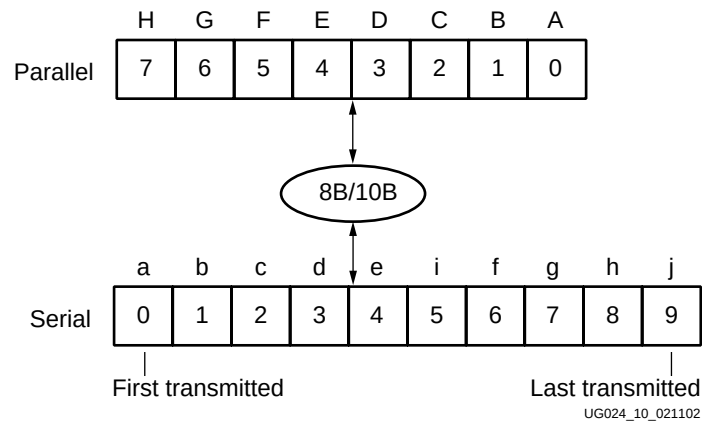


Figure 3-9: 8B/10B Parallel to Serial Conversion

The serial data bit sequence is dependent on the width of the parallel data. The most significant byte is always sent first regardless of the whether 1-byte, 2-byte, or 4-byte paths are used. The least significant byte is always last. **Figure 3-10** shows a case when the serial data corresponds to each byte of the parallel data. TXDATA [31:24] is serialized and sent out first followed by TXDATA [23:16], TXDATA [15:8], and finally TXDATA [7:0]. The 2-byte path transmits TXDATA [15:8] and then TXDATA [7:0].

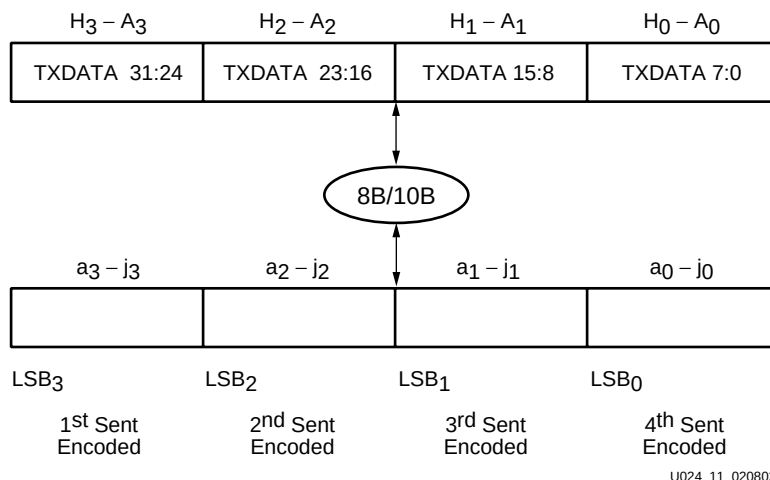


Figure 3-10: 4-Byte Serial Structure

### HDL Code Examples: Transceiver Bypassing of 8B/10B Encoding

8B/10B encoding can be bypassed by the transceiver. The TX8B10BBYPASS is set to 1111; the RXDECODE attribute is set to "FALSE" to create the extra two bits needed for a 10-bit data bus; and TXCHARDISPMODE, TXCHARDISPVAL, RXCHARISK, and RXRUNDISP are added to the 8-bit data bus.

Availability for download of code examples with 8B/10B bypassing is planned for a later date.

## CRC Operation

Cyclic Redundancy Check (CRC) is a procedure to detect errors in the received data. There are four possible CRC modes, USER\_MODE, ETHERNET, INFINIBAND, and FIBRE\_CHAN. These are only modifiable for the GT\_XAUI and GT\_CUSTOM. Each mode has a start-of-packet (SOP) and end-of-packet (EOP) setting to determine where to start and end the CRC monitoring. USER\_MODE allows the user to define the SOP and EOP by setting the CRC\_START\_OF\_PKT and CRC\_END\_OF\_PKT to one of the valid K-characters (**Table 3-18**). The CRC is controlled by RX\_CRC\_USE and TX\_CRC\_USE. Whenever these attributes are set to TRUE, CRC is used. A CRC error can be "forced" with the use of TXFORCECRCERR. This causes TX\_CRC\_FORCE\_VALUE to be XORed with the computed CRC, to test the CRC error logic.

### CRC Generation

Rocket I/O transceivers support a 32-bit invariant CRC (fixed 32-bit polynomial shown below) for Gigabit Ethernet, Fibre Channel, Infiniband, and user-defined modes.

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$$

The CRC recognizes the SOP (Start of Packet), EOP (End of Packet), and other packet features to identify the beginning and end of data. These SOP and EOP are defined by the

mode, except in the case where CRC\_MODE is USER\_DEFINED. The user-defined mode uses CRC\_START\_OF\_PKT and CRC\_END\_OF\_PKT to define SOP and EOP.

The transmitter computes 4-byte CRC on the packet data between the SOP and EOP (excluding the CRC placeholder bytes). The transmitter inserts the computed CRC just before the EOP. The transmitter modifies trailing Idles or EOP if necessary to generate correct running disparity for Gigabit Ethernet and FibreChannel. The receiver recomputes CRC and verifies it against the inserted CRC. Figure 3-11 shows the packet format for CRC generation. The empty boxes are only used in certain protocols (Ethernet). The user logic must create a four-byte placeholder for the CRC; otherwise, data is overwritten.



UG024\_07\_021102

Figure 3-11: CRC Packet Format

## CRC Latency

Enabling CRC increases the transmission latency from TXDATA to TXP and TXN. The enabling of CRC does not affect the latency from RXP and RXN to RXDATA. The typical and maximum latencies, expressed in TXUSRCLK2/RXUSRCLK2 cycles, are shown in Table 3-19. For timing diagrams expressing these relationships, please see Module 3 of the Virtex-II Pro Data Sheet.

Table 3-19: Effects of CRC on Transceiver Latency

|              | TXDATA to TXP and TXN,<br>in TXUSRCLK2 Cycles |         | RXP and RXN to RXDATA,<br>in RXUSRCLK2 Cycles |         |
|--------------|-----------------------------------------------|---------|-----------------------------------------------|---------|
|              | Typical                                       | Maximum | Typical                                       | Maximum |
| CRC Disabled | 8                                             | 11      | 25                                            | 42      |
| CRC Enabled  | 14                                            | 17      | 25                                            | 42      |

## CRC Limitations

There are several limitations to the Rocket I/O CRC. First, CRC is not supported in byte-striped data. If byte-striped (channel bonding) is required, CRC must be computed in CLBs prior to the byte-striping. The CRC support of Infiniband is incomplete, because the 16-bit variant CRC must be done in the CLBs making the transceiver core CRC function redundant. For this case, set TX\_CRC\_USE = FALSE.

## CRC Modes

The Rocket I/O transceiver has four CRC modes: USER\_MODE, FIBRECHANNEL, ETHERNET, and INFINIBAND. These CRC modes are briefly explained below.

### USER\_MODE

USER\_MODE is the simplest CRC methodology. The CRC checks for the SOP and EOF, calculates CRC on the data, and leaves the four remainders directly before the EOP. The CRC form for the user-defined mode is shown in Figure 3-12, along with the timing for asserting RXCHECKINGCRC and RXCRCERR High with respect to the incoming data.

TXCRCFORCEERR and RXCRCERR are both useful during testing. When using CRC, testing the CRC logic can be done by setting CRCFORCEERR High to incorporate an error into the data that is transmitted. When the data is received in a testing mode, such as serial loopback, the data is "corrupted" and the receiver should signal an error with the use of

RXCRCERR when the RXCHECKINGCRC is asserted High. User logic determines the procedure when a RXCRCERR occurs.

## FIBRECHANNEL

The Fibre Channel CRC is similar to the USER\_MODE CRC (Figure 3-12) with one exception: In FibreChannel, SOP and EOP are the protocol delimiters, while USER\_MODE uses the two attributes CRC\_START\_OF\_PKT and CRC\_END\_OF\_PKT to define SOP and EOP. Both USER\_MODE and Fibre Channel, however, disregard the SOP and EOP in CRC computation.

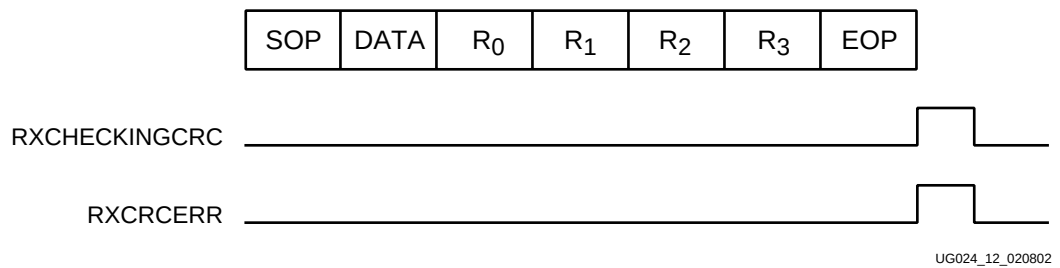


Figure 3-12: USER\_MODE / FIBRE\_CHANNEL Mode

## ETHERNET

The Ethernet CRC is more complex (Figure 3-13). The SOP, EOP, and Preamble are neglected by the CRC. The extension bytes are special "K" characters in special cases. The extension bytes are untouched by the CRC as are the Trail bits, which are added to maintain packet length.

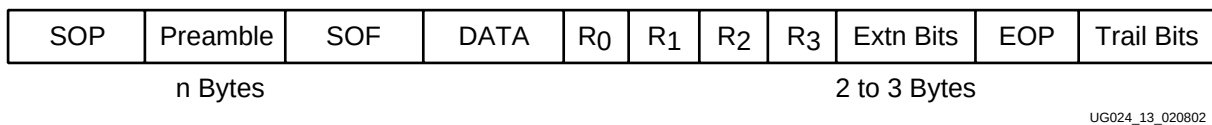


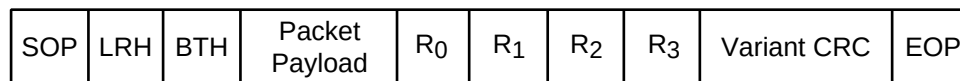
Figure 3-13: Ethernet Mode

## INFINIBAND

The Infiniband CRC is the most complex mode, and is not supported in the CRC generator. Infiniband CRC contains two computation types: an invariant 32-bit CRC, the same as in Ethernet protocol; and a variant 16-bit CRC, which is not supported in the hard core. Infiniband CRC must be implemented entirely in the FPGA fabric.

There are also two Infiniband Architecture (IBA) packets, a local and a global. Both of these IBA packets are shown in Figure 3-14.

### Local IBA



### Global IBA

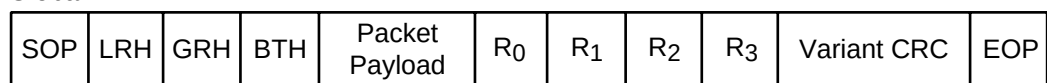


Figure 3-14: Infiniband Mode

The CRC is calculated with certain bits masked in LRH and GRH, depending on whether the packet is local or global. The size of these headers is shown in [Table 3-20](#).

**Table 3-20: Global and Local Headers**

| Packet | Description           | Size     |
|--------|-----------------------|----------|
| LRH    | Local Routing Header  | 8 Bytes  |
| GRH    | Global Routing Header | 40 Bytes |
| BTH    | IBA Transport Header  | 12 Bytes |

The CRC checks the LNH (Link Next Header) of the LRH. LRH is shown in [Figure 3-15](#), along with the bits the CRC uses to evaluate the next packet.

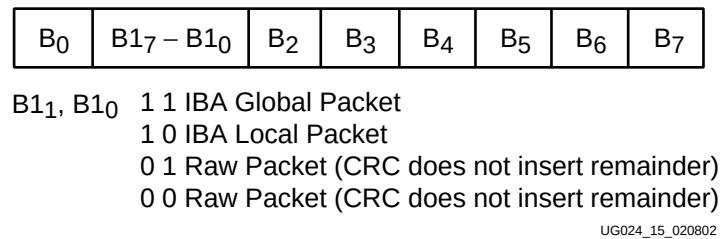


**Figure 3-15: Local Route Header**

Because of the complexity of the CRC algorithms and implementations, especially with Infiniband, a more in-depth discussion is beyond the scope of this manual.

## Channel Bonding (Channel-to-Channel Alignment)

Channel bonding is the technique of tying several serial channels together to create one aggregate channel. Several channels are fed on the transmit side by one parallel bus and reproduced on the receive side as the identical parallel bus. The maximum number of serial differential pairs that can be bonded is 16. For implementation guidelines, see [Implementation Tools](#), [page 1131](#).

Channel bonding allows those primitives that support it to send data over multiple "channels." Among these primitives are GT\_CUSTOM, GT\_INFINIBAND, GT\_XAUI, and GT\_AURORA. To "bond" channels together, there is always one "master." The other channels can either be a SLAVE\_1\_HOP or SLAVE\_2\_HOPS. SLAVE\_1\_HOP is a slave to a master that can also be daisy chained to a SLAVE\_2\_HOPS. A SLAVE\_2\_HOPS can only be a slave to a SLAVE\_1\_HOP and its CHBONDO does not connect to another transceiver. To designate a transceiver as a master or a slave, the attribute CHAN\_BOND\_MODE must be set to one of three designations: Master, SLAVE\_1\_HOP, or SLAVE\_2\_HOPS. To shut off channel bonding, set the transceiver attribute to "off." The possible values that can be used are shown in [Table 3-21](#).

**Table 3-21: Bonded Channel Connections**

| Mode         | CHBONDI         | CHBONDO         |
|--------------|-----------------|-----------------|
| OFF          | NA              | NA              |
| MASTER       | NA              | slave 1 CHBONDI |
| SLAVE_1_HOP  | master CHBONDO  | slave 2 CHBONDI |
| SLAVE_2_HOPS | slave 1 CHBONDO | NA              |

The channel bonding sequence is similar to the clock correction sequence. This sequence is set to the appropriate sequence for the primitives supporting channel bonding. The GT\_CUSTOM is the only primitive allowing modification to the sequence. These sequences are comprised of one or two sequences of length up to 4 bytes each, as set by CHAN\_BOND\_SEQ\_LEN and CHAN\_BOND\_SEQ\_2\_USE. Other control signals include the attributes:

- CHAN\_BOND\_WAIT
- CHAN\_BOND\_OFFSET
- CHAN\_BOND\_LIMIT
- CHAN\_BOND\_ONE\_SHOT

Typical values for these attributes are:

CHAN\_BOND\_WAIT = 8

CHAN\_BOND\_OFFSET = CHAN\_BOND\_WAIT

CHAN\_BOND\_LIMIT = 2 x CHAN\_BOND\_WAIT

Lower values are not recommended. Use higher values only if channel bonding sequences are farther apart than 17 bytes.

**Table 3-22** shows different settings for CHAN\_BONDONE\_SHOT and ENCHANSYNC in Master and Slave applications.

**Table 3-22: Master/Slave Channel Bonding Attribute Settings**

|                    | Master                     | Slave    |
|--------------------|----------------------------|----------|
| CHAN_BOND_ONE_SHOT | TRUE or FALSE as desired   | FALSE    |
| ENCHANSYNC         | Dynamic control as desired | Tie High |

## HDL Code Examples: Channel Bonding

Availability for download of code examples implementing channel bonding is planned for a later date.



## Analog Design Considerations

### Serial I/O Description

The Rocket I/O transceiver transmits and receives serial differential signals. This feature operates at a nominal supply voltage of 2.5 VDC. A serial differential pair consists of a true ( $V_P$ ) and a complement ( $V_N$ ) set of signals. The voltage difference represents the transferred data. Thus:  $V_P - V_N = V_{DATA}$ . Differential switching is performed at the crossing of the two complementary signals. Therefore, no separate reference level is needed.

A graphical representation of this concept is shown in [Figure 4-1](#).

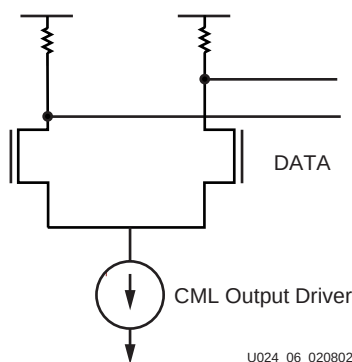


**Figure 4-1: Differential Amplifier**

The Rocket I/O transceiver is implemented in Current Mode Logic (CML). A CML output consists of transistors configured as shown in [Figure 4-1](#). CML uses a positive supply and offers easy interface requirements. In this configuration, both legs of the driver,  $V_P$  and  $V_N$ , sink current, with one leg always sinking more current than its complement. The CML output consists of a differential pair with 50 $\Omega$  (or, optionally, 75 $\Omega$ ) source resistors. The signal swing is created by switching the current in a common-drain differential pair.

The differential transmitter specification is shown in [Table 4-1](#), page 1120.



Table 4-1: Differential Transmitter Parameters

| Parameter   |                                                   | Min | Typ | Max  | Units | Conditions                                  |
|-------------|---------------------------------------------------|-----|-----|------|-------|---------------------------------------------|
| $V_{OUT}$   | Serial output differential peak to peak (TXP/TXN) | 800 |     | 1600 | mV    | Output differential voltage is programmable |
| $V_{TTX}$   | Output termination voltage supply                 | 1.8 |     | 2.8  | V     |                                             |
| $V_{TCM}$   | Common mode output voltage range                  | 1.5 |     | 2.5  | V     |                                             |
| $V_{ISKEW}$ | Differential output skew                          |     |     | 15   | ps    |                                             |

## Pre-emphasis Techniques

In pre-emphasis, the initial differential voltage swing is boosted to create a stronger rising or falling waveform. This method compensates for high frequency loss in the transmission media that would otherwise limit the magnitude of this waveform. The effects of pre-emphasis are shown in four scope screen captures, Figure 4-2 through Figure 4-5 on the pages following. The STRONG notation in Figure 4-3 is used to show that the waveform is greater in voltage magnitude, at this point, than the LOGIC or normal level (i.e., no pre-emphasis).

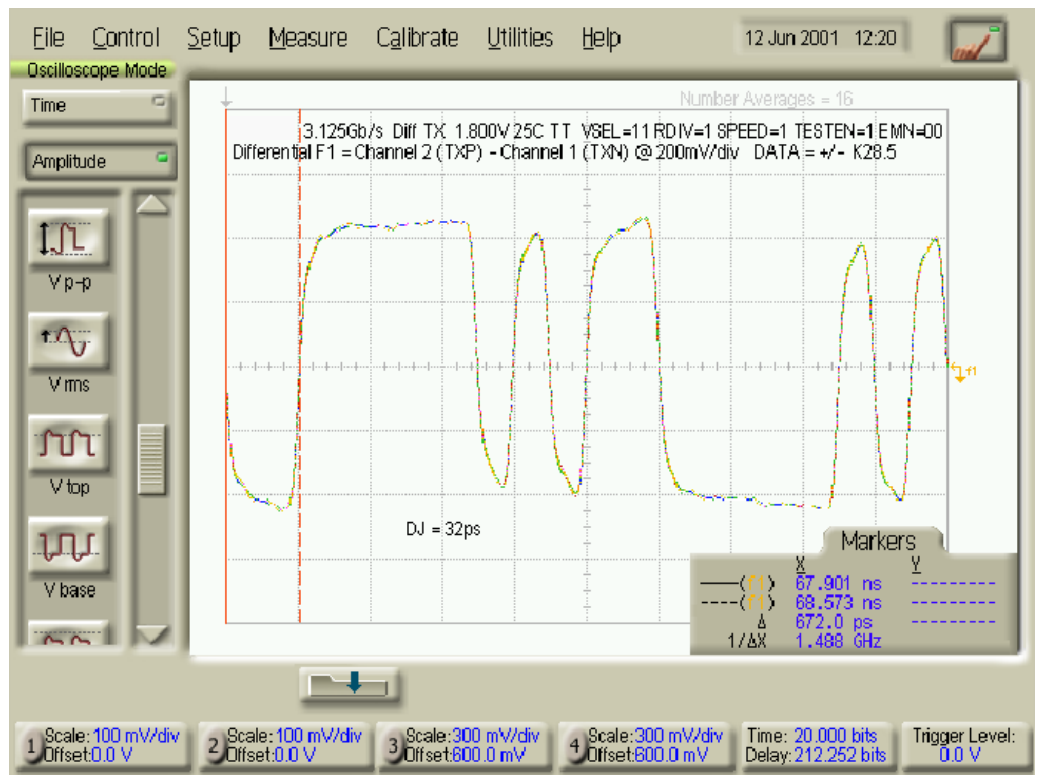
A second characteristic of Rocket I/O transceiver pre-emphasis is that the STRONG level is reduced after some time to the LOGIC level, thereby minimizing the voltage swing necessary to switch the differential pair into the opposite state.

Lossy transmission lines cause the dissipation of electrical energy. This pre-emphasis technique extends the distance that signals can be driven down lossy line media and increases the signal-to-noise ratio at the receiver.

The four levels of pre-emphasis are shown in Table 4-2.

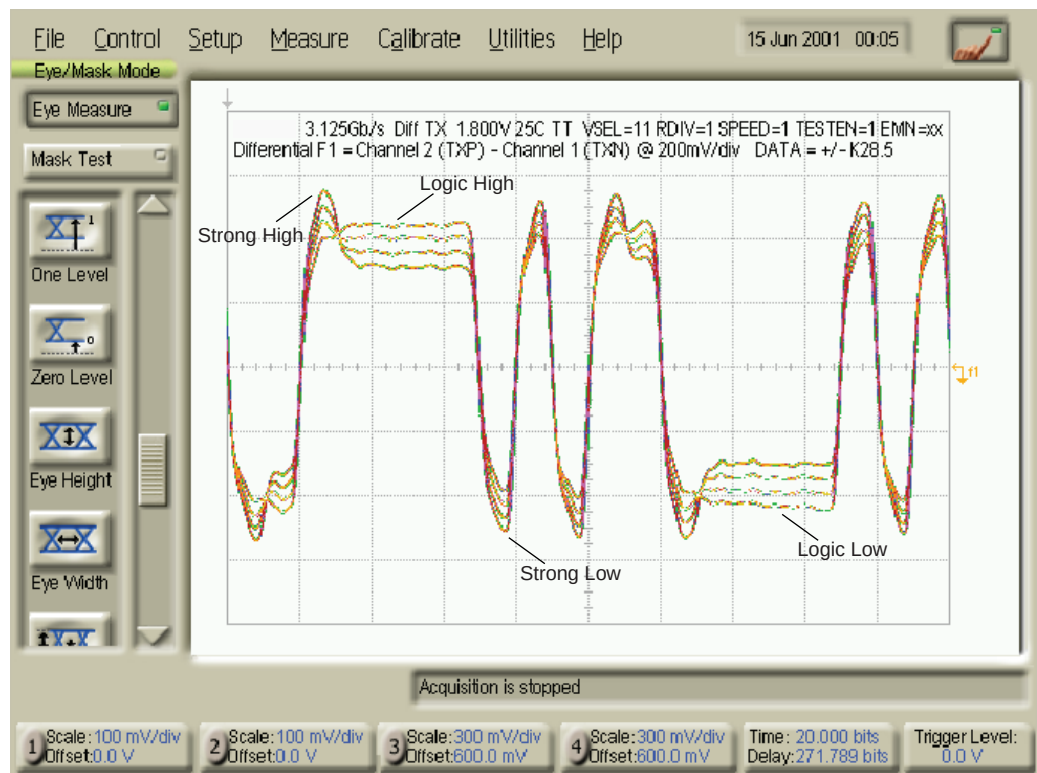
Table 4-2: Pre-emphasis Values

| Attribute Values | Emphasis (%) |
|------------------|--------------|
| 0                | 10           |
| 1                | 20           |
| 2                | 25           |
| 3                | 33           |



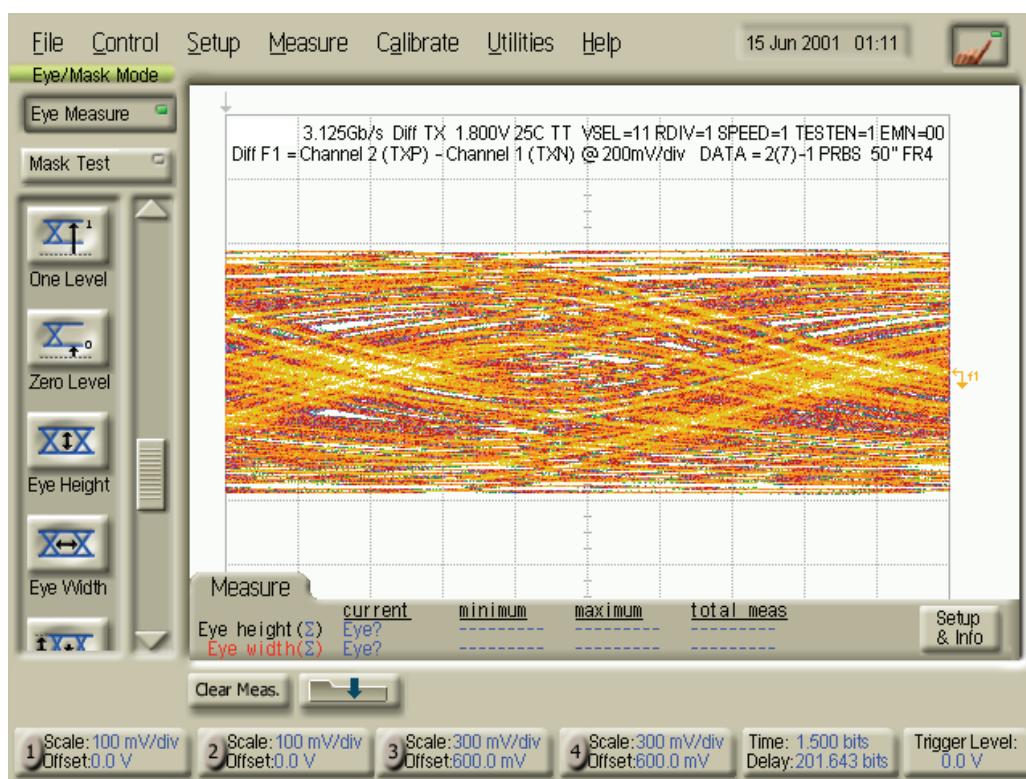
UG024\_17\_020802

Figure 4-2: Alternating K28.5+ with No Pre-Emphasis



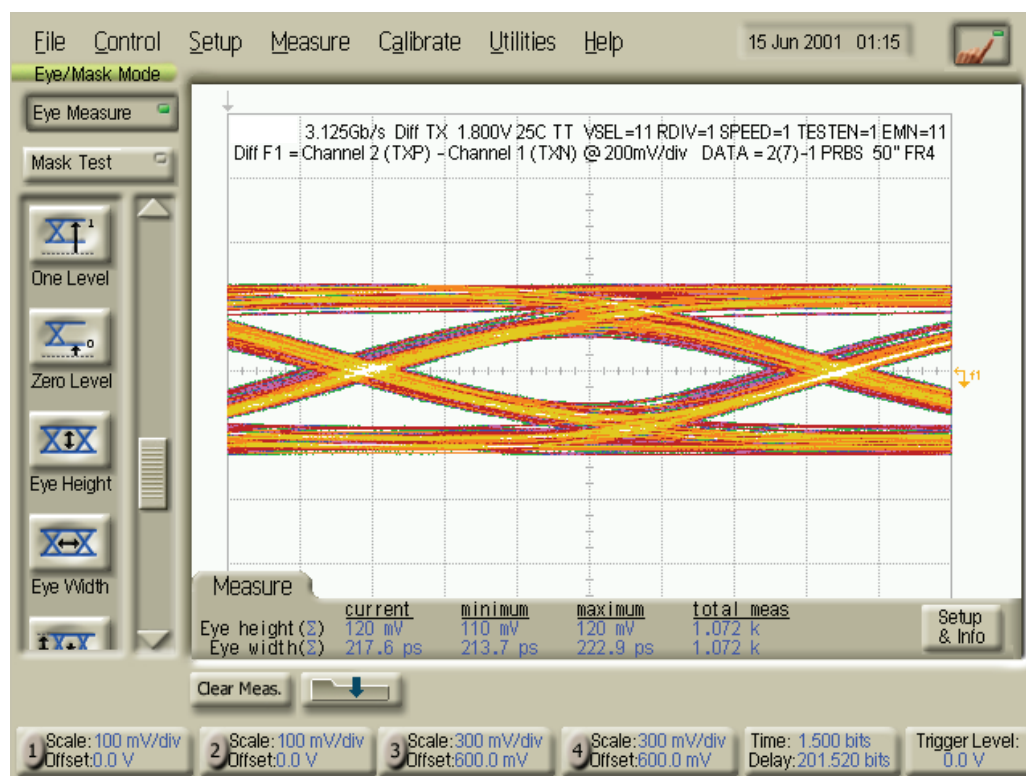
UG024\_18\_020802

Figure 4-3: K28.5+ with Pre-Emphasis



UG024\_19\_020802

Figure 4-4: Eye Diagram: without Pre-Emphasis



UG024\_19a\_020802

Figure 4-5: Eye Diagram: with 30% Pre-Emphasis

## Differential Receiver

The differential receiver accepts the  $V_P$  and  $V_N$  signals, carrying out the difference calculation  $V_P - V_N$  electronically.

All input data must be differential and nominally biased to a common mode voltage of 0.5 V – 2.5 V, or AC coupled. Internal terminations provide for simple 50Ω or 75Ω transmission line connection.

The differential receiver parameters are shown in [Table 4-3](#).

**Table 4-3: Differential Receiver Parameters**

| Parameter   |                                                            | Min | Typ | Max   | Units             | Conditions |
|-------------|------------------------------------------------------------|-----|-----|-------|-------------------|------------|
| $V_{IN}$    | Serial input differential peak to peak (RXP/RXN)           | 175 |     | 1,000 | mV                |            |
| $V_{ICM}$   | Common mode input voltage range                            | 500 |     | 2500  | mV                |            |
| $T_{ISKEW}$ | Differential input skew                                    |     |     | 75    | ps                |            |
| $T_{JTOL}$  | Receive data total jitter tolerance (peak to peak)         |     |     | 0.65  | UI <sup>(1)</sup> |            |
| $T_{DJTOL}$ | Receive data deterministic jitter tolerance (peak to peak) |     |     | 0.41  | UI                |            |

**Notes:**

1. UI = Unit Interval

## Jitter

*Jitter* is defined as the short-term variations of significant instants of a signal from their ideal positions in time (ITU). Jitter is typically expressed in a decimal fraction of Unit Interval (UI), e.g. 0.3 UI.

### Total Jitter (DJ + RJ)

#### Deterministic Jitter (DJ)

DJ is data pattern dependant jitter, attributed to a unique source (e.g., Inter Symbol Interference (ISI) due to loss effects of the media). DJ is linearly additive.

#### Random Jitter (RJ)

RJ is due to stochastic sources, such as substrate, power supply, etc. RJ is additive as the sum of squares, and follows a bell curve.

## Clock and Data Recovery

The serial transceiver input is locked to the input data stream through Clock and Data Recovery (CDR), a built-in feature of the Rocket I/O transceiver. CDR keys off the rising and falling edges of incoming data and derives a clock that is representative of the incoming data rate.

The derived clock, RXRECCLK, is presented to the FPGA fabric at 1/20th the incoming data rate. This clock is generated and locked to as long as it remains within the specified component range. This range is shown in [Table 4-4](#).

Table 4-4: CDR Parameters

| Parameter                            |                                                                    | Min | Typ | Max   | Units  | Conditions                         |
|--------------------------------------|--------------------------------------------------------------------|-----|-----|-------|--------|------------------------------------|
| Frequency Range                      | Serial input differential (RXP/RXN)                                | 175 |     | 1,000 | MHz    | Peak-to-peak                       |
| Frequency Offset                     |                                                                    |     |     |       | ppm    |                                    |
| T <sub>DCREF</sub>                   | REFCLK duty cycle                                                  | 45  | 50  | 55    | %      |                                    |
| T <sub>RCLK</sub> /T <sub>FCLK</sub> | REFCLK rise and fall time (see Virtex-II Pro Data Sheet, Module 3) |     |     | 75    | ps     | Between 20% and 80% voltage levels |
| T <sub>GJT</sub>                     | REFCLK total jitter                                                |     |     | 40    | ps     | Peak-to-peak                       |
| T <sub>LOCK</sub>                    | Clock recovery frequency acquisition time                          |     | 10  |       | μs     |                                    |
| T <sub>UNLOCK</sub>                  |                                                                    |     |     |       | cycles |                                    |

A sufficient number of transitions must be present in the data stream for CDR to work properly. The CDR circuit is guaranteed to work with 8B/10B encoding. Further, CDR requires approximately 5,000 transitions upon power-up to guarantee locking to the incoming data rate. Once lock is achieved, up to 75 missing transitions can be tolerated before lock to the incoming data stream is lost.

Care must be taken if a custom serial data stream is engineered so that the transition frequency rate requirement of 8B/10B encoding is met. An additional feature of CDR is its ability to accept an external precision clock, REFCLK, as an optional input. REFCLK acts either to clock incoming data or to assist in synchronizing the derived RXRECCLK.

For further clarity, the TXUSRCLK is used to clock data from the FPGA core to the TX FIFO. The FIFO depth accounts for the slight phase difference between these two clocks. If the clocks are locked in frequency, then the FIFO acts much like a pass-through buffer.

# PCB Design Requirements

In order to ensure reliable operation of the Rocket I/O transceivers, certain requirements must be met by the designer. This section outlines these requirements governing power filtering networks, high-speed differential signal traces, and reference clocks. Any designs that do not adhere to these requirements will not be supported by Xilinx, Inc.

## Power Filtering

Each Rocket I/O transceiver has five power supply pins, all of which are sensitive to noise. **Table 4-5** summarizes the power supply pins, their names, and associated voltages.

**Table 4-5: Transceiver Power Supplies**

| Supply    | 2.5V | 1.5V - 2.8V | Description                                            |
|-----------|------|-------------|--------------------------------------------------------|
| AVCCAUXRX | X    |             | Analog RX supply                                       |
| AVCCAUTX  | X    |             | Analog TX supply                                       |
| VTRX      |      | X           | RX termination supply                                  |
| VTTX      |      | X           | TX termination supply                                  |
| GNDA      |      |             | Analog ground for transmit and receive analog supplies |

To operate properly, the Rocket I/O transceiver requires a certain level of noise isolation from surrounding noise sources. For this reason, it is required that dedicated voltage regulators be used to power the Rocket I/O circuitry. These power supply circuits must not be shared with any other supplies (including FPGA supplies  $V_{CCINT}$ ,  $V_{CCO}$ ,  $V_{CCAUX}$ , and  $V_{REF}$ ). Voltage regulators may be shared among transceiver power supplies of the same voltage.

The required voltage regulator is the Linear Technology LT1963-2.5 device. This regulator must be used in the circuit specified by the manufacturer. **Figure 4-6** shows the schematic with values for a 2.5V supply, as would be used for AVCCAUXRX and AVCCAUTX.

Refer to the manufacturer's Web page at <http://www.linear-tech.com> for further information about this device.

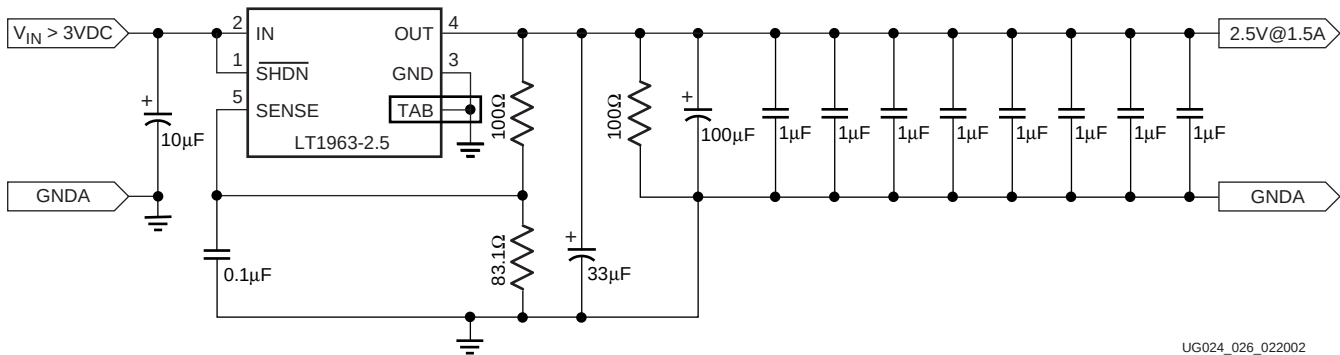


**Figure 4-6: Power Supply Circuit Using LT1963 Regulator**

To achieve the necessary isolation from power supply noise, filter networks are required on the power supply pins. The topology of these capacitor and ferrite bead circuits is given in **Figure 4-7**.

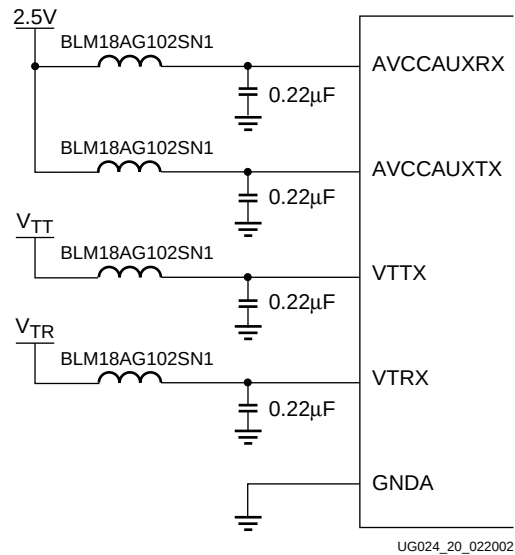


Figure 4-7: Power Filtering Network for One Transceiver

Each transceiver power pin requires one capacitor and one ferrite bead. The capacitors must be of value 0.22 µF in an 0603 SMT package of X7R dielectric material at 10% tolerance, rated to at least 5V. These capacitors must be placed within 1 cm of the pins they are bypassing. The ferrite bead is the Murata BLM18AG102SN1.

Figure 4-8 and Figure 4-9 show an example layout of the power filtering network for four transceivers. The device is in an ff672 package, which has eight transceivers total—four on the top edge and four on the bottom edge. Figure 4-8 shows the top PCB layer, with lands for the capacitors and ferrite beads of the VTTX and VTRX supplies. The ferrite beads are L1, L2, L3, L4, L9, L11, L12, and L21; the capacitors are C85, C90, C94, C96, C98, C100, C119, and C124. Figure 4-9 shows the bottom PCB layer, with lands for the capacitors and ferrite beads of the AVCCAUTX and AVCCAUXRX supplies. The ferrite beads are L10, L13, L15, L16, L19, L33, and L34; the capacitors are C141, C144, C211, C221, C223, C225, C227, and C229.

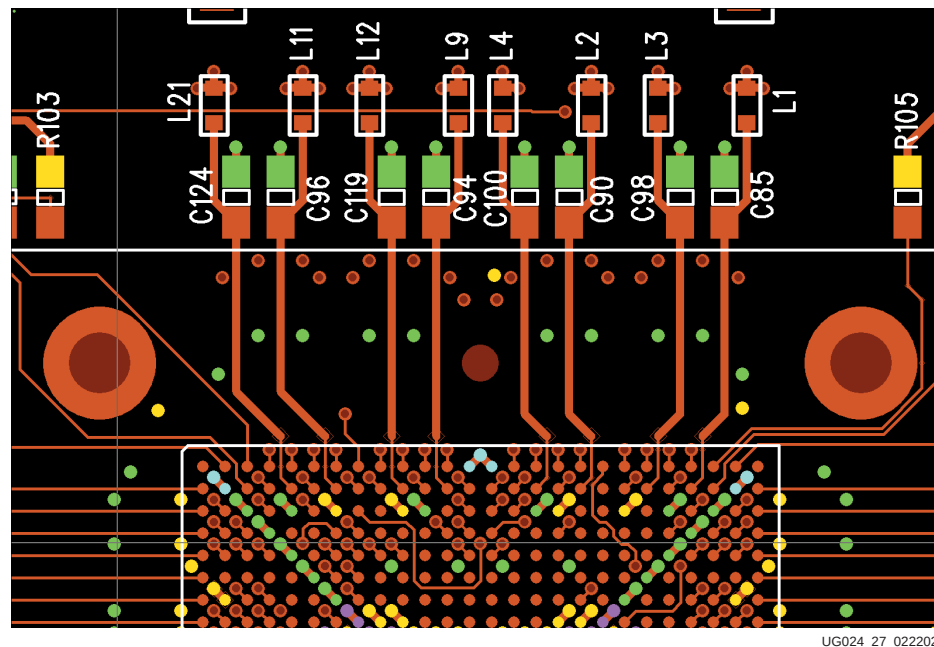


Figure 4-8: Example Power Filtering PCB Layout for Four MGTs, Top Layer



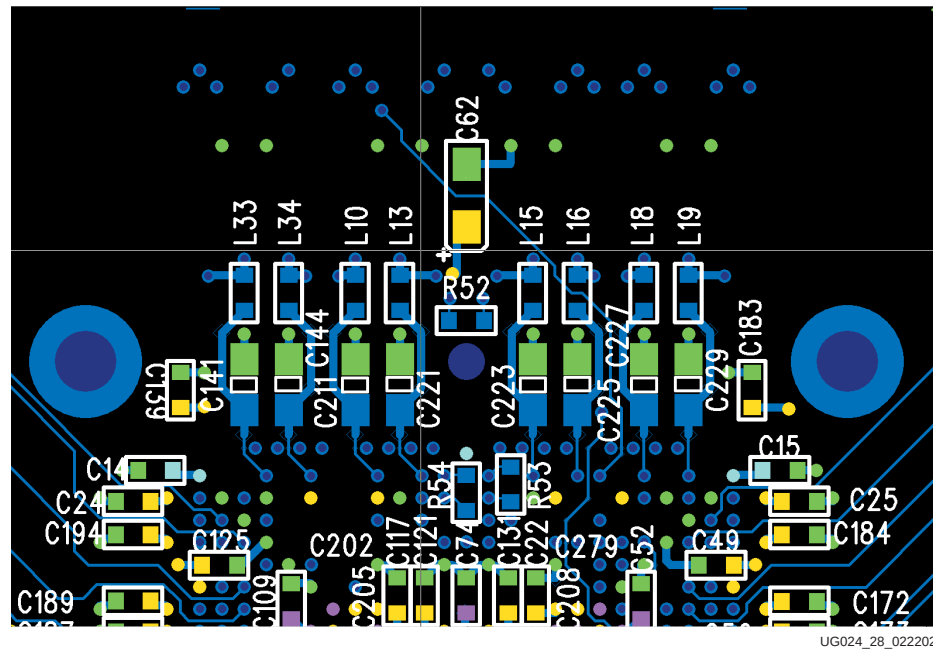


**Figure 4-9: Example Power Filtering PCB Layout for Four MGTs, Bottom Layer**

Differential impedance of traces on the finished PCB should be verified with Time Domain Reflectometry (TDR) measurements.

Power and ground paths in the PCB must be designed to have the lowest inductance possible. To this end, dedicated planes for ground must be used. When vias are a part of the power distribution path (as they are when connecting a bypass capacitor to its associated power and ground layers), multiple vias should be used to decrease the total inductance of the path. If power filtering capacitors are mounted on the same side of the board as the FPGA, designers can choose to place power layers closer to the surface of the board to shorten the path of power travel through vias.

## High-Speed Serial Trace Design

### Routing Serial Traces

All Rocket I/O transceiver I/Os are placed on the periphery of the BGA package to facilitate routing and inspection (since JTAG is not available on serial I/O pins). Two output/input impedance options are available in the Rocket I/O transceivers: 50Ω and 75Ω. Controlled impedance traces with a corresponding impedance should be used to connect the Rocket I/O transceiver to other compatible transceivers. In chip-to-chip PCB applications, 50Ω termination and 100Ω differential transmission lines are recommended.

When routing a differential pair, the complementary traces must be matched in length to as close a tolerance as is feasible. Length mismatches produce common mode noise and radiation. Severe length mismatches produce jitter and unpredictable timing problems at the receiver. Matching the differential traces to within 50 mils (1.27 mm) produces a robust design. Since signals propagate in FR4 PCB traces at approximately 180 ps per inch, a difference of 50 mils produces a timing skew of roughly 9 ps. Use SI CAD tools to confirm these assumptions on specific board designs.

All signal traces must have an intact reference plane beneath them. Stripline and microstrip geometries may be used. The reference plane should extend no less than five trace widths to either side of the trace to ensure predictable transmission line behavior.

Routing of a differential pair is optimally done in a point-to-point fashion, ideally remaining on the same PCB routing layer. As vias represent an impedance discontinuity, layer-to-layer changes should be avoided wherever possible. It is acceptable to traverse the



PCB stackup to reach the transmitter and receiver package pins. If serial traces must change layers, care must be taken to ensure an intact current return path. For this reason, routing of high-speed serial traces should be on signal layers that share a reference plane. If the signal layers do not share a reference plane, a capacitor of value 0.01  $\mu\text{F}$  should be connected across the two reference layers close to the vias where the signals change layers. If both of the reference layers are DC coupled (if they are both ground), they can be connected with vias close to where the signals change layers.

To control crosstalk, serial differential traces should be spaced at least five trace separation widths from all other PCB routes, including other serial pairs. A larger spacing is required if the other PCB routes carry especially noisy signals, such as TTL and similar.

The Rocket I/O transceiver is designed to function at 3.125 Gb/s through 20 inches of FR4 with two high-bandwidth connectors. Longer trace lengths require either a low-loss dielectric or considerably wider serial traces.

## Differential Trace Design

The characteristic impedance of a pair of differential traces depends not only on the individual trace dimensions, but also on the spacing between them. The Rocket I/O transceivers require either a 100 $\Omega$  or 150 $\Omega$  differential trace impedance (depending on whether the 50 $\Omega$  or 75 $\Omega$  termination option is selected). To achieve this differential impedance requirement, the characteristic impedance of each individual trace must be slightly higher than half of the target differential impedance. A field solver should be used to determine the exact trace geometry suited to the specific application (Figure 4-10). This task should not be left up to the PCB vendor.

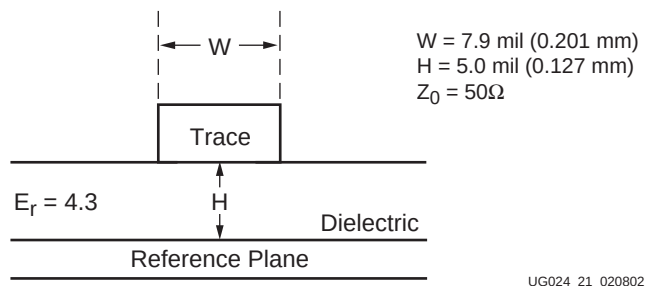


Figure 4-10: Single-Ended Trace Geometry

Tight coupling of differential traces is recommended. Tightly coupled traces (as opposed to loosely coupled) maintain a very close proximity to one another along their full length. Since the differential impedance of tightly coupled traces depends heavily on their proximity to each other, it is imperative that they maintain constant spacing along their full length, without deviation. If it is necessary to separate the traces in order to route through a pin field or other PCB obstacle, it can be helpful to modify the trace geometry in the vicinity of the obstacle to correct for the impedance discontinuity (increase the individual trace width where trace separation occurs).

Figure 4-11 and Figure 4-12 show examples of PCB geometries that result in 100 $\Omega$  differential impedance.

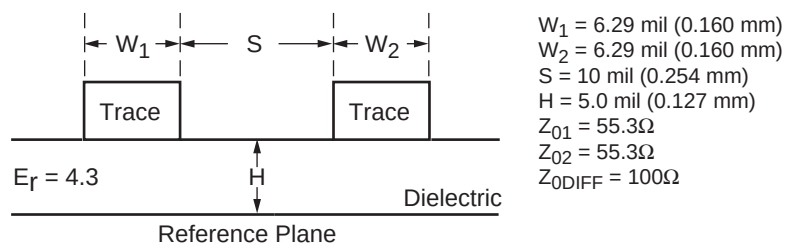


Figure 4-11: Microstrip Edge-Coupled Differential Pair

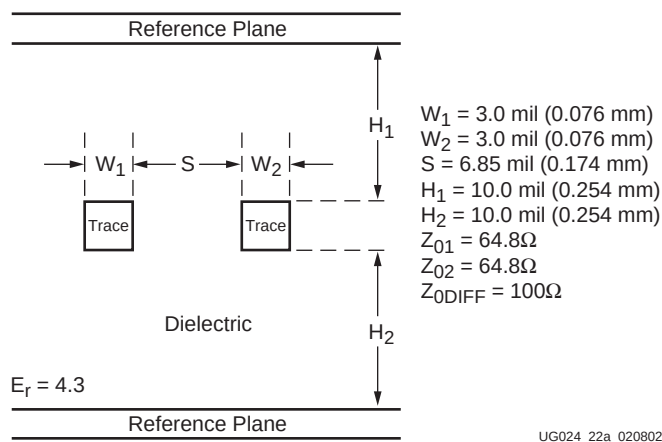


Figure 4-12: Stripline Edge-Coupled Differential Pair

## AC and DC Coupling

AC coupling (use of DC blocking capacitors in the signal path) should be used in cases where transceiver differential voltages are compatible, but common mode voltages are not. Some designs require AC coupling to accommodate hot plug-in, and/or differing power supply voltages at different transceivers. This is illustrated in Figure 4-13.

DC coupling (direct connection) is preferable in cases where Rocket I/O transceivers are interfaced with other Rocket I/O transceivers or other Mindspeed transceivers that have compatible differential and common mode voltage specifications. Passive components are not required when DC coupling is used. This is illustrated in Figure 4-14.

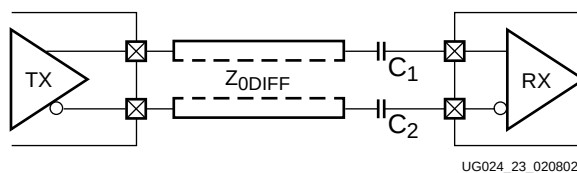


Figure 4-13: AC-Coupled Serial Link

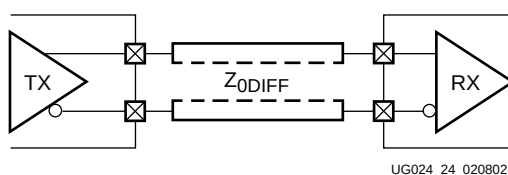


Figure 4-14: DC-Coupled Serial Link

## Power Consumption

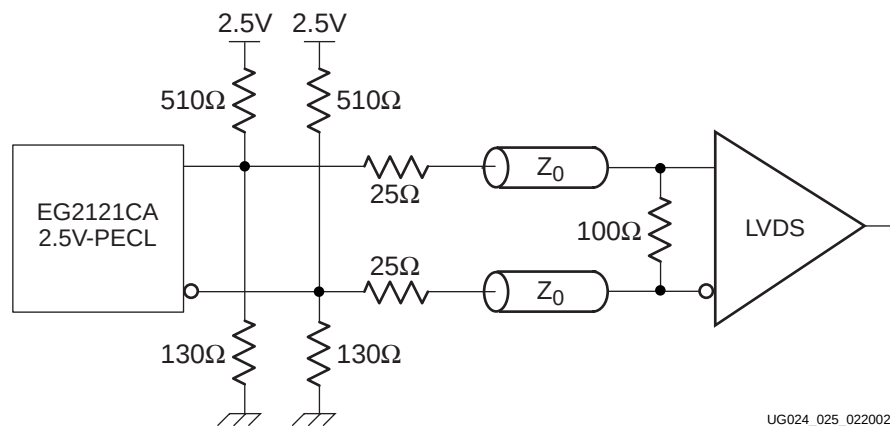
The maximum power consumption per port is 350 mW at 3.125 Gb/s operation. All unused serial I/O can be turned off to consume zero power.

### POWERDOWN

POWERDOWN allows "shutting off" the transceiver in case it is not needed for the design, or will not be transmitting or receiving for a long period of time. When POWERDOWN is enabled, the transceiver does not use any power. The clocks are disabled and do not propagate through the core. The 3-state TXP and TXN pins are set to high-Z, while the outputs to the fabric are frozen but *not* set to high-Z. Unused transceivers are automatically powered down to save on power consumption.

## Reference Clock

A high degree of accuracy is required from the reference clock. For this reason, it is required that an EPSON EG-2121CA 2.5V oscillator be used. The power supply circuit specified by the manufacturer must be used, and the circuit in Figure 4-15 must be used to interface the LVPECL outputs of the oscillator with the LVDS inputs of the transceiver reference clock.



UG024\_025\_022002

Figure 4-15: Reference CLock Oscillator Interface

# *Simulation and Implementation*

---

## Simulation Models

### Smart Model

Smart models are an encrypted version to the actual HDL code. These models allow the user to simulate with the actual functionality without having access to the code itself. A simulator with smart model capability is required to use the smart models.

### HSPICE

HSPICE is an analog design model that allows simulation of the RX and TX high-speed transceiver. The following HSPICE deck is an example of how to set up such a simulation.

### Behavioral

Behavioral models allow for simulation without the need to upgrade the simulator to support smart models.

## Implementation Tools

### Synthesis

During synthesis, the transceiver is treated as a "black box." This requires that a wrapper be used that describes the modules port.

### Par

For place and route, the transceiver has one restriction. This is required when channel bonding is implemented. Because of the delay limitations on the CHBONDO to CHBONDI ports, linking of the Master to a Slave\_1\_hop must run either in the X or Y direction, but not both.

In [Figure 5-1](#), the two Slave\_1\_hops are linked to the master in only one direction. To navigate to the other slave (a Slave\_2\_hops), both X and Y displacement is needed. This slave needs one level of daisy-chaining, which is the basis of the Slave\_2\_hops setting.

[Figure 5-2](#) shows the channel bonding mode and linking for a 2VP50, which contains more transceivers (16) per chip. To ensure the timing is met on the link between the CHBONDO

and CHBONDI ports, a constraint must be added to check the time delay. The UCF example below shows and describes this.

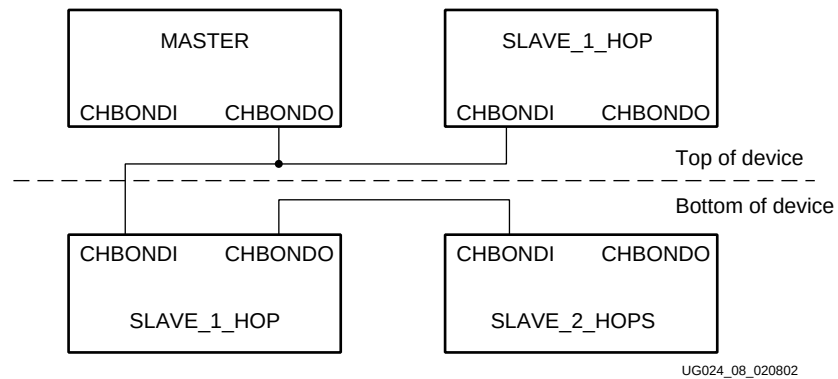


Figure 5-1: 2VP2 Implementation

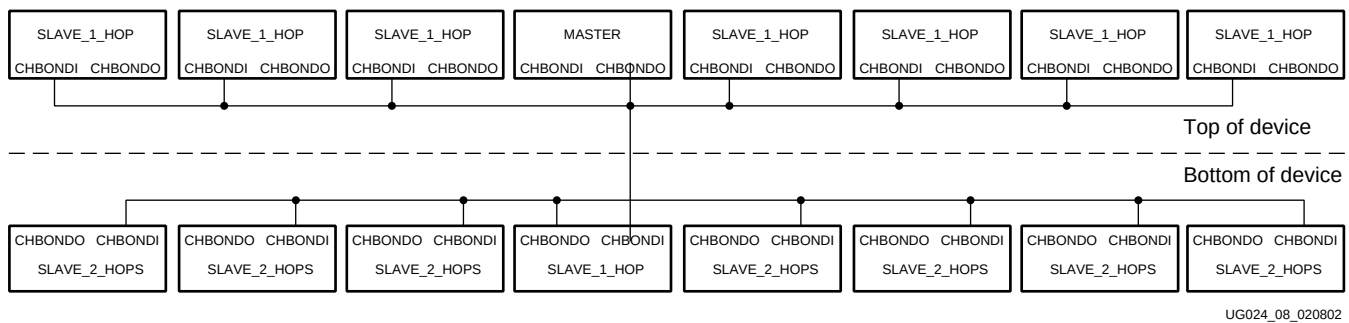


Figure 5-2: 2VP50 Implementation

## UCF Example

```
NET "chbond_*" MAXDELAY = 4.2 ns ;
```

4.2 ns is estimated as the channel bonding delay. This is based upon an RXUSRCLK of 156.25 MHz (6.4 ns period), less 0.2 ns for estimated clock skew, less 2.0 ns for estimated clock-to-out/setup-time adjustment:

$$6.4 \text{ ns} - 0.2 \text{ ns} - 2.0 \text{ ns} = 4.2 \text{ ns}$$

This design used four Rocket I/O multi-gigabit transceivers, consisting of one master, two Slave\_1\_hop, and one Slave\_2\_hops. The net `chbond_m_s01[3:0]` connects the master and two Slave\_1\_hop. The net `chbond_s1_s2[3:0]` connects one Slave\_1\_hop and one Slave\_2\_hops. NET "chbond\_\*" MAXDELAY = 4.2 ns ; constrains all these connections.

## Implementing Clock Schemes

Sometimes certain FPGA resources are needed for specific logic. With Rocket I/O clocking schemes, the user has several resource choices. If the transceivers implemented are only at the top or bottom of the device, the REFCLK of the transceivers is not required to run through a clock tree resource. This saves this resource for other user logic. However, it does require additional I/O pins to be used (one for the DCM and one for the transceiver).

Figure 3-3, page 1090, shows this scenario, which is similar to Figure 3-1 minus the clock-tree resource. If transceivers from both the top and bottom of the device are used or device I/Os are at a premium, the clock tree resource is used allowing one less I/O pin used.

## Diagnostic Signals

Often a diagnostic check is needed upon power-up. Rocket I/O transceivers have several inputs and outputs to run these checks.

### LOOPBACK

LOOPBACK allows the user to send the data that is being transmitted directly to the receiver of the transceiver. [Table 5-1](#) shows the three modes for loopback.

*Table 5-1: LOOPBACK Modes*

| Input Value | Mode                   | Description                                                                                                                                                                                                                                                                     |
|-------------|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 00          | Normal Mode            | The normal mode is selected during normal operation. The transmitted data is sent out the differential transmit ports (TXN, TXP) and are sent to another transceiver without being sent to its own receiver logic. During normal operation, the LOOPBACK should be set to "00". |
| 01          | External Serial Mode   | The external serial mode is used to check that the entire transceiver is working properly. This includes testing of the 8B/10B encoding/decoding. This emulates what another transceiver would receive as data from this specific transceiver design.                           |
| 10          | Internal Parallel Mode | For testing of interfacing logic, the Internal Parallel Mode allows the use of linking the transmit and receive interface logic without having to go to another transceiver in the cases of 8B/10B bypassed or to reduce data latency from TXDATA to RXDATA.                    |



# *Rocket I/O™ Cell Models*

---

## Summary

This appendix documents the Rocket I/O™ Multi-Gigabit Transceiver cell models. The following information lists the Verilog module declarations of the model and pins associated with each of the Rocket I/O communication standards available in the Virtex-II Pro family.

## Verilog Module Declarations

### GT\_AURORA\_1

```
module GT_AURORA_1 (
 CHBONDDONE,
 CHBONDO,
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,
 TXP,
 TXRUNDISP,
 CHBONDI,
 CONFIGENABLE,
 CONFIGIN,
 ENCHANSYNC,
 LOOPBACK,
 POWERDOWN,
 REFCLK,
 REFCLK2,
 REFCLKSEL,
 RXN,
```



```

RXP,
RXPOLARITY,
RXRESET,
RXUSRCLK,
RXUSRCLK2,
TXBYPASS8B10B,
TXCHARDISPMODE,
TXCHARDISPVAL,
TXCHARISK,
TXDATA,
TXFORCECRCERR,
TXINHIBIT,
TXPOLARITY,
TXRESET,
TXUSRCLK,
TXUSRCLK2
);

```

## GT\_AURORA\_2

```

module GT_AURORA_2 (
 CHBONDDONE,
 CHBONDO,
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,
 TXP,
 TXRUNDISP,
 CHBONDI,
 CONFIGENABLE,
 CONFIGIN,
 ENCHANSYNC,
 LOOPBACK,
 POWERDOWN,
 REFCLK,
 REFCLK2,
 REFCLKSEL,
 RXN,
 RXP,
 RXPOLARITY,
 RXRESET,
 RXUSRCLK,
 RXUSRCLK2,
 TXBYPASS8B10B,
 TXCHARDISPMODE,

```

```
TXCHARDISPVAL,
TXCHARISK,
TXDATA,
TXFORCECRCERR,
TXINHIBIT,
TXPOLARITY,
TXRESET,
TXUSRCLK,
TXUSRCLK2
);
```

## GT\_AURORA\_4

```
module GT_AURORA_4 (
 CHBONDDONE,
 CHBONDO,
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,
 TXP,
 TXRUNDISP,
 CHBONDI,
 CONFIGENABLE,
 CONFIGIN,
 ENCHANSYNC,
 LOOPBACK,
 POWERDOWN,
 REFCLK,
 REFCLK2,
 REFCLKSEL,
 RXN,
 RXP,
 RXPOLARITY,
 RXRESET,
 RXUSRCLK,
 RXUSRCLK2,
 TXBYPASS8B10B,
 TXCHARDISPMODE,
 TXCHARDISPVAL,
 TXCHARISK,
 TXDATA,
 TXFORCECRCERR,
 TXINHIBIT,
 TXPOLARITY,
 TXRESET,
```

```
TXUSRCLK,
TXUSRCLK2
);
```

## GT\_CUSTOM

```
module GT_CUSTOM (
 CHBONDDONE,
 CHBONDO,
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,
 TXP,
 TXRUNDISP,
 CHBONDI,
 CONFIGENABLE,
 CONFIGIN,
 ENCHANSYNC,
 LOOPBACK,
 POWERDOWN,
 REFCLK,
 REFCLK2,
 REFCLKSEL,
 RXN,
 RXP,
 RXPOLARITY,
 RXRESET,
 RXUSRCLK,
 RXUSRCLK2,
 TXBYPASS8B10B,
 TXCHARDISPMODE,
 TXCHARDISPVAL,
 TXCHARISK,
 TXDATA,
 TXFORCECRCERR,
 TXINHIBIT,
 TXPOLARITY,
 TXRESET,
 TXUSRCLK,
 TXUSRCLK2
);
```

## GT\_ETHERNET\_1

```
module GT_ETHERNET_1 (
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,
 TXP,
 TXRUNDISP,
 CONFIGENABLE,
 CONFIGIN,
 LOOPBACK,
 POWERDOWN,
 REFCLK,
 REFCLK2,
 REFCLKSEL,
 RXN,
 RXP,
 RXPOLARITY,
 RXRESET,
 RXUSRCLK,
 RXUSRCLK2,
 TXBYPASS8B10B,
 TXCHARDISPMODE,
 TXCHARDISPVAL,
 TXCHARISK,
 TXDATA,
 TXFORCECRCERR,
 TXINHIBIT,
 TXPOLARITY,
 TXRESET,
 TXUSRCLK,
 TXUSRCLK2
);
```

## GT\_ETHERNET\_2

```
module GT_ETHERNET_2 (
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
```

```

RXCRCERR,
RXDATA,
RXDISPERR,
RXLOSSOFSYNC,
RXNOTINTABLE,
RXREALIGN,
RXRECCLK,
RXRUNDISP,
TXBUFERR,
TXKERR,
TXN,
TXP,
TXRUNDISP,
CONFIGENABLE,
CONFIGIN,
LOOPBACK,
POWERDOWN,
REFCLK,
REFCLK2,
REFCLKSEL,
RXN,
RXP,
RXPOLARITY,
RXRESET,
RXUSRCLK,
RXUSRCLK2,
TXBYPASS8B10B,
TXCHARDISPMODE,
TXCHARDISPVAL,
TXCHARISK,
TXDATA,
TXFORCECRCERR,
TXINHIBIT,
TXPOLARITY,
TXRESET,
TXUSRCLK,
TXUSRCLK2
);

```

## GT\_ETHERNET\_4

```

module GT_ETHERNET_4 (
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,

```

```
TXP,
TXRUNDISP,
CONFIGENABLE,
CONFIGIN,
LOOPBACK,
POWERDOWN,
REFCLK,
REFCLK2,
REFCLKSEL,
RXN,
RXP,
RXPOLARITY,
RXRESET,
RXUSRCLK,
RXUSRCLK2,
TXBYPASS8B10B,
TXCHARDISPMODE,
TXCHARDISPVAL,
TXCHARISK,
TXDATA,
TXFORCECRCERR,
TXINHIBIT,
TXPOLARITY,
TXRESET,
TXUSRCLK,
TXUSRCLK2
);
```

## GT\_FIBRE\_CHAN\_1

```
module GT_FIBRE_CHAN_1 (
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,
 TXP,
 TXRUNDISP,
 CONFIGENABLE,
 CONFIGIN,
 LOOPBACK,
 POWERDOWN,
 REFCLK,
 REFCLK2,
 REFCLKSEL,
 RXN,
 RXP,
```

```

 RXPOLARITY,
 RXRESET,
 RXUSRCLK,
 RXUSRCLK2,
 TXBYPASS8B10B,
 TXCHARDISPMODE,
 TXCHARDISPVAL,
 TXCHARISK,
 TXDATA,
 TXFORCECRCERR,
 TXINHIBIT,
 TXPOLARITY,
 TXRESET,
 TXUSRCLK,
 TXUSRCLK2
);

```

## GT\_FIBRE\_CHAN\_2

```

module GT_FIBRE_CHAN_2 (
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,
 TXP,
 TXRUNDISP,
 CONFIGENABLE,
 CONFIGIN,
 LOOPBACK,
 POWERDOWN,
 REFCLK,
 REFCLK2,
 REFCLKSEL,
 RXN,
 RXP,
 RXPOLARITY,
 RXRESET,
 RXUSRCLK,
 RXUSRCLK2,
 TXBYPASS8B10B,
 TXCHARDISPMODE,
 TXCHARDISPVAL,
 TXCHARISK,
 TXDATA,
 TXFORCECRCERR,
 TXINHIBIT,

```

```
TXPOLARITY,
TXRESET,
TXUSRCLK,
TXUSRCLK2
);
```

## GT\_FIBRE\_CHAN\_4

```
module GT_FIBRE_CHAN_4 (
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,
 TXP,
 TXRUNDISP,
 CONFIGENABLE,
 CONFIGIN,
 LOOPBACK,
 POWERDOWN,
 REFCLK,
 REFCLK2,
 REFCLKSEL,
 RXN,
 RXP,
 RXPOLARITY,
 RXRESET,
 RXUSRCLK,
 RXUSRCLK2,
 TXBYPASS8B10B,
 TXCHARDISPMODE,
 TXCHARDISPVAL,
 TXCHARISK,
 TXDATA,
 TXFORCECRCERR,
 TXINHIBIT,
 TXPOLARITY,
 TXRESET,
 TXUSRCLK,
 TXUSRCLK2
);
```



## GT\_INFINIBAND\_1

```
module GT_INFINIBAND_1 (
 CHBONDDONE,
 CHBONDO,
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,
 TXP,
 TXRUNDISP,
 CHBONDI,
 CONFIGENABLE,
 CONFIGIN,
 ENCHANSYNC,
 LOOPBACK,
 POWERDOWN,
 REFCLK,
 REFCLK2,
 REFCLKSEL,
 RXN,
 RXP,
 RXPOLARITY,
 RXRESET,
 RXUSRCLK,
 RXUSRCLK2,
 TXBYPASS8B10B,
 TXCHARDISPMODE,
 TXCHARDISPVAL,
 TXCHARISK,
 TXDATA,
 TXFORCECRCERR,
 TXINHIBIT,
 TXPOLARITY,
 TXRESET,
 TXUSRCLK,
 TXUSRCLK2
);
```

## GT\_INFINIBAND\_2

```
module GT_INFINIBAND_2 (
 CHBONDDONE,
 CHBONDO,
 CONFIGOUT,
```

```
RXBUFSTATUS,
RXCHARISCOMMA,
RXCHARISK,
RXCHECKINGCRC,
RXCLKCORCNT,
RXCOMMADET,
RXCRCERR,
RXDATA,
RXDISPERR,
RXLOSSOFSYNC,
RXNOTINTABLE,
RXREALIGN,
RXRECCLK,
RXRUNDISP,
TXBUFERR,
TXKERR,
TXN,
TXP,
TXRUNDISP,
CHBONDI,
CONFIGENABLE,
CONFIGIN,
ENCHANSYNC,
LOOPBACK,
POWERDOWN,
REFCLK,
REFCLK2,
REFCLKSEL,
RXN,
RXP,
RXPOLARITY,
RXRESET,
RXUSRCLK,
RXUSRCLK2,
TXBYPASS8B10B,
TXCHARDISPMODE,
TXCHARDISPVAL,
TXCHARISK,
TXDATA,
TXFORCECRCERR,
TXINHIBIT,
TXPOLARITY,
TXRESET,
TXUSRCLK,
TXUSRCLK2
);
```

## GT\_INFINIBAND\_4

```
module GT_INFINIBAND_4 (
 CHBONDDONE,
 CHBONDO,
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
```

```

RXDATA,
RXDISPERR,
RXLOSSOFSYNC,
RXNOTINTABLE,
RXREALIGN,
RXRECCLK,
RXRUNDISP,
TXBUFERR,
TXKERR,
TXN,
TXP,
TXRUNDISP,
CHBONDI,
CONFIGENABLE,
CONFIGIN,
ENCHANSYNC,
LOOPBACK,
POWERDOWN,
REFCLK,
REFCLK2,
REFCLKSEL,
RXN,
RXP,
RXPOLARITY,
RXRESET,
RXUSRCLK,
RXUSRCLK2,
TXBYPASS8B10B,
TXCHARDISPMODE,
TXCHARDISPVAL,
TXCHARISK,
TXDATA,
TXFORCECRCERR,
TXINHIBIT,
TXPOLARITY,
TXRESET,
TXUSRCLK,
TXUSRCLK2
);

```

## GT\_XAUI\_1

```

module GT_XAUI_1 (
 CHBONDDONE,
 CHBONDO,
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,

```

```
TXBUFERR,
TXKERR,
TXN,
TXP,
TXRUNDISP,
CHBONDI,
CONFIGENABLE,
CONFIGIN,
ENCHANSYNC,
LOOPBACK,
POWERDOWN,
REFCLK,
REFCLK2,
REFCLKSEL,
RXN,
RXP,
RXPOLARITY,
RXRESET,
RXUSRCLK,
RXUSRCLK2,
TXBYPASS8B10B,
TXCHARDISPMODE,
TXCHARDISPVAL,
TXCHARISK,
TXDATA,
TXFORCECRCERR,
TXINHIBIT,
TXPOLARITY,
TXRESET,
TXUSRCLK,
TXUSRCLK2
);
```

## GT\_XAUI\_2

```
module GT_XAUI_2 (
 CHBONDDONE,
 CHBONDO,
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,
 TXP,
 TXRUNDISP,
 CHBONDI,
 CONFIGENABLE,
```

```

CONFIGIN,
ENCHANSYNC,
LOOPBACK,
POWERDOWN,
REFCLK,
REFCLK2,
REFCLKSEL,
RXN,
RXP,
RXPOLARITY,
RXRESET,
RXUSRCLK,
RXUSRCLK2,
TXBYPASS8B10B,
TXCHARDISPMODE,
TXCHARDISPVAL,
TXCHARISK,
TXDATA,
TXFORCECRCERR,
TXINHIBIT,
TXPOLARITY,
TXRESET,
TXUSRCLK,
TXUSRCLK2
);

```

## GT\_XAUI\_4

```

module GT_XAUI_4 (
 CHBONDDONE,
 CHBONDO,
 CONFIGOUT,
 RXBUFSTATUS,
 RXCHARISCOMMA,
 RXCHARISK,
 RXCHECKINGCRC,
 RXCLKCORCNT,
 RXCOMMADET,
 RXCRCERR,
 RXDATA,
 RXDISPERR,
 RXLOSSOFSYNC,
 RXNOTINTABLE,
 RXREALIGN,
 RXRECCLK,
 RXRUNDISP,
 TXBUFERR,
 TXKERR,
 TXN,
 TXP,
 TXRUNDISP,
 CHBONDI,
 CONFIGENABLE,
 CONFIGIN,
 ENCHANSYNC,
 LOOPBACK,
 POWERDOWN,
 REFCLK,
 REFCLK2,
 REFCLKSEL,

```

```
RXN,
RXP,
RXPOLARITY,
RXRESET,
RXUSRCLK,
RXUSRCLK2,
TXBYPASS8B10B,
TXCHARDISPMODE,
TXCHARDISPVAL,
TXCHARISK,
TXDATA,
TXFORCECRCERR,
TXINHIBIT,
TXPOLARITY,
TXRESET,
TXUSRCLK,
TXUSRCLK2
);
```



# **Volume 4: Design Flow**

## ***Virtex-II Pro™ Platform FPGA Developer's Kit***

March 2002 Release







The Xilinx logo shown above is a registered trademark of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

"Xilinx" and the Xilinx logo are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, XC5210 are registered Trademarks of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Bencher, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Bencher, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, Nano-Blaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, Rocket I/O, SelectI/O, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM      IBM Logo      PowerPC      PowerPC Logo      Blue Logic      CoreConnect      CodePack

All other trademarks are the property of their respective owners.

Xilinx does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 2002 Xilinx, Inc. All Rights Reserved.

# *Introduction to Virtex-II Pro™ Design Flow*

---

## **Introduction**

Virtex-II Pro based designs have the PPC405, Rocket I/O™ transceiver, and potentially a large part of the user's system within a single FPGA. These additional capabilities add complexities to how the system is developed and verified. The user must build software applications to be transferred to the hardware domain so that the evolving design can be verified. The PPC405 and the Rocket I/O transceiver require accurate simulation models that integrate into the verification procedure. Designs using the processor are more likely to be collections of various cores rather than a custom design; proper management of design libraries becomes important.

The Virtex-II Pro Design Flow provided in the Developer's Kit is a command-line based design procedure that addresses the additional complexities and simplifies the design process from a user entry point-of-view.

The following is a typical non-Virtex-II Pro design procedure:

1. Create the design in HDL
2. Simulate HDL
3. Synthesize
4. Implement onto FPGA with Xilinx implementation tools
5. Simulate with timing
6. Generate configuration bitstream

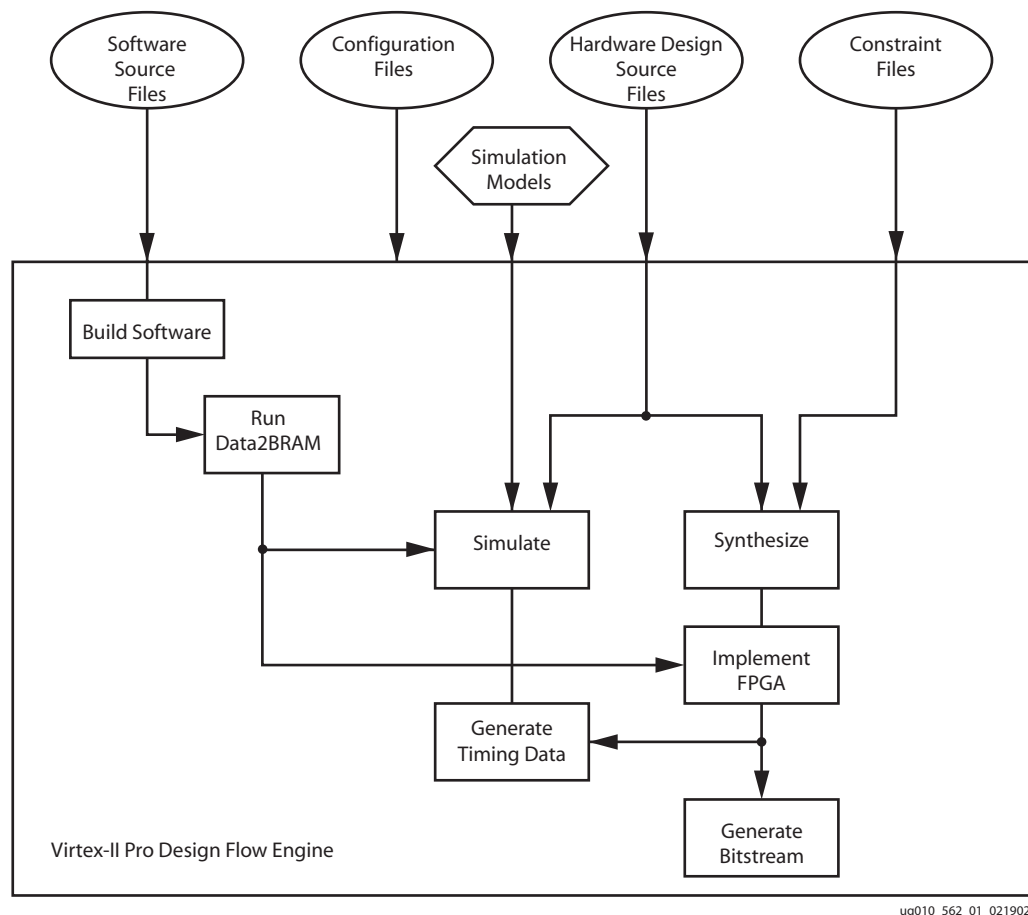
However, there are several issues that need to be addressed from this design process:

1. Integrating software into the design flow
2. Managing in a simple way how IP cores and subsystems are added to a particular design
3. Providing an automation system that relieves much of the burden of integrating the tools with each other.

The most significant issue is the incorporation of software within the hardware design process. The introduction of the PPC405 gives the expectation of running software on the FPGA and, therefore, software must now be integrated into the design flow. Building the application executable itself is still relatively unaltered. The design flow addresses HDL simulation with the software application running on the embedded PPC405 and using the Data2BRAM utility to store some or all of the application within the FPGA's block SelectRAM™ memory. Data2BRAM translates the application executable into constraints that initialize the FPGA block SelectRAM memory with some or all of the software.

The second issue is also a result of the introduction of the PPC405. The user's design appears more like a collection of subdesigns with well-defined interfaces that are connected using IBM CoreConnect™ buses. Since there is likely to be libraries of these subdesigns, there needs to be a simple way to add and remove a subdesign with the user requiring as little work as possible.

The last issue is the integration of each of the specialized EDA tools from various vendors, including Xilinx. A supervisory design flow manager is necessary for the user to avoid maintaining redundant file lists for each separate tool and to avoid making each step of the design flow a confusing multi-step process in itself. For the Developer's Kit, a key issue is to remove much of the burden of dealing with the individual tools from the user. Absolute freedom with each of the tools may be compromised to a degree to simplify the overall process.



ug010\_562\_01\_021902

Figure 1-1: Virtex-II Pro Design Flow

The top of the diagram outside of the box indicate input files for the flow engine. These files are either created by the user or provided in the Developer's Kit (IP cores and simulation models). The activities are described within the flow engine box. Each portion is summarized below and explained in greater detail in the rest of this Volume.

## Design Flow Input Files

### Software Source Files

Software source files are those files required to build the software application. This includes C source files, assembly language files, and map files for the linker.

### Configuration Files

The user must set up the configuration files that effect the design flow engine and all of the programs executed in the course of the flow (i.e. Synplify, Data2BRAM, the simulation tool, MAP, PAR, etc.).

Prominent among the configuration files for the design flow engine are **flow.cfg** and the LST file. These files guide the engine to use appropriate software and hardware files in each phase of the design process. Switching between different simulation models, software applications, and versions of subsystems can be done with very little effort once these two files are fully written.

Other configuration files alter the execution of the tools used in the flow engine. For example, Data2BRAM BMM files describe the relationship between code and physical memories and ModelSim 'do' files can be used to set up waveforms and even execute the simulation.

### Hardware Design Source Files

The standard design files will be HDL source that the user creates to describe custom subsystems of the design, e.g. PLB masters/slaves and OPB masters/slaves. Other source files may be pre-synthesized EDIF and NGO cores that are created by the Xilinx CORE Generator™ utility or from third-party vendors.

### Constraint Files

Constraint files are used to input timing and location constraints for synthesis and place & route.

### Simulation Models

The PPC405 and the Rocket I/O transceivers use Synopsys® SmartModel® simulation models provided by Xilinx. SmartModel simulation models are accurate models of intellectual property and are simulation efficient in terms of performance and resource usage.

Additionally, for the PPC405 there is a Bus Functional Model (BFM) when using IBM CoreConnect technology. A PPC405 BFM is a low-level model that allows the user to specify bus transactions for the PLB and DCR interfaces of the processor. It is extremely useful in design verification when the software application is not available to run on the full SmartModel. It is also useful in debugging certain conditions that are difficult to replicate in a software program.

Also, included in simulation models are vendor-provided models such as memory models and other component models that exist in the system either within the FPGA or outside of it.

## Design Flow Processes

### Build Software

The process of building the software executable is no different than for other embedded software development. The memory locations of the hardware peripherals must

correspond with those expected by the application. Also, when storing program code or data within the FPGA and/or outside of it, the code/data must be at memory addresses where there is physical memory within the system.

## Run Data2BRAM

The Xilinx Data2BRAM program is a utility that translates a software executable into information that can be used in the hardware flow. Data2BRAM initializes memory models (both on-chip and off-chip) for HDL simulation. It also passes constraints to the FPGA implementation tools to initialize the block SelectRAM memory with the desired software code or data that is mapped into FPGA on-chip memory.

## Simulate

During this step, the design is simulated using source HDL, simulation models, and the output of Data2BRAM; based upon its configuration files, the design flow engine determines which files are necessary for this simulation. It is possible to simulate the entire system with the end-application running. However, for trouble-shooting, switching the PPC405 model from the full SmartModel to Bus Functional Model may greatly simplify debugging.

## Synthesize

After verifying functionality, the design flow engine chooses all of the required source HDL files to synthesize into a Virtex-II Pro FPGA. Xilinx or vendor EDIF and NGO cores are passed through as black boxes during this step. There is flexibility in setting design specific options for the synthesis tool.

## Implement FPGA

This multi-step process uses several Xilinx ISE tools: NGDBUILD, MAP, PAR, and TRCE. During execution of these programs, the flow engine combines the results of the Synthesize step, the EDIF and NGO cores, the user constraints, and the Data2BRAM output, then maps the results onto the Virtex-II Pro FPGA.

## Generate Timing Data

After FPGA implementation, timing data can be extracted from the resulting NCD file and a more accurate simulation can be produced. A simulation using this timing data may uncover race conditions within the design or missed timing constraints that result in functional failure.

## Generate Bitstream

When satisfied with the FPGA implementation, this step produces a configuration bitstream and optionally PROM formatted files.

## Further Reading

The Virtex-II Pro™ Developer's Kit contains a wealth of valuable information that will assist you in your design efforts. The documentation contained within the eight volumes is organized to assist you in quickly finding relevant materials. To obtain the most recent revision of this documentation, please see

[http://support.xilinx.com/xlnx/xil\\_tt\\_product.jsp?sProduct=Virtex-II+Pro](http://support.xilinx.com/xlnx/xil_tt_product.jsp?sProduct=Virtex-II+Pro).

### Documentation Provided by Xilinx

*Virtex-II Pro™ Platform FPGA Handbook*

*Virtex-II Pro™ Platform FPGA Developer's Kit Documentation:*

*Volume 1, Advance Product Specification*

*Volume 2, PowerPC® 405 Processor*

*Volume 3, Rocket I/O™ Transceiver User Guide*

*Volume 4, Design Flow*

*Volume 5, Software Development Tools*

*Volume 6, Reference Systems*

*Volume 7, Hardware IP Specifications*

*Volume 8, Software IP and Applications*

### IBM® CoreConnect™ Documentation

The Virtex-II Pro Developer's Kit integrates seamlessly with the IBM CoreConnect Toolkit. This toolkit is not included with the Developer's Kit, but is required if bus functional simulation is desired. The toolkit provides a number of features which enhance design productivity and allow you to get the most from the Developer's Kit. To obtain the toolkit, you must be a licensee of the IBM CoreConnect Bus Architecture. Licensing CoreConnect provides access to a wealth of documentation, Bus Functional Models, Hardware IP, and the toolkit.

Xilinx provides a Web-based licensing mechanism that allows you to obtain the CoreConnect toolkit from our website. To license CoreConnect, use an Internet browser to access [http://www.xilinx.com/ipcenter/processor\\_central/register\\_coreconnect.htm](http://www.xilinx.com/ipcenter/processor_central/register_coreconnect.htm). Once your request has been approved (typically within 24 hours), you will receive an e-mail granting access to a protected website. You may then download the toolkit.

If you prefer, you can also license CoreConnect directly from IBM.

If you would like further information on CoreConnect Bus Architecture, please see IBM's CoreConnect website at <http://www.ibm.com/chips/products/coreconnect>.

Once you have licensed the CoreConnect toolkit, and installed it with the Developer's Kit, the following documents will be available to you in the following locations:

#### IBM CoreConnect Bus Architecture Specifications

*IBM CoreConnect Processor Local Bus (PLB) Architecture Specification*  
see \$CORECONNECT/published/corecon/64bitPlbBus.pdf

*IBM CoreConnect On-chip Peripheral Bus (OPB) Architecture Specification*  
see \$CORECONNECT/published/corecon/OpbBus.pdf

*IBM CoreConnect Device Control Register (DCR) Bus Architecture Specification*  
see \$CORECONNECT/published/corecon/DcrBus.pdf

#### IBM CoreConnect Toolkit Documentation

*PLB Bus Functional Model Toolkit - User's Manual*  
see \$CORECONNECT/published/corecon/PlbToolkit.pdf

*OPB Bus Functional Model Toolkit - User's Manual*

see \$CORECONNECT/published/corecon/OpbToolkit.pdf

*DCR Bus Functional Model Toolkit - User's Manual*

see \$CORECONNECT/published/corecon/DcrToolkit.pdf

*CoreConnect Test Generator - User's Manual*

see \$CORECONNECT/published/corecon/ctg.pdf

**Note:** \$CORECONNECT is an environment variable that is created when installing the Developer's Kit or CoreConnect Toolkit.

## Software Development Documentation

There are many sources of documentation available for those who wish to learn more about Software Development. It is recommended that a web search be conducted using a favorite search engine for keywords such as "PowerPC+Software+Development". Alternatively, a technical bookstore should be able to provide many valuable resources. The books listed below are a very small fraction of those available.

### Books About Programming in C

There are many good books about the C and C++ programming languages. A few of these are listed below:

#### Books about C

Kernigham Brian, Ritchie Dennis. 1988. *The C Programming Language*. Prentice Hall

Wang Paul. 1992. *An Introduction to ANSI C on Unix*. Wadsworth.

Kumar Ram, Agrawal Rakesh. 1992. *Programming in ANSI C*. West Publishing Company.

#### Books about C++

Stroustrup Bjarne. 1991. *The C++ Programming Language*. Addison-Wesley.

Lafore Robert. 1995. *Object-Oriented Programming in C++*. Waite Group Press.

### Online Documents About Programming in C

The Internet offers plenty of documentation about how to program in C and C++. One of the best approaches to finding documentation online is to use a search engine (such as Google, <http://www.google.com>) and search on "introduction to programming in C."

# *Design Flow Overview*

---

## Introduction

A wide variety of software and hardware tools are used to build a Virtex-II Pro™ design. The design flow is a tool chain methodology that exists to simplify the entire design process by providing integration between the tools and automating tasks. See **Related Documentation** for details on how a software application is built, how a hardware system is created, or how to use a given tool. The main focus of the design flow is integrating the programs with each other to accomplish the system design.

The system design process can be loosely divided into the following tasks:

- Builds the software application
- Simulates the hardware description
- Simulates the hardware with the software application
- Simulates the hardware into the FPGA using the software application in on-chip memory
- Runs timing simulation
- Configures the configuration bitstream for the FPGA

## Related Documentation

For more information about creating a software application, refer to:

- **PPC405 Software: Demonstration and Test Applications**
- **Stand-Alone Board Support Package**
- **GNU Software Tool Chain**

For information about extending and modifying hardware designs, refer to:

- **Simple PPC405 Reference System**
- **Embedded PPC405 Reference System**

For more information about the **make** utility, see:

<http://www.gnu.org/software/make/make.html>

For more information about Perl, see: <http://www.perl.com/pub/q/documentation> and <http://www.perldoc.com/>

## Software

This section describes the bulk of the software portion of the design flow.

The software and hardware segments of the design flow only loosely interact with each other through a well-defined interface. Consequently, the two can be logically divided and



worked on somewhat independently. Prior to doing this, the user should determine the intellectual property that will exist in the system and the allocation of system memory.

Figure 2-1 shows the steps that build a software executable that writes “Hello World” to a UART. The **make** utility is used to automate the software build process, gathering the necessary source files, compiling them and linking them together to generate an executable.

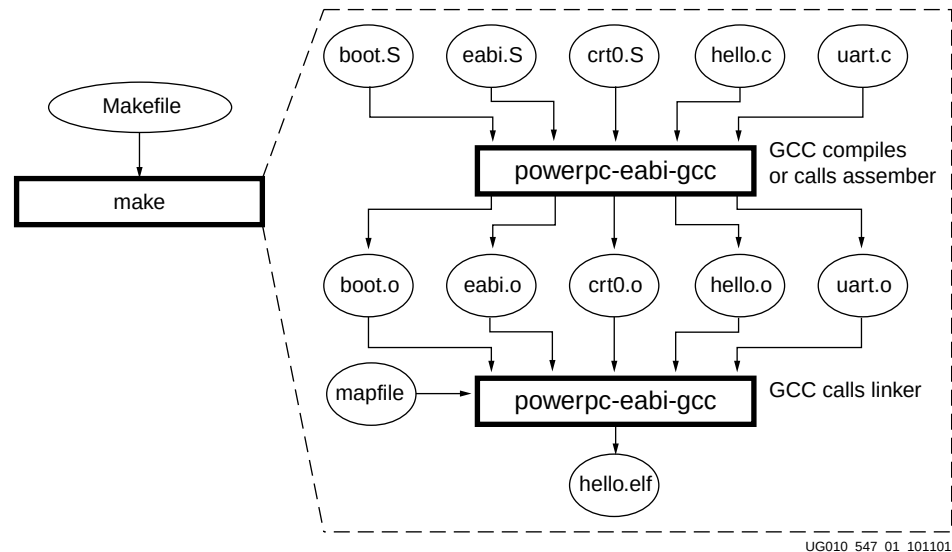


Figure 2-1: Steps for Building Hello.elf

The sources listed at the top include those specific to this application and some provided by the kit. The files supplied with the kit are known as the Board Support Package (BSP), a set of software modules combined into the **bsp.a** library. On the one hand, the BSP offers an interface to peripheral devices and to low-level PowerPC core functions. A stand-alone application uses this interface to access the hardware. On the other hand, the BSP is required when an application is linked with the C library. In this case the BSP provides functionality that allows the C library to access the hardware. For more information on the BSP, see the *Stand-Alone Board Support Package* document.

Both **C** and **assembler** source files are compiled and assembled into object files using GNU **GCC**. The object files and any libraries are linked together with a map file (or link script) again using **GCC**. The map file (or link script) distributes the application’s instruction code and data segments to various memory locations in the system. The memory locations should correspond directly to the physical memory in the system, either on-chip or off-chip. The output of the linker is an executable file that is in ELF format. This executable is sometimes referred to as an ELF file or **.elf**. This file is used extensively in the hardware portions of the design flow.

The remainder of this section describes programs used to build the software. For more information on the software process, refer to **Related Documentation**.

## make

The **make** utility automates the build process. It reads the commands to perform from the Makefile and executes them step-by-step. **make** has the ability to step over commands that need not be executed since the result of the command already exists. This behavior speeds up the build process.

The Makefile contains targets. When **make** is called without parameters, the target **all** is executed. A specific target can be executed by giving its name as a parameter to the **make** program.

Example 1: Call **make**, build the target **all** in the current directory, and use the Makefile:

```
$ make
```

Example 2: Call **make**, build first the target **clean** and then the target **all** in the current directory, and use the Makefile:

```
$ make clean all
```

Example 3: Call **make**, build the target **all** in the current directory, and use the file named **makefile.mak** as the Makefile:

```
$ make -f makefile.mak
```

Example 4: Call **make**, build the target **clean** in the **/tmp** directory, and use the Makefile (in the **/tmp** directory):

```
$ make -C /tmp
```

## GCC

**GCC** acts as a compiler, translating files written in C into a form that can be understood and executed by the target processor. **GCC** also functions as the user interface to the GNU assembler and to the GNU linker, calling the assembler and the linker with the correct parameters.

Example 1: Compile all C source files (\*.c), assemble all assembler source files (\*.S), and generate the executable **hello.elf**. In this example, **GCC** compiles the C source files, initiates the assembler to assemble the assembler source files, and calls the linker to generate the executable **hello.elf**. The linker uses a built-in map file since no other map file is specified.

```
$ powerpc-eabi-gcc -o hello.elf *.c *.S
```

Example 2: Compile all C source files (\*.c) and assemble all assembler source files (\*.S) to object files (\*.o). Link the object files and generate the executable **hello.elf**. In this example, the intermediary object files are generated.

```
$ powerpc-eabi-gcc -c *.c *.S
$ powerpc-eabi-gcc -o hello.elf *.o
```

Example 3: Compile all C source files (\*.c) and assemble all assembler source files (\*.S) and generate the executable **hello.elf**. **GCC** compiles the C source files, initiates the assembler to assemble the assembler source files, and calls the linker to generate the executable **hello.elf**. The linker uses the specified map file as linker script.

```
$ powerpc-eabi-gcc -Wl,-T,mapfile -o hello.elf *.c *.S
```

Compiler flags, shown in [Table 2-1](#), influence the code generation of **GCC**.

Table 2-1: Compiler Flags

| Compiler Flag | Influence                                                                                                                                                                                                                                      |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -c            | Generate intermediary object files.                                                                                                                                                                                                            |
| -o <name>     | Name the output file. This can be the name of the executable or the object file.                                                                                                                                                               |
| -gdwarf       | Generate debugging information in the dwarf format. This debugging information is included in the object files and in the executable. Do not use with one of the -O optimization levels since optimized code might become impossible to debug. |
| -Ox           | Generate optimized code. X is a place holder for the optimization level. Possible optimization levels range from 0 to 3 or s. Higher numbers indicate optimization for higher speed. S indicates optimization for smaller space.               |
| -mcpu=403     | Generate code for the IBM 403 processor. Note, <b>GCC</b> does not yet offer a compiler switch to generate code for the 405. This compiler switch also turns on code generation for floating point emulation in software.                      |

Table 2-1: Compiler Flags

| Compiler Flag | Influence                                                                                                               |
|---------------|-------------------------------------------------------------------------------------------------------------------------|
| -msdata=eabi  | Place small data in the small data areas following the EABI specification.                                              |
| -Wall         | Generate all possible compiler warnings.                                                                                |
| -Wl           | Pass a parameter to the linker, e.g. -Wl, -T, mapfile tells the linker to use the file <b>mapfile</b> as linker script. |
| -lc           | Use the C library during linking.                                                                                       |

## Binutils

**Binutils** is a collection of the following binary utilities:

- The portable GNU assembler (**as**)
- The archive utility (**ar**) for creating, modifying and extracting from libraries
- The GNU linker (**ld**)
- The utility (**objdump**) to display information from object files and executables

### as

The GNU assembler utility, **as**, processes assembler source code into object code. Normally, **as** is not executed manually but is invoked by **GCC** during compilation.

**Note:** Consult additional documentation about **as** for stand-alone use.

### ar

The GNU archiver utility, **ar**, creates, extracts, and modifies from libraries. A library consists of a set of object files combined into one single file. The following are examples of create and extract :

Example 1: Create the library **hello.a** consisting of all object files:

```
$ powerpc-eabi-ar -r hello.a boot.o eabi.o crt0.o hello.o uart.o
```

Example 2: Extract the object files **boot.o** and **crt0.o** from the library **hello.a**.

```
$ powerpc-eabi-ar -x hello.a boot.o crt0.o
```

### ld

The GNU linker, **ld**, links object files and libraries into an executable. Normally, **ld** is not executed manually but is invoked by **GCC**.

**Note:** Consult additional documentation about **ld** for stand-alone use.

Example 1: Use **GCC** to link a set of object files (\*.o) into the executable **hello.elf**. Tell the linker to use the specified map file as linker script.

```
$ powerpc-eabi-gcc -Wl, -T, mapfile -o hello.elf *.o
```

Example 2: Use **GCC** to link a set of object files (\*.o), the board support package (**bsp.a**) and the C library into the executable **hello.elf**. Tell the linker to use the specified mapfile as linker script.

```
$ powerpc-eabi-gcc -Wl, -T, mapfile -o hello.elf *.o -lc bsp.a
```

The map file (or linker script) tells the linker how to combine object files and where to map them in the memory space. Below is a fairly complete example of a map file:

```
1. STACKSIZE = 0x400;
2. MEMORY
3. {
4. bram :ORIGIN = 0xfffffe000, LENGTH = 0x2000-4
5. boot :ORIGIN = 0xfffffffffc, LENGTH = 4
6. }
```

```

7.
8. ENTRY(_boot)
9.
10. SECTIONS
11. {
12. .text :{ *(.text) } > bram
13. .data :{ *(.data) } > bram
14. .stack :
15. {
16. . = ALIGN(8);
17. . = . + STACKSIZE;
18. } > bram
19. __stack = ADDR(.stack) + sizeof(.stack);
20. .boot :{ *(.boot) } > boot
21. }

```

In line 1, a constant of 1 KB for the stack size is defined. Lines 2 to 6 specify the memory map. In this example, the memory consists of two regions. The first region is named **bram**, starts at address 0xFFFFE000 and has a size of 8188 bytes. The second region is named **boot**, starts at address 0xFFFFF000 (the location of the reset vector on the PPC405) and has a size of 4 bytes.

In line 8, the linker is told that the entry point to the program is named **\_boot**. This name must exist in one of the object files (e.g. in **boot.o**). The lines 10 to 21 tell the linker how the different sections of the object files have to be mapped into memory. In line 12, the code is filled into the memory region named **bram**, and in line 13 the same is done for the data.

In lines 14 to 19, space for the stack is reserved. First, in line 16, the stack is aligned to an 8-byte boundary. This is done by aligning the dot (location pointer) to eight bytes. In line 17, the dot is increased by the size of the stack. In line 18, the stack is mapped into the memory region named **bram**. Finally, in line 19, a constant is defined that points to the top of the stack. This constant can be accessed from the initialization code, e.g. **crt0.o**, to set up the stack pointer register. In line 20, the section containing the boot code is mapped to the memory region named **boot**.

## objdump

The GNU **objdump** utility displays information from object files and executables. It allows the user to inspect file headers, to browse the contents of an object file or an executable, and to disassemble the object code. If **objdump** is used on an executable, the mapping of the sections into memory can be inspected.

Example 1: Display the header information of the object file **hello.o** and the executable **hello.elf**.

```

$ powerpc-eabi-objdump -x hello.o
$ powerpc-eabi-objdump -x hello.elf

```

Example 2: Disassemble the executable **hello.elf**.

```

$ powerpc-eabi-objdump -d hello.elf

```

Example 3: Show all information about the object file **hello.o**

```

$ powerpc-eabi-objdump -s hello.o

```

## GDB

GDB is a source-level debugger for C, C++, and Fortran. Consult additional documentation for more information about **gdb**.

## Cross-Compilation and the **powerpc-eabi** Target

GNU software is normally distributed as source code. For your convenience, **GCC**, **binutils**, the **newlib** C library, and **GDB** have been cross-compiled from the source code for a **powerpc-eabi** target and are ready to use.

The **powerpc-eabi** target follows the EABI specification. The application's programmer must be aware of this in a few points:

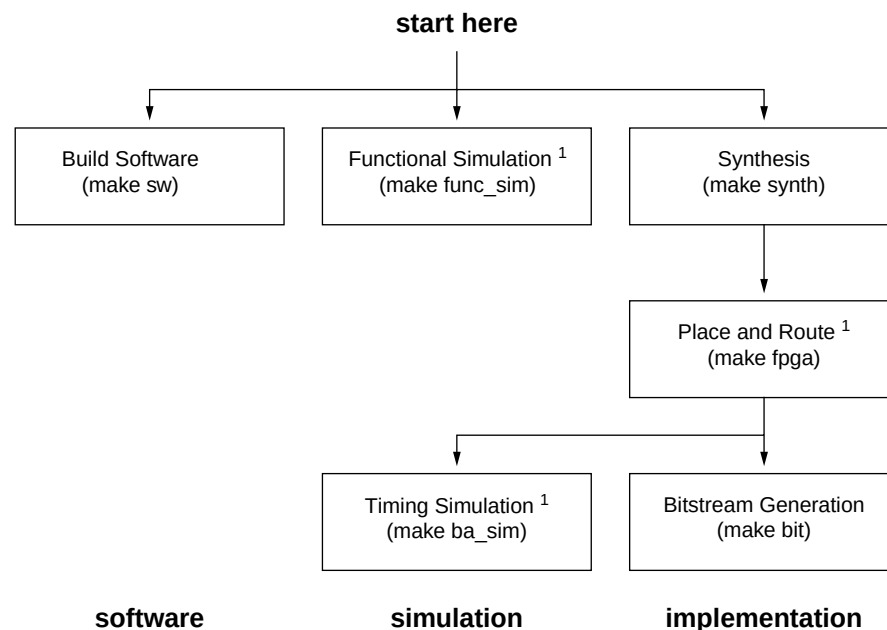
- Small data sections: The compiler references small data relative to two registers.
  - The register R13 has to point to the readable and writeable **.sdata** and **.sbss** data sections.
  - The register R2 has to point to the read-only **.sdata2** data section.
- **\_\_eabi**: The compiler inserts a call to **\_\_eabi** into the **main()** function. This is the place where registers R2 and R13 can be set up.
- Stack: The stack has to be set up on an 8-byte boundary.

More information on the EABI standard can be found in the following two documents:

- Developing *PowerPC Embedded Application Binary Interface (EABI) Compliant Programs* (IBM Microelectronics) [http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569970071B0D6/\\$file/eabi\\_app.pdf](http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569970071B0D6/$file/eabi_app.pdf)
- *Power PC Embedded Application Binary Interface* (Motorola) <http://www.esofta.com/pdfs/ppceabi.pdf>

## Hardware

The hardware portion of the design flow is shown in **Figure 2-2**.



**Note:**

1. This step may optionally include building and using the software executable

ug010\_547\_02\_021202

**Figure 2-2: Hardware Overview**

The ultimate goal of the hardware design flow is to generate a bitstream for the FPGA portion of the system. To achieve this, a **make** utility is used to automate both simulation and implementation of the FPGA. A **Perl** script named '**genscript**' is also used to handle some of the more complex issues. (**Perl** is a scripting language suited for text manipulation.)

The typical sequence of events is to create an HDL (only Verilog is supported at this time) description of the system. The design kit provides a number of HDL cores that can be combined into a PPC405 system. The system is simulated using simulation models for components external to the FPGA. The simulation can be done without the availability of the software executable by using the Bus Functional Model (BFM) of the PPC405 and specifying bus cycles to execute. However, specifying bus cycles requires that tests be very simple. If the software is available, much more can be simulated. The executable ELF file is converted using the Xilinx utility **data2bram** into Verilog files that can be inserted into the simulation.

After the system simulates correctly, the system can be implemented onto the FPGA. The HDL source files are first converted into an EDIF file with a logic synthesis tool. Then, the EDIF file is placed onto the FPGA using the Xilinx implementation tools to produce a Xilinx NCD file. The NCD file can be converted into Verilog files for a timing simulation based upon the routed FPGA design. Otherwise, the user can generate a bitstream and PROM files for the NCD for direct testing on an actual FPGA.

Each step in [Figure 2-2](#) has a specific **make** target associated with it. These targets will be fully detailed later in this document.

## Targets

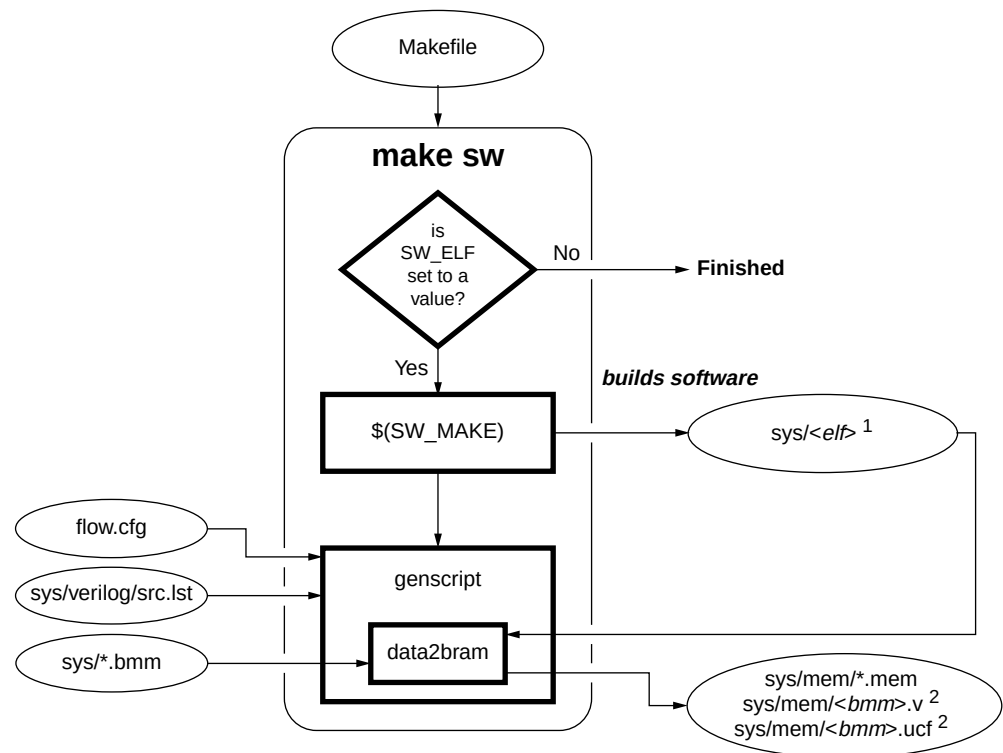
Simulation, synthesis, and FPGA implementation are all accomplished by preparing the necessary source files and executing the **make** program with a single target that indicates which task is to be performed. Typing **make** with no target in the **\$V2PRO/platforms/<name of system>** will give a full list of possible arguments. The primary targets are listed in [Table 2-1](#).

**Table 2-1: Primary Make Targets**

| Target   | Description                                                                                |
|----------|--------------------------------------------------------------------------------------------|
| sw       | Generates software executable and memory initialization files for simulation and synthesis |
| func_sim | Performs functional simulation on pre-synthesis HDL                                        |
| synth    | Synthesizes the design                                                                     |
| fpga     | Performs place and route of the design                                                     |
| ba_sim   | Performs back-annotated timing simulation                                                  |
| bit      | Generates a bitstream and PROM files                                                       |

A single invocation of **make** with one of the above targets initiates a series of actions. The following diagrams show for each target: the tools used, source files required, intermediate files that are generated, and output files that are created. The top-level **Makefile** never needs to be modified by the user. The files on the left side of each diagram are input files provided by the user that are modified. The files on the right side of each diagram are output files generated by the tools. These output files may be fed into a later tool or may be the final desired output.

For clarity, input files are shown with specific paths that correspond to the reference systems. Although the flow does not restrict file locations and file names except for **flow.cfg** and **syn/synplify.opt**, it is strongly recommended that the user conform with these locations and names.



#### Notes:

- 1: *<elf>* is the same file name as \$(SW\_ELF)
- 2: *<bmm>* is the name of the .bmm file used

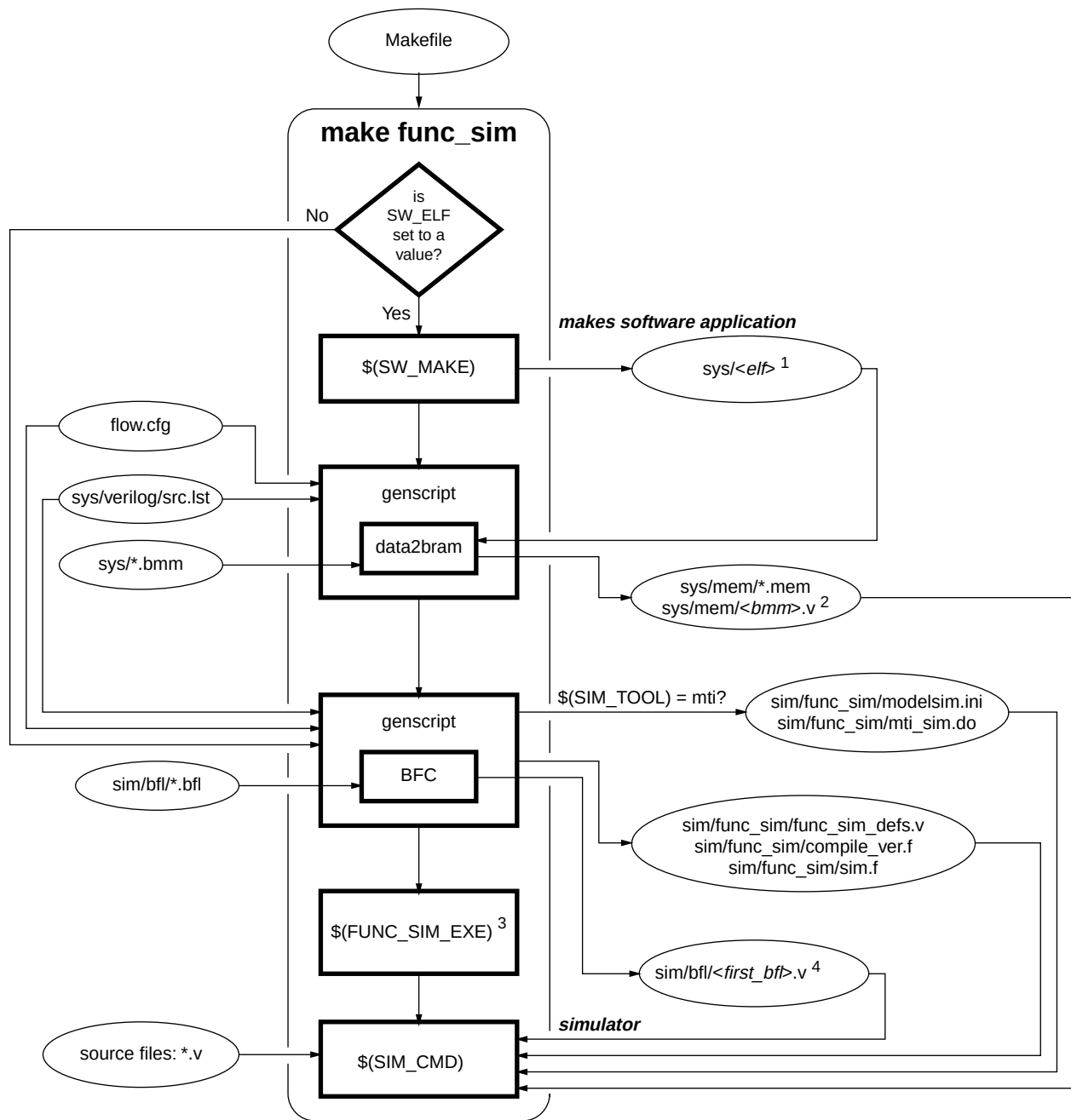
UG010\_547\_03\_101201

Figure 2-3: Generating Software Executable and Memory Initialization Files

At times, it is useful to only generate the software executable and the simulation and implementation files that are derived from it. The software executable can then be downloaded to a PPC405 using debugging programs such as GDB. The other derived files can be examined to ensure that **data2bram** BMM files are correctly specified. In the case of the other **make** targets, this process is done automatically within that target and is tailored for that particular function. For instance, in the description of **func\_sim**, the software executable is rebuilt and only simulation files are derived from the executable.

This **make** target executes the **MAKE\_SW** command line to generate the software executable and copies it locally to the **sys** directory. The command line is typically a call to a remote makefile, as described in the Software section in this document. If **data2bram** BMM files are specified in the **src.lst** file, **.mem** files, simulation files, **.ucf** files and all other **data2bram** output files may be created based upon the executable. (For additional information, see the *Using Data2BRAM* document.) All of these derived files are located in the **sys/mem** directory.

## func\_sim



UG010\_547\_04\_101101

**Notes:**

- 1: *<elf>* is the same file name as \$(SW\_ELF)
- 2: *<bmm>* is the name of the .bmm file used
- 3: runs any optional executables prior to invoking simulator (e.g. start\_terminal)
- 4: *<first\_bfl>* is the base name (without .bfl suffix) of the first .bfl file that is used (Boolean is true)

**Figure 2-4: Performing a Functional Simulation**

Functional simulation is the simulation of pre-synthesis Verilog files and behavioral simulation Verilog files. There are two basic types of functional simulation based upon the desired simulation model for the PPC405 CPU: PPC405 BFM (Bus Functional Model) and



PPC405 SWIFT model. In addition to the two basic types, there is also the option of pre-loading external memory and block SelectRAM™ memory from the software executable. Prior to invoking **make func\_sim**, the user must have all the input files shown in the diagram ready. Information about **flow.cfg** and **src.lst** are described elsewhere in this document in full detail.

The following steps are specific to the functional simulation target:

1. State which simulation tool is being used. Currently, two simulation tools are supported: ModelSim and Verilog-XL. It is possible that the Verilog-XL setting will also allow other simulation tools with similar invocation to be used; however, these tools have not been tested. The choice of simulation tool is made in **flow.cfg**.
2. Set the **SIM\_TOOL** variable to either **mti** or **vx1**. The user must also state the simulation tool executable by assigning **SIM\_CMD** to the name of the executable (typically **vsim** for ModelSim and **verilog** for Verilog-XL). This variable is introduced more for Verilog-XL where the user may be compiling in their own PLIs to create a new executable that may not be named **verilog**. In the case of other executables for ModelSim, the standard executable names are assumed (e.g. **vlib**, **vlog**, ...).
3. Specify the command-line options for these executables.
  - a. The **FUNC\_SIM\_VERILOG\_OPT** variable allows the user to set what Verilog-specific command-line options are used. This is necessary for ModelSim as the simulation is a two step process, compilation and simulation.
  - b. The **FUNC\_SIM\_VERILOG\_OPT** is the command-line option for the compilation executable **vlog**. For example, the user may be using Verilog library directories that require a '+libext+' directive – this is the appropriate place for it.
  - c. **FUNC\_SIM\_OPT** is the command-line option for the simulation executable. For ModelSim, the executable is typically **vsim**. In the case of Verilog-XL, both **FUNC\_SIM\_VERILOG\_OPT** and **FUNC\_SIM\_OPT** are added as options to the simulation command.
  - d. As a final option for simulation, the user may specify a command to run prior to running the simulator by setting the **FUNC\_SIM\_EXE** variable. In this kit, the typical usage is to start the UART terminals that interact with the simulation by assigning **start\_terminal** to **FUNC\_SIM\_EXE**.
4. Specify the options directly related to the system. The user chooses which PPC405 simulation model to use by editing the **flow.cfg** file and assigning the **PPC\_MODEL** to either **swift** or **bfm**. If a software application is being used (more typical with a SWIFT model simulation), the user also assigns values to **SW\_MAKE** and **SW\_ELF**. **SW\_MAKE** should be set to the command line that generates the PowerPC executable **SW\_ELF**. During the **make** process, this executable will be copied locally and converted into Verilog simulation code using the **data2bram** utility. A BMM file is required to describe how the software application is distributed among the block SelectRAM memory and external memories. Information on BMM files and **data2bram** are located in the *Using Data2BRAM* document.
5. Specify the top-level simulation module.
  - a. The Xilinx simulation module **glbl** and the Bus Functional test vector module (output of the Bus Functional Compiler if Bus Functional Language files are being used in the simulation) are automatically added as top-level modules.
  - b. **SIM\_TOP** is the name of the top-level simulation module that needs to be specified in some tools.
6. The **flow.cfg** file points to the top-level LST file with the **SRC\_LST** variable – typically pointing to **src/verilog/src.lst**. All **.bmm**, **.bfl**, and **.v** files should be represented within the **.lst** file. The syntax for this file is described in **LST File (src.lst)**, page 1179.

The Boolean expressions allow control over which files are used in the functional simulation versus other simulations or synthesis.

7. Update all of the **.bmm**, **.bfl**, and **.v** files, prior to executing **make func\_sim**. When using ModelSim, scripts can be listed that will be run after compiling the design. These **.do** files can be used to accomplish tasks such as bringing up a waveform display or beginning the simulation.
8. After executing **make func\_sim**, the Makefile first checks whether **SW\_ELF** is set to a file. If it is, the **SW\_MAKE** command is executed to (re)generate the application **SW\_ELF**. Then, **SW\_ELF** is copied to the local **sys** directory and **data2bram** is run using the ELF file and **.bmm** file indicated in the **.lst** file. The output files generated include **.mem** files for initializing memory models and a Verilog file containing initialization strings for the FPGA's block SelectRAM memory. The Verilog file is named after the first **.bmm** file used in the **.lst** file (name is constructed by replacing **.bmm** with **.v**).
9. Next in the **make func\_sim**, any BFL (Bus Functional Language) files are compiled into a single Verilog stimulus file. Since the BFC (Bus Functional Compiler) must be executed in the same directory where the **.bfl** files are, all such files are copied locally to the **sim/bfl** directory. Again, the name of the first used **.bfl** file in the **.lst** file determines the name of the output file (name is constructed by replacing **.bfl** with **.v**). The output file is likewise in the **sim/bfl** directory and is automatically included in the simulation – there is no need to list the resulting Verilog file in the **.lst** file (in fact, that would cause an error). Generally, all intermediate files generated by the design flow should not be included in the **.lst** file, only user created files should be present.
10. Once these peripheral tasks are completed, the **.lst** file is read to find out the bulk of HDL files being used for a functional simulation. In this stage, several files are created (all in the **sim/func\_sim** directory): **func\_sim\_defs.v**, **compile\_ver.f**, **sim.f**. The **func\_sim\_defs.v** automatically adds *define* definitions to the simulation based upon the **flow.cfg** variable definitions (refer to **Automatic define Statements**, page 1181).  
**Note:** This as a potential problem if there are similarly named variables.  
 This file is always included, there is no way to exclude it. The definitions in the file can be used to optionally include certain segments of HDL code depending on the type of simulation being done. The **compile\_ver.f** file is a complete listing of Verilog files for this simulation, as well as any compile options being used. The **sim.f** file is a complete listing of the simulation options.
11. In the final pre-processing step, simulation tool-specific files are created. In the case of Verilog-XL, no more files are necessary. In the case of ModelSim, a standard **modelsim.ini** released with this kit is copied to the local **sim/func\_sim** directory. Additionally, a ModelSim script file **mti\_sim.do** is created and is used to run the simulation. This script will create a local work directory **work** within **sim/func\_sim** for this simulation. It also compiles the Verilog modules and runs any **.do** scripts used in the **.lst** file.
12. At this point, all of the required simulation files are ready. The **make func\_sim** process continues by running the **FUNC\_SIM\_EXE** command and then invoking the simulation tool **SIM\_CMD** with the appropriate run arguments.

## synth

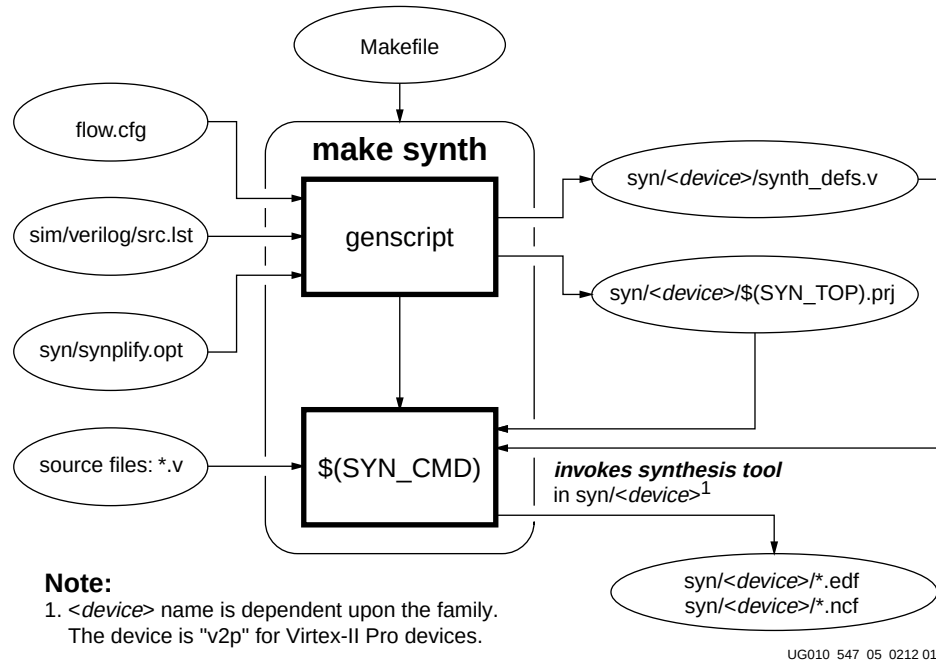


Figure 2-5: Synthesizing the Design

The synthesis process is relatively simple. The HDL files are compiled, and an **.edf** (and potentially an **.ncf**) is created.

The preparation prior to **make synth** is to first update the **flow.cfg**. There are several variables that relate to synthesis: **SYN\_TOOL**, **SYN\_CMD**, **SYN\_OPT**, **SYN\_TOP**, **FAMILY**, **DEVICE**, **PACKAGE**, and **SPEEDGRADE**. **SYN\_TOOL** is the synthesis tool to use; currently only Synplify™ is supported. **SYN\_CMD** is the name of the synthesis executable and **SYN\_OPT** are the optional command line arguments for this executable. The **SYN\_TOP** variable specifies the top-level module to synthesize – the resulting output files will also be **\$(SYN\_TOP).edf** and **\$(SYN\_TOP).ncf**.

**Note:** The **FAMILY**, **DEVICE**, **PACKAGE**, and **SPEEDGRADE** variables refer to the target FPGA device.

All files used for synthesis should be listed in the **.lst** file. A file's Boolean expression should be chosen such that it is used during synthesis.

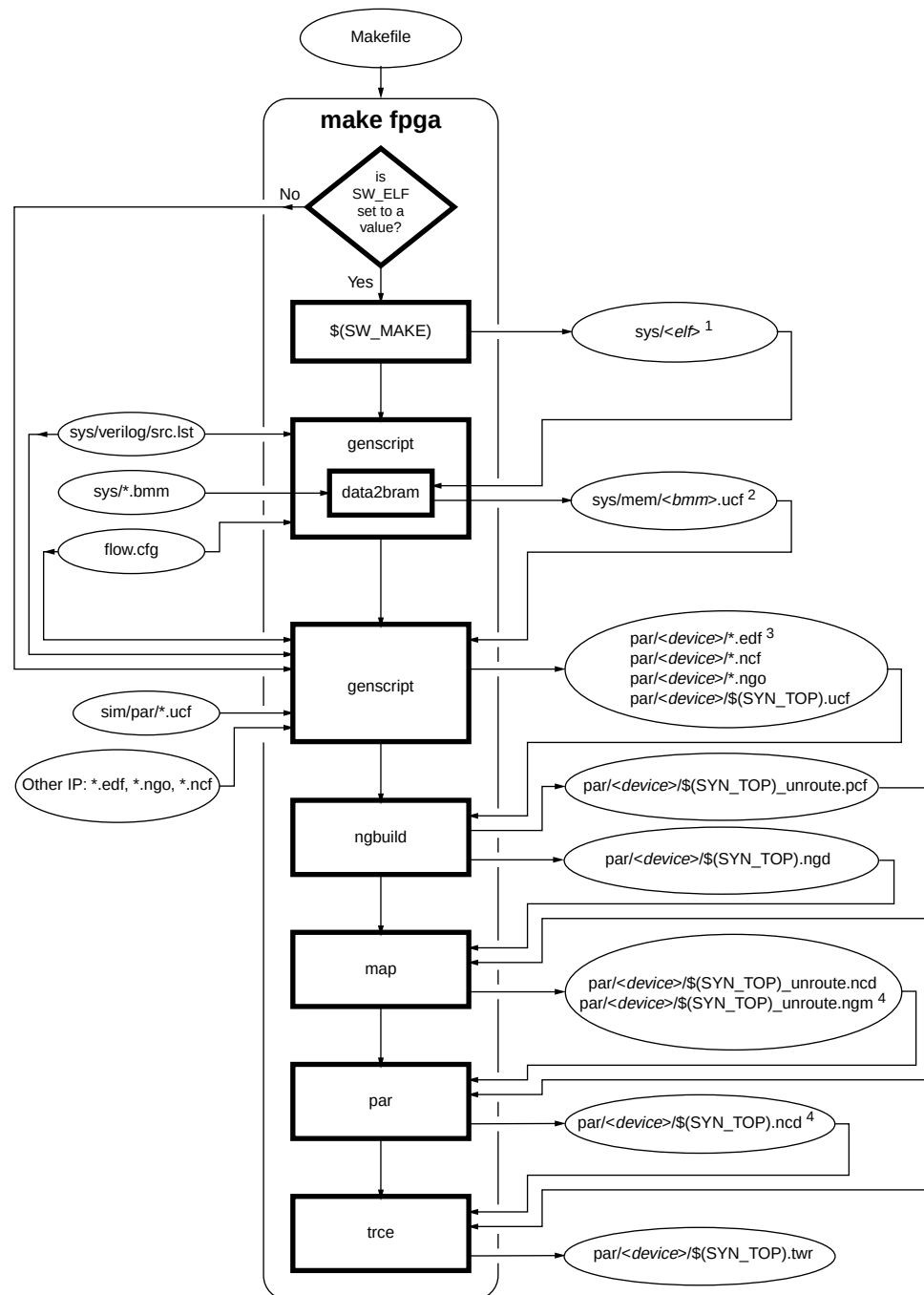
For Synplify, additional options that are normally in the **.prj** file can be specified in **syn/synplify.opt**. Certain options are already set and should not be included within this file. These preset options are described in this document (See **Synplify - syn/synplify.opt**, page 1181).

Upon executing **make synth**, the first step that occurs is that **syn/<device>/synth\_defs.v** is created. This file adds **`define** statements based upon the **flow.cfg** variable definitions (See **Automatic `define Statements**, page 1181). The inclusion of this file is automatic with no way to exclude it.

When using the Synplify synthesis program, the second step generates the project file **syn/<device>/\$(SYN\_TOP).prj** (where <device> is FPGA-family dependent). Commands to add source files are included along with the preset options described in the **syn/synplify.opt** description section. Additionally, the contents of **syn/synplify.opt** are appended to the end of the project file.

The final step of **make synth** invokes the synthesis tool using the project file, generating **\$(SYN\_TOP).edf** and potentially **\$(SYN\_TOP).ncf**, if constraints are specified to be written out.

## fpga


**Notes:**

- 1: `<elf>` is the same file name as `$(SW_ELF)`
- 2: `<bmm>` is the name of the .bmm file used
- 3: `<device>` is dependent upon the family. The device is "v2p" for Virtex-II Pro devices
- 4: `$(SYN_TOP).ncd` and `$(SYN_TOP)_unroute.ngm` are copied to `sim/ba_sim` for use in `make ba_sim`

UG010\_547\_06\_021302

Figure 2-6: Placing and Routing the Design

The **make fpga** process runs the Xilinx FPGA implementation tools on the result of **make synth**. The EDIF produced by the latter is combined with any other EDIF or NGO files (as

in the case of pre-synthesized intellectual property) to form a complete system is to be implemented onto the FPGA using the implementation tools **ngdbuild**, **map**, **par**, and **trce**. On-chip memory may optionally be loaded with all or part of a software application.

The pertinent variables within **flow.cfg** that relate to this process are: **NGDBUILD\_OPT**, **MAP\_OPT**, **PAR\_OPT**, and **TRCE\_OPT**. These are optional command-line arguments that alter the behavior of the various programs. Some command-line switches are already inferred by the design flow and should not be added to these option variables. These inferred switches are listed in [Table 2-2, page 1176](#).

If block SelectRAM components are initialized with the contents of the software application, the **SW\_MAKE** and **SW\_ELF** variables must be defined within **flow.cfg**. The **SW\_MAKE** variable is the command that will generate the **SW\_ELF** file.

Next, all required **.bmm**, **.edf**, **.ngo**, **.ncf**, and **.ucf** files should be described in the **.lst** file (typically the **.lst** file is **sys/verilog/src.lst**). The **syn/<device>/\$(SYN\_TOP).edf** and **syn/<device>/\$(SYN\_TOP).ncf** files are not added since they are considered output files of the design flow and are consequently automatically included. Various cores that are black boxes in the main design can be selected by specifying the **.edf** or **.ngo** files along with any constraints in **.ncf** files. User constraint (**.ucf**) files listed in the **.lst** file will be concatenated with the block SelectRAM initialization strings to form a single composite **.ucf** file. Therefore, the **.ucf** files should not attempt to initialize the same block SelectRAM components that are used to store the software executable.

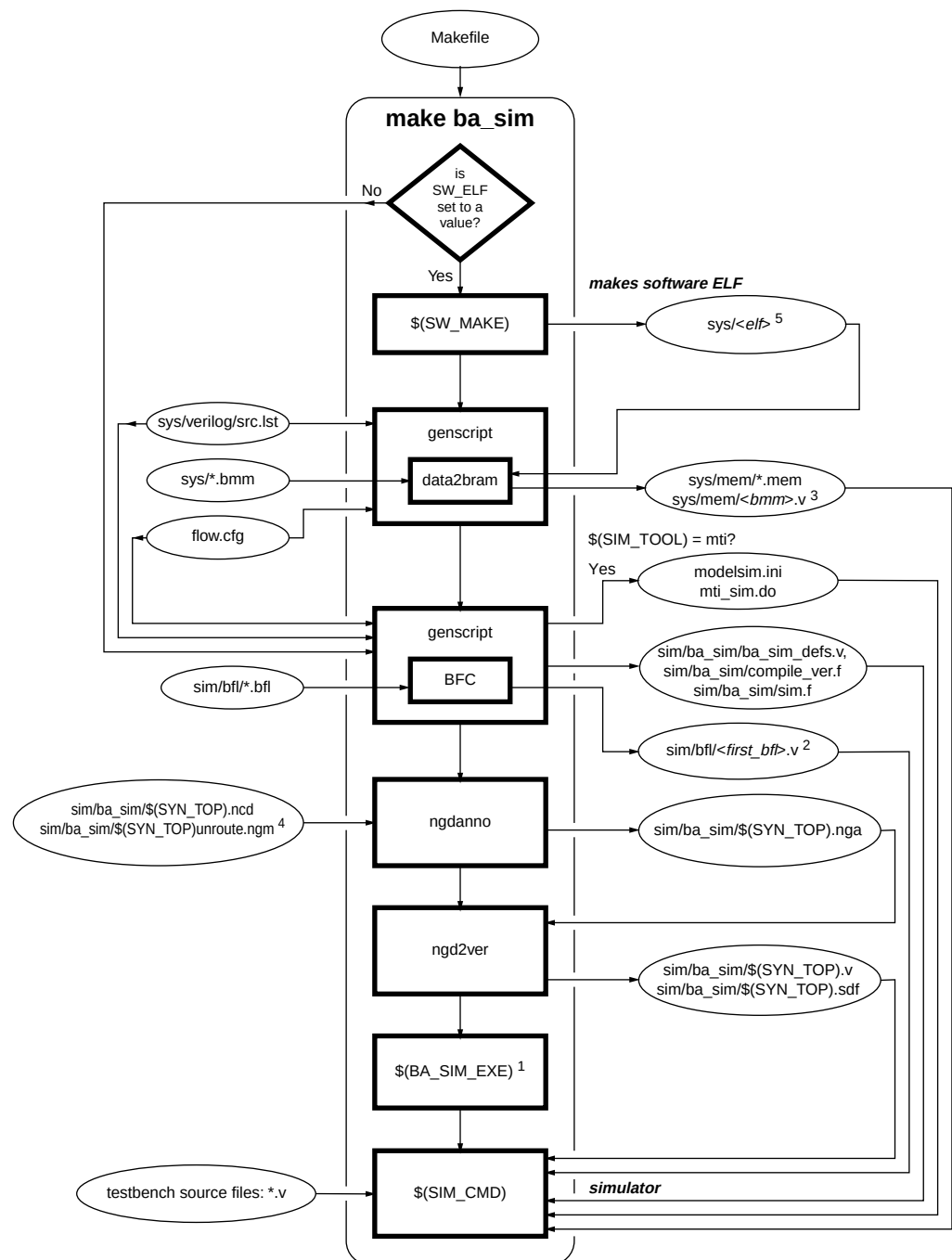
After this preparation, **make** can be invoked with the **fpga** target. The first step of **make fpga** checks whether the **SW\_ELF** variable is set. If it has been assigned a value, **SW\_MAKE** is invoked to generate the **SW\_ELF** file, which is then copied locally to the **sys** directory. Then **data2bram** is invoked using the software application as well as the **.bmm** file specified in **.lst** to generate a **.ucf** file. The **.ucf** file contains block SelectRAM initialization strings for the application.

The second step of '**make fpga**' combines the above **.ucf** file with others specified in the **.lst** file to generate **par/<device>/\$(SYN\_TOP).ucf**. Then, **syn/<device>/\$(SYN\_TOP).edf** and **syn/<device>/\$(SYN\_TOP).ncf** as well as all other **.edf**, **.ngo**, and **.ncf** files are copied to the local **par/<device>** directory.

All of the necessary files are now in the local directory and the implementation tools are run in the following order: **ngdbuild**, **map**, **par**, and **trce**. Many files are generated by running these programs. The main design files are named **\$(SYN\_TOP)\_unroute.ncd** (the output of **map**) and **\$(SYN\_TOP).ncd** (the output of **par**).

Finally, the '**make fpga**' command also copies the resulting **.ngm** and **.ncd** (after **par**) files to the **sim/ba\_sim** directory to assist with the '**make ba\_sim**' process.

# ba\_sim



## Notes:

1. Runs any optional executables prior to invoking simulator (e.g., start\_terminal)
- 2: <first\_bfl> is the base name (without .bfl suffix) of the first .bfl file that is used (Boolean is true)
3. <bmm> is the name of the .bmm file used
4. \$(SYN\_TOP).ncd and \$(SYN\_TOP)\_unroute.ngm are copied from the par/<device> directory during **make fpga**
5. <elf> is the same file name as \$(SW\_ELF)

UG010\_547\_07\_021302

Figure 2-7: Performing Back-Annotated Timing Simulation

Back-annotated simulation is the simulation of the routed FPGA design with timing. The procedure is very similar to that of the **make func\_sim** process with the exception that a netlist of the design is used instead of behavioral HDL source. The **flow.cfg** simulation options should reflect the use of the netlist, and the **.lst** file should not specify only testbench files because all other functionality exists in the netlist. It is unnecessary to include the netlist file because it is automatically included by the design flow.

The **\$(SYN\_TOP).ncd** file and **\$(SYN\_TOP)\_unroute.ngm** file have been previously copied to the **sim/ba\_sim** directory during **make fpga**. These files are run through **ngdanno** and **ngd2ver** to create a Verilog netlist and a SDF file with timing information. These files are automatically included in the simulation.

Another major difference is that with the netlist, the block SelectRAM memory is already initialized, therefore the Verilog output file of **data2bram** that normally accomplishes this task is unnecessary and not included in the simulation.

The remaining steps are identical to that of **func\_sim**; therefore, refer to the description of that section replacing **func\_sim** with **ba\_sim** where appropriate.

## bit

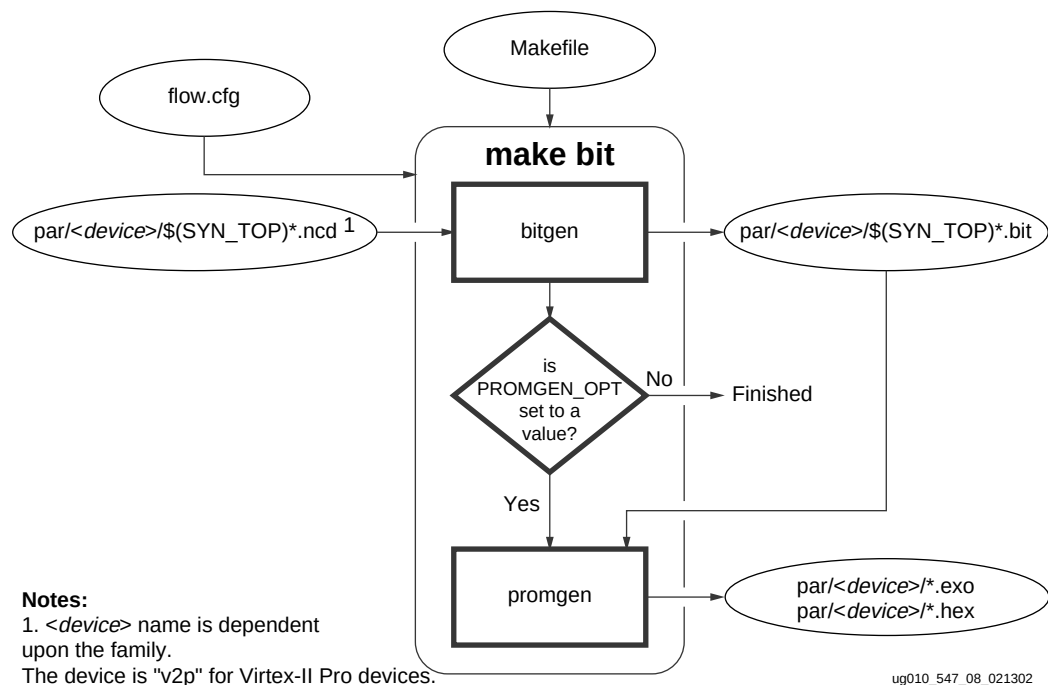


Figure 2-8: Generating Bitstream and PROM Files

Bitstreams and PROM files are created by the **make bit** process. The generation of PROM files is optional and occurs if **PROMGEN\_OPT** in **flow.cfg** is assigned to any value. As always, when generating bitstreams for JTAG download, **-g startupclk:jtagclk** should be added to the command line options for **bitgen** (the variable **BITGEN\_OPT**). When generating PROM files, add **-g startupclk:cclk** should be used (this is the default behavior if the **-g startupclk** option is not specified). The output **.bit**, and **.exo**, **.hex**, ... files are located within the **par/<device>** directory.



## File Syntax

### flow.cfg

Purpose: For the user to supply information regarding tools, design, software application, and tool customization options.

Some of the information supplied:

- HDL - output files are generated in this language
- simulation tool
- simulation executable name
- synthesis tool
- synthesis executable name
- FPGA
- command-line options to use during invocation

The syntax of the configuration file is the same as a **makefile**, with the following restrictions:

- comment line:  
# begins a comment line
- variable definition line:  
LHS = <RHS>
  - LHS (Left Hand Side)
    - can only contain characters A-Z, 0-9, \_
    - must be upper case
    - every LHS in [Table 2-2, page 1176](#) must be present in the configuration file
    - user definitions MUST begin with USER\_ to avoid name collisions with the scripts. Example:
      - USER\_FAST\_DESIGN
  - RHS (Right Hand Side)
    - can be omitted for some variables (see [Table 2-2](#))
    - no white space is allowed in file or directory names
    - '\\' (back slash) characters are converted to '/' (forward slash), therefore, using PC notation for file names is acceptable, but for any other uses, back slashes will not be preserved.
    - should normally be *lower* case except for file and directory path names. This is especially true if it is a valid value listed in [Table 2-2](#) (case much match exactly)
    - previously defined variables and environment variables may be used in the form of \$(VAR) .

There are two potential sources of variable definitions: the *configuration file* and the *environment*. It is preferable that a name be present in only one source. However, if a name does overlap between sources, the priority (highest to lowest) is:

1. Configuration file
2. Environment (only a few environment variables are used, such as **V2PRO**).

Table 2-2: Configuration Variables

| LHS     | RHS      | Valid Values | Description                                    |
|---------|----------|--------------|------------------------------------------------|
| SYN_TOP | Required |              | Name of top-level module for synthesis         |
| SIM_TOP | Required |              | Name of top-level module for simulation        |
| TARGET  | Required |              | Hardware target system standard targets: v2pdk |

Table 2-2: Configuration Variables (Continued)

| LHS                  | RHS      | Valid Values                                                                     | Description                                                                                               |
|----------------------|----------|----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| FAMILY               | Required | virtex2p                                                                         | FPGA family                                                                                               |
| DEVICE               | Required | <i>Any valid Virtex-II Pro device</i><br>example: xc2vp20                        | Virtex-II Pro FPGA device                                                                                 |
| PACKAGE              | Required | <i>see datasheet</i>                                                             | FPGA package                                                                                              |
| SPEEDGRADE           | Required | <i>see datasheet</i>                                                             | FPGA speed grade                                                                                          |
| SW_MAKE              | Optional | <i>&lt;any valid command line&gt;</i><br>example:<br>make -C \$(V2PRO)/.../hello | Command line to run to build the software application used in hardware simulation and FPGA implementation |
| SW_ELF               | Optional |                                                                                  | Name of software application ELF file (including directory path)                                          |
| HDL                  | Required | verilog                                                                          | HDL language to use for simulation and synthesis                                                          |
| SRC_LST              | Required |                                                                                  | Name of LST file (LST file format is described in this document)                                          |
| SIM_TOOL             | Required | mti, vxl                                                                         | Simulation tool                                                                                           |
| SIM_CMD              | Required |                                                                                  | Command name of simulation tool (e.g. for mti: vsim)                                                      |
| FUNC_SIM_VERILOG_OPT | Optional |                                                                                  | Verilog compile options for functional simulation                                                         |
| FUNC_SIM_VHDL_OPT    | Optional |                                                                                  | Reserved                                                                                                  |
| FUNC_SIM_OPT         | Optional |                                                                                  | Command line options for the simulation tool for functional simulation                                    |
| BA_SIM_VERILOG_OPT   | Optional |                                                                                  | Verilog compile options for timing simulation                                                             |
| BA_SIM_VHDL_OPT      | Optional |                                                                                  | Reserved                                                                                                  |
| BA_SIM_OPT           | Optional |                                                                                  | Command line options for the simulation tool for back-annotated simulation                                |
| PPC_MODEL            | Required | swift, bfm                                                                       | Simulation model to use for PPC405                                                                        |
| FUNC_SIM_EXE         | Optional | <i>&lt;any valid command line&gt;</i>                                            | Command line to execute prior to invoking simulation tool for functional simulation                       |
| BA_SIM_EXE           | Optional | <i>&lt;any valid command line&gt;</i>                                            | Command line to execute prior to invoking simulation tool for back-annotated simulation                   |
| SYN_TOOL             | Required | synplify                                                                         | Synthesis tool                                                                                            |
| SYN_CMD              | Required |                                                                                  | Command name for synthesis tool                                                                           |
| SYN_OPT              | Optional |                                                                                  | Command-line options for synthesis tool.<br>Inferred options: <i>&lt;none&gt;</i>                         |
| DATA2BRAM_OPT        | Optional |                                                                                  | Command-line options for data2bram.<br>Inferred options: -bd -bm -o -p                                    |
| NGDBUILD_OPT         | Optional |                                                                                  | ngdbuild command-line options.<br>Inferred option: -p                                                     |

Table 2-2: Configuration Variables (Continued)

| LHS          | RHS      | Valid Values | Description                                              |
|--------------|----------|--------------|----------------------------------------------------------|
| MAP_OPT      | Optional |              | map command-line options.<br>Inferred option: <none>     |
| PAR_OPT      | Optional |              | par command-line options.<br>Inferred option: -w         |
| TRCE_OPT     | Optional |              | trce command-line options.<br>Inferred option: <none>    |
| BITGEN_OPT   | Optional |              | bitgen command-line options.<br>Inferred option: -w      |
| PROMGEN_OPT  | Optional |              | promgen command-line options.<br>Inferred option: <none> |
| NGDANNO_OPT  | Optional |              | ngdanno command-line options.<br>Inferred option: <none> |
| NGD2VER_OPT  | Optional |              | ngd2ver command-line options.<br>Inferred option: -w     |
| NGD2VHDL_OPT | Optional |              | ngd2vhd1 command-line options.<br>Inferred option: -w    |

**Note:** Inferred options are options set by the flow scripts. Do NOT include these options in the value for the \*\_OPT variable

## LST File (src.lst)

This file contains all files used in the design and acts as a single database for all of the design steps: simulation, synthesis, and FPGA implementation. It contains the following files:

- All HDL files (.v)
  - recommend listing in bottom-up order for any order sensitive tools
- IP core files (.edf, .ngo, .ncf)
- Data2BRAM files (.bmm)
- User constraint files (.ucf)
- Bus-functional language files (.bfl)
- MTI DO file (.do)
- ModelTech

Syntax:

```
<=file type ID> filename <boolean> ;
```

Examples (each entry must be on only *one* physical line; line extensions are not supported):

```
$V2PRO/source/sim_models/other/sram_mem/sram256k_16.v ((MAKE = func_sim) OR (MAKE = ba_sim));
$CORECONNECT/tk/PLB_TOOLKIT4X/verilog/plb_master_comp.v ((MAKE = func_sim) OR MAKE =
ba_sim)) and (PPC_MODEL = bfm));
../..sim/func_sim/wave.do (MAKE = func_sim);
=VERILOG_INCLUDE_DIR $CORECONNECT/tk/PLB_TOOLKIT4X/verilog ((MAKE = func_sim) OR (MAKE = ba_sim));
```

The fields in '<>' (brackets) are optional fields

### File Type ID

The optional *<=file type ID>* field specifies file type. File types are normally based upon the file name suffix, but there are a few cases where there is no suffix. The user can also override the file type with this method if the file has an unusual suffix.

Table 2-3: File Types

| File Type ID        | File Suffix     | Description                                                                                              |
|---------------------|-----------------|----------------------------------------------------------------------------------------------------------|
| BMM                 | .bmm            | Xilinx <b>data2bram</b> BMM file                                                                         |
| VERILOG             | .v              | Verilog source file                                                                                      |
| VERILOG_INCLUDE_DIR |                 | Verilog include directory                                                                                |
| VERILOG_LIB_DIR     |                 | Verilog library directory (user must still use command-line option +libext+ to indicate the file suffix) |
| BFL                 | .bfl            | CoreConnect Bus-Functional Language file                                                                 |
| MTI_DO              | .do             | ModelTech MTI DO file                                                                                    |
| LST                 | .lst            | Another LST file (nesting is allowed)                                                                    |
| NGO                 | .ngo            | Xilinx NGO file                                                                                          |
| EDIF                | .edf .edn .edif | EDIF file                                                                                                |
| NCF                 | .ncf            | Xilinx NCF file                                                                                          |
| UCF                 | .ucf            | Xilinx UCF file                                                                                          |

## Filename

The **filename** field contains the name of the file. The user may use absolute paths or relative paths (with respect to the location of the including LST file). Use of environment variables (e.g. \$V2PRO) is acceptable and encouraged for portability.

## Boolean

The optional **<Boolean>** field describes when the file is used. It uses variables defined in the configuration file to evaluate expressions. Additionally, another variable **MAKE** is defined that specifies which make target is currently being executed (for **make func\_sim**, MAKE = func\_sim). Expressions are composed of the following operators:

Table 2-4: Boolean Operators

| Operator | Description       |
|----------|-------------------|
| =        | is equal to       |
| !=       | is not equal to   |
| AND      | logical AND       |
| OR       | logical OR        |
| NOT      | unary logical NOT |

Boolean expressions have the following rules:

- The entire Boolean expression should be surrounded by a set of parentheses '()'
- Each subexpression should also be surrounded by a set of parenthesis '()' to control the order of evaluation.

Boolean expression examples:

- (MAKE = synth)
- ((MAKE = func\_sim) OR (MAKE = ba\_sim))
- (MAKE != ba\_sim)
- (NOT ((MAKE = func\_sim) OR (MAKE = ba\_sim)))

Shorthand Notation: When making a specific target, only certain file types are used; the rest are ignored and therefore the Boolean expressions can be simplified. For example, EDIF files are ignored when making all targets except **fpga**. Therefore, it is not necessary to add (MAKE = fpga) to the Boolean expression. The following table lists for each file type the targets where that particular file could be used.

Table 2-5: File Type versus Make Target for Shorthand Notation

| File Type ID        | Make Target that Uses File        |
|---------------------|-----------------------------------|
| BMM                 | ba_sim, func_sim, fpga, sw        |
| VERILOG             | ba_sim, func_sim, synth           |
| VERILOG_INCLUDE_DIR | ba_sim, func_sim, synth           |
| VERILOG_LIB_DIR     | ba_sim, func_sim                  |
| BFL                 | ba_sim, func_sim                  |
| MTI_DO              | ba_sim, func_sim                  |
| LST                 | ba_sim, func_sim, fpga, synth, sw |
| NGO                 | fpga                              |

Table 2-5: File Type versus Make Target for Shorthand Notation (*Continued*)

| File Type ID | Make Target that Uses File |
|--------------|----------------------------|
| EDIF         | fpga                       |
| NCF          | fpga                       |
| UCF          | fpga                       |

## Tool-Specific Files

### Synthesis Options File

#### `syn/${SYN_TOOL}.opt`

Certain synthesis tools allow a great deal of options that cannot be passed merely on the command line. The `syn/${SYN_TOOL}.opt` file facilitates such usage. The options contained within this file will be added to the synthesis batch script that is created.

#### Synplify - `syn/synplify.opt`

The following information is provided automatically by the design flow and should NOT be included in the `synplify.opt` file:

```
project -result_file
set_option -top_module
set_option -technology
set_option -part
set_option -package
set_option -speed_grade
```

## Special Notes

Read this section for special notes under various conditions. It is important to read the areas that apply to the system in question.

## HDL = verilog

### Automatic ``define` Statements

To assist in building testbenches and source code, some definitions in the configuration file and the special MAKE target variable will translate into ``define` statements that are passed along to the Verilog simulator and the synthesis tool. These ``define` statements can be used for conditional code within Verilog source files.

Specifically, any variable in the configuration file that meets the following requirements will be converted:

- RHS has no white space
- RHS uses only a-z, A-Z, 0-9, \_

For LHS = RHS, there would be a ``define LHS_RHS`.

Case is maintained in the ``define` statement (remember that Verilog is case-sensitive).

For example, when running `make synth` and having the following entries in the configuration file:

```
FAMILY = virtex2p
USER_TEST = 1
```

The following definitions are produced:

```
`define MAKE_synth
`define FAMILY_virtex2p
`define USER_TEST_1
```

## Example Files

### flow.cfg

A sample **flow.cfg** based upon the embedded platform is described in detail below.

#### Design Name

```
#Design Name
SYN_TOP = top # top-level name for synthesis
SIM_TOP = testbench # top-level name for simulation
```

The top of the design hierarchy must be identified to run synthesis and simulation tools. Typically, there is a different top level for simulation versus synthesis due to the addition of a testbench.

#### Target System

```
#Target System
TARGET = v2pdk # ml1 / ml2 / v2pdk
```

This variable allows some differentiation if you have multiple target systems. For instance, if there is little difference between two boards, the variation can be handled by specifying a different value for the **TARGET**. By using **TARGET** in a Boolean expression of a **.lst** file entry, different source files can be included based upon the value.

#### FPGA Device

```
#FPGA Device
FAMILY = virtex2p # virtexe / virtex2 / virtex2p
DEVICE = xc2vp20 # e.g. xc2v6000
PACKAGE = ff896 # e.g. fg1156
SPEEDGRADE = -7 # e.g. -8
```

The FPGA device to be used must be specified. For Virtex-II Pro designs, the value should be set to **virtex2p**. Currently, synthesis tools may not handle a Virtex-II Pro device, but the flow will automatically adjust to synthesize as a Virtex-II device without any user intervention.

#### Software Application

```
#Software Application - Hello UART
SW_MAKE = make -C $(V2PRO)/source/sw/apps/hello_uart hello_debug_v2pdk
SW_ELF = $(V2PRO)/source/sw/apps/hello_uart/v2pdk/debug/hello.elf
```

These variables specify the software application. The first is the command to build the software application which should be the value given to **SW\_ELF**. It is assumed that the first does generate the second – the user is responsible for this being true. Note the use of the environment variable **V2PRO**; this is the syntax to use environment variables and previously defined variables in this file.

#### HDL Language

```
#HDL Language
HDL = verilog # verilog
```

The hardware description language (HDL) must be specified. Currently, only Verilog is supported; therefore, this is for future features.

```
SRC_LST = sys/$(HDL)/src.lst # source LST file
```

The top-level LST file needs to be specified because there is no assumption on what file name is being used. Typically, this would be set to **sys/verilog/src.lst**.

## Simulation

```
#Simulation
SIM_TOOL = mti # vxl / mti
SIM_CMD = vsim #
```

The simulation tool must be specified to use either ModelSim or Verilog-XL. Additionally, the simulation executable must also be given.

```
FUNC_SIM_VERILOG_OPT = +libext+.v # inferred: <none>
FUNC_SIM_VHDL_OPT = # inferred: <none>
FUNC_SIM_OPT = # inferred: <none>
BA_SIM_VERILOG_OPT = +libext+.v # inferred: <none>
BA_SIM_VHDL_OPT = # inferred: <none>
BA_SIM_OPT = +delay_mode_path +no_notifier +transport_path_delays
+transport_int_delays # inferred: <none>
```

The VHDL options should be left blank as their function is not currently implemented. The \*\_SIM\_VERILOG\_OPT variables refer to the command-line options used in the compilation phase of a compiled Verilog simulator (such as ModelSim). For example, if using ModelSim, **FUNC\_SIM\_VERILOG\_OPT** specifies the options for **vlog** when making the **func\_sim** target. The \*\_SIM\_OPT variables are options used in the simulation phase of a compiled Verilog simulator. Refer to the simulation tool's documentation for information on what command-line options are allowed in each case. If an interpretive simulator is used (such as Verilog-XL), there is only one command-line. In this case, both \*\_SIM\_VERILOG\_OPT options and \*\_SIM\_OPT options are used as command-line arguments for the simulator.

```
PPC_MODEL = swift # swift / bfm
#PPC_MODEL = bfm # swift / bfm
```

The simulation model for the PPC405 must be chosen. The user may choose either the SWIFT model or the Bus Functional Model (BFM). When using the SWIFT model, the simulator must have the capability of simulating SWIFT models.

## Script/Programs to Execute with Simulation

```
Script/Programs to Execute with Simulation
FUNC_SIM_EXE = start_terminals
BA_SIM_EXE = start_terminals
```

As a convenience, if a user specifies a program with these variables, it is invoked just prior to simulation. The kit uses this functionality in the reference platforms to bring up terminals that are used with the simulation. Only a single command may be specified, the user should create a script (UNIX) or batch file (DOS) if more than one activity is desired.

## Synthesis

```
#Synthesis
SYN_TOOL = synplify # synplify
SYN_CMD = synplify
SYN_OPT = # inferred (synplify): <none>
```

Similar to simulation, the type of synthesis tool must be specified along with the name of the executable. Only Synplify is supported at this time. To run Synplify without the GUI (batch mode), add '-batch' to SYN\_OPT.



## FPGA Implementation

```
#FPGA Implementation
DATA2BRAM_OPT = # inferred: -bd -bm -o -p
NGDBUILD_OPT = # inferred: -p
MAP_OPT = -pr b # inferred: <none>
PAR_OPT = -ol 5 -xe 2 # inferred: -w
TRCE_OPT = -v 10 -skew # inferred: <none>
BITGEN_OPT = -g startupclk:jtagclk # inferred: -w
PROMGEN_OPT = # inferred: <none>
NGDANNO_OPT = # inferred: <none>
NGD2VER_OPT = -ne # inferred: -w
NGD2VHDL_OPT = # inferred: -w
BITGEN_OPT = -g startupclk:cclk -g userid:0x10000006
PROMGEN_OPT = -w -p exo -x xc18v04 xc18v04 xc18v04 -u 0
```

Each of the Xilinx FPGA implementation tools have options that affect the program's behavior. These variables allow the user to specify these options. Note that some options are automatically specified, these should NOT be included in value for the \*\_OPT variable. See [Table 2-2, page 1176](#) for a full list of inferred options. To disable PROM file generation, leave the value of PROMGEN\_OPT blank.

## User Variables

```
User defined definitions
IMPORTANT: Begin user definitions with USER_ to avoid collisions now
and in the future.

Uses for USER variables
1. use in boolean equations of .lst files to further differentiate
when files are used.
USER_1 = 1
```

The user may create their own variables. The prefix USER\_ should begin any user-defined variable to avoid collisions with variables used in the flow scripts. These variables can be used in the Boolean expressions within the .lst file and also can be used in the testbench to control behavior.

## src.lst

A subsection of a **src.lst** file is explained below to demonstrate the syntax and capabilities of the design flow.

```
=VERILOG_INCLUDE_DIR $CORECONNECT/tk/OPB_TOOLKIT2X/verilog ((MAKE =
func_sim) OR (MAKE = ba_sim));
```

The first term specifies that this is a Verilog include directory which is searched when resolving include files in the Verilog source files. The second term is the name of the directory. Notice that the environment variable '\$CORECONNECT' is used – environment variables are resolved during processing. The third term is a Boolean expression that indicates the conditions that must be true for this directory to be used. In this case, the directory is used only during simulation (MAKE is automatically assigned to the current **make** target). It is important that there are sufficient parentheses to unambiguously define the order of evaluation.

It is important to note that entries are processed in the order listed in the **src.lst** file. Order is preserved when passing the list of files and directories to the simulation and synthesis executables.

```
$SYNPLICITY/lib/xilinx/virtex2p.v ((MAKE = synth) AND ((FAMILY =
virtex2p));
```

This entry differs from the previous one because the file type is not explicitly stated. It is assumed from the file extension that this is a Verilog source file. The entry is the

SynplifyVirtex-II Pro primitive library and is consequently only used during synthesis of a Virtex-II Pro device.

```
=VERILOG_LIB_DIR $XILINX/verilog/src/unisims ((MAKE = func_sim) OR
(MAKE = ba_sim));
```

A Verilog library directory corresponds with the **-y** options of most Verilog simulators. Unresolved modules are searched for within these directories based upon the name of the module. The extension name for library modules must be specified in the simulator options (+libext+ option). Typically, a **.v** extension name is used for library files, in which case, the user adds **+libext+.v** to **FUNC\_SIM\_VERILOG\_OPT** in **flow.cfg**.

```
$V2PRO/source/sim_models/Xilinx/proc_block_BFM/proc_block_BFM.v
((MAKE = func_sim) and (PPC_MODEL = bfm));
```

In this case, the **PPC\_MODEL** variable of **flow.cfg** is used as well as the standard **MAKE**. This file is the BFM model of the PPC405 and is, therefore, only used in BFM mode.

```
$V2PRO/source/hw/verilog/plb_arbiter/src.lst ((MAKE = func_sim) OR
(MAKE = synth));
```

This is a typical hardware IP entry. The type is another **.lst** file allowing for modular design. This secondary list file will be expanded if the Boolean condition is met. Since the IP will be included within the FPGA, the source files are only used during functional simulation and synthesis (during back-annotated simulation, the netlist will be used).

```
global_params.v ((MAKE = func_sim) OR (MAKE = synth));
```

Relative file name paths are assumed to be relative to the **src.lst** that contains the entry. For instance, if **/home/examples/src.lst**, included the above line, it would refer to **/home/examples/global\_params.v**. This is true regardless of any **src.lst** nesting. For example, if **/home/work/src.lst** included **/home/examples/src.lst** as an entry, the above line would still refer to **/home/examples/global\_params.v**.

```
../../../../sim/bfl/test_emb_ref_des.bfl ((PPC_MODEL = bfm) AND (MAKE
= func_sim) AND (TARGET != ml1));
```

BFL files need to be specified within **src.lst**. Multiple BFL files can be specified.

```
../../../../par/top.ucf (TARGET != ml1);
```

For this file, the **MAKE** is omitted from the Boolean expression – using the shorthand notation. It is redundant because UCF files will only be used during **make fpga**.

```
../memory_init.bmm (TARGET != ml1);
```

The **.bmm** file used to convert ELF files to simulation and the UCF file must be specified. Again, it is unnecessary to specify the **make** target via **MAKE**.

For complete **src.lst** examples, examine the reference platforms.

## Requirements

The flow uses **GNU** and **PERL** (version 5), also shipped. **Table 2-6** shows other requirements:

**Table 2-6: Design Flow Tool Support**

| Category        | Supported by Flow    |
|-----------------|----------------------|
| Simulation Tool | ModelSim, Verilog-XL |
| Synthesis Tool  | Synplify             |
| FPGA Family     | Virtex-II Pro        |

The following directory structures are used with the design flow:

- ```

-----platforms
|   \-----<name of system>
|       \-----par
|           \-----v2p
|       \-----sim
|           \-----ba_sim
|           \-----bfl
|           \-----func_sim
|           \-----testbench
|               \-----verilog
|               \-----vhdl
|   \-----syn
|       \-----v2p
|   \-----sys
|       \-----mem
|       \-----verilog
|       \-----vhdl

```

- ```

-----apps
|
| \-----hello_uart
| |
| | \-----v2pdk
| | |
| | | \-----debug
| | | \-----release
|
| \-----libs
| |
| | \-----bsp
| | |
| | | \-----build
| | | |
| | | | \-----v2pdk
| | | | |
| | | | | \-----debug
| | | | | \-----release
| | | |
| | | \-----v2pdk
| | | |
| | | | \-----debug
| | | | \-----release
| | |
| | | \-----include
|
| \-----mapfiles

```

- Note:** \* - indicates a file generated by the design flow

```

\-----platforms
|
| \-----simple
| |
| | Makefile
| | flow.cfg
| | make_ba_sim.log*
| | make_bit.log*
| | make_fpga.log*
| | make_func_sim.log*
| | make_synth.log*
| | make_sw.log*
| | \-----par
| | |
| | | top.ucf
| | | \-----v2p
| | | |
| | | | top.bit*
| | | | top.edf*
| | | | top.ngd*
| | | | top_unroute.pcf*

```

```

top_unroute.ncd*
top_unroute.ngm*
top.ncd*
top.twr*
top.ucf*
`-----sim
`-----ba_sim
| ba_sim_defs.v*
| compile_ver.f*
| compile_vhdl.f*
| modelsim.ini*
| mti_sim.do*
| sim.f*
| top.ncd*
| top.nga*
| top.sdf*
| top.v*
| top_unroute.ngm*
| wave.do
| `-----work
`-----bfl
| test_simple_ref_des.bfl
| test_simple_ref_des.v*
`-----func_sim
| compile_ver.f*
| compile_vhdl.f*
| func_sim_defs.v*
| modelsim.ini*
| mti_sim.do*
| sim.f*
| wave.do
| `-----work
`-----testbench
| `-----verilog
| | sim_params.v
| | testbench.v
| | view_reg.v
| `-----vhdl
`-----syn
| synplify.opt
| virtex2p.v
| `-----v2p
| | synth_defs.v*
| | top.edf*
| | top.prj*
`-----sys
| bram_init.bmm
| hello.elf*
| `-----mem
| | bram_init.v*
| | bram_init.ucf*
| `-----verilog
| | bram_block.v
| | clk_rst_startup.v
| | global_params.v
| | ip_wrapper.v
| | plb_bus_logic.v
| | src.lst
| | top.v
| `-----vhdl

```



## PPC405 Bus Functional Model Usage

### Overview

The PPC405 Bus Functional Model (BFM) is a convenient tool that can be used to generate specific bus transactions in whatever sequence is desired. It drops in to replace the normal PPC405 model and is connected so that coordinated transactions across the three buses (**DCR**, **ICU**, and **DCU**) can be accomplished with ease.

This document describes the usage of the PPC405 BFM included at

**\$V2PRO0/source/sim\_models/Xilinx/proc\_block\_BFM/proc\_block\_BFM.v.**

The PPC405 BFM allows independent control of the DCR, DCU, and ICU ports. The user can specify individual bus cycles for each of the three interfaces which is very useful for testing the functionality of slaves.

**Note:** All other PPC405 inputs are ignored and all other outputs are set to 0; the user should not use any other ports besides DCR, DCU, and ICU.

This document describes issues specific to the PPC405 BFM and does not cover general bus functional model information in any great detail. Specific details about the CoreConnect™ bus functional models, bus functional compilers, and bus functional language can be found in:

- Processor Local Bus Functional Model Toolkit
- Device Control Register Bus Functional Model Toolkit

### Description

Structurally, the PPC405 BFM appears as shown in **Figure 3-1**:

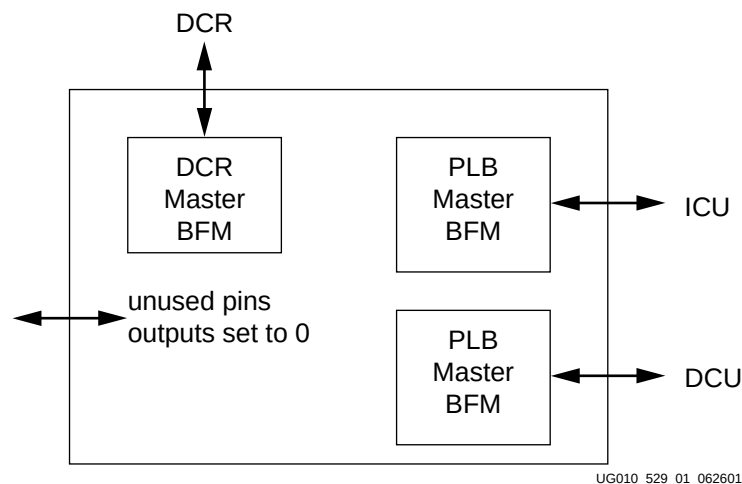


Figure 3-1: Structure of PPC405 BFM

The PPC405 BFM uses the same port interface as the PPC405 SWIFT model, and contains two PLB master BFMs and one DCR master BFM. Since it has the same pinout and top-level module name, the PPC405 BFM can be swapped interchangeably with the PPC405 SWIFT model, although only the three bus ports will be active in the PPC405 BFM.

Each BFM has two 32-bit buses (**synch\_in[0:31]** and **synch\_out[0:31]**) to coordinate commands with other BFMs and potentially external events. For convenience, the buses for all three BFMs have been combined within the PPC405 BFM.

The **synch\_in[0:31]** input bus is a common bus to all three BFMs. The user should assign this wire in the testbench to the OR of the **synch\_out** signals of every BFM. The testbench can reach this bus by **hier\_cpu\_name.synch\_in[0:31]** where **hier\_cpu\_name** is replaced with the hierarchical path of the PPC405 BFM instance.

The **synch\_out[0:31]** buses of all three underlying BFMs are OR'ed together into a single **synch\_out[0:31]** bus. The testbench can reach this bus by **hier\_cpu\_name.synch\_out[0:31]** where **hier\_cpu\_name** is replaced with the hierarchical path of the PPC405 BFM instance.

As an example, a common testbench with bus monitors would include the following statements:

```
assign synch_in = plb_monitor_synch_out
 | dcr_monitor0_synch_out
 | opb_monitor_synch_out
 | testbench.top.PPC405.synch_out;
// connect synch bus to PPC405 BFM if it is used
assign testbench.top.PPC405.synch_in = synch_in;
```

In this example, there are three bus monitors that communicate with the PPC405 BFM. The three **synch\_out** buses have been logically OR'd with the **synch\_out** from the PPC405 BFM to form a shared **synch\_in**. All of the BFMs use this **synch\_in** bus.

Also for convenience, the error signals of each BFM have also been wired together into the single signal **hier\_cpu\_name.bfm\_error\_detect**. The testbench can monitor whether any errors have occurred.

## Bus Restrictions

Although the underlying BFMs are generic, the PPC405 will only generate a small subset of possible transactions. Only these transactions should be used with the PPC405 BFM model; unknown results will occur if other transactions are performed. The allowed transactions are:

- ICU: 4-word or 8-word reads  
**Note:** The PPC405 ICU and DCU can each have a maximum of only two concurrent read transactions. The user should adjust the testbench or the BFL to prevent a greater number of reads from occurring.
- DCU: 1-word (with byte enables) or 8-word reads/writes<sup>1</sup>
- DCR: no restrictions

## BFL

The sequence of bus cycles for the three BFMs is described within a single Bus Functional Language (BFL) file. Sample BFL files can be found in the design examples of this release. The syntax and description of a BFL file can be found at:

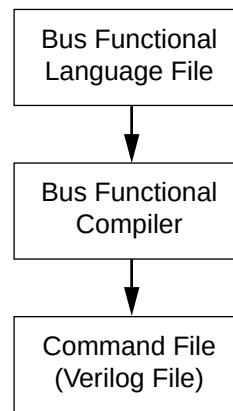
- Processor Local Bus Functional Model Toolkit  
**\$SCORECONNECT/published/corecon/PlbToolkit.pdf**
- Device Control Register Bus Functional Model Toolkit  
**\$SCORECONNECT/published/corecon/DcrToolkit.pdf**

Within the PPC405 BFM, the three BFM's are named: **M\_ICU**, **M\_DCU**, and **M\_DCR**. Therefore, use these names for the device name in your BFL file:

- **hier\_cpu\_name/M\_ICU**
- **hier\_cpu\_name/M\_DCU**
- **hier\_cpu\_name/M\_DCR**

**Note:** Replace **hier\_cpu\_name** with the hierarchical name of the PPC405 instance (using '/' as a hierarchy separator).

Once the BFL file is completed, it must be compiled to create a Verilog file to be included in the simulation (Figure 3-2). Use the bus functional compiler provided in the release.



UG010529\_02\_062501

Figure 3-2: **Compilation Procedure**

The bus functional compiler must be invoked in the directory where the BFL file resides:

```
$ run_BFC bflname.bfl
```

The output command file will be named **bfl\_test.v** and declare a module **bfl\_test**. Include **bfl\_test** as a top-level module in the simulation.

**Note:** Do not name your file **bfl\_test.bfl** as this file is overwritten by the run\_BFC script.





## SmartModel<sup>®</sup> Usage

---

### Overview

The Xilinx Virtex-II Pro<sup>™</sup> family of devices offers the designer an unsurpassed array of features in the form of the IBM<sup>®</sup> PowerPC<sup>®</sup> microprocessor and the Rocket I/O<sup>™</sup> transceiver. These complex and proprietary pieces of logic require Synopsys<sup>®</sup> SWIFT<sup>™</sup> interface SmartModel<sup>®</sup> Library models for simulation. This chapter details usage of the SmartModel simulation model library. Proper installation of the models and configuration of the simulation environment are required prior to use. Open the **install.htm** file on the Virtex-II Pro<sup>™</sup> Developer's Kit (March 2002) CD-ROM for more information.

### Description

SmartModels are simulator-independent simulation models that are derived from the actual design and are, therefore, accurate evaluation models. The models allow for accurate results with good simulation performance while still protecting valuable intellectual property. Simulating with SmartModel libraries requires a simulator that supports the SWIFT interface.

More information on Synopsys SmartModel libraries can be obtained at:

<http://www.synopsys.com/products/lm/doc/smartmodel.html>

[http://www.synopsys.com/products/lm/ip/vmc\\_ds.html](http://www.synopsys.com/products/lm/ip/vmc_ds.html)

### Available SmartModels

The SmartModels available in the Developer's Kit are required to simulate the PPC405 CPU and the Rocket I/O transceiver. **Table 4-1** shows the SWIFT-interface SmartModel model and the corresponding Xilinx library element that it supports.

**Table 4-1: Models for Simulating Xilinx Library Elements**

| SmartModel Simulation Model | Xilinx Library Element | Description                                     |
|-----------------------------|------------------------|-------------------------------------------------|
| PPC405_SWIFT                | PPC405                 | SmartModel model for the PowerPC 405 CPU        |
| GT_SWIFT                    | Rocket I/O primitives  | SmartModel model for the Rocket I/O transceiver |

## Using Models in a Design

Library elements that have SmartModel simulation models are instantiated in the same way that other Xilinx library elements are instantiated. For simulation, the **unisim** and **simplim** library descriptions use wrappers that call the SmartModel elements. It is necessary to include these HDL wrappers in the simulation. Additionally, by adding the programming language interface (PLI) tasks that are available in SWIFT-enabled simulators to the testbench, it is possible to have visibility into the PPC405 model. The Rocket I/O transceiver (GT\_SWIFT model) has no internal visibility.

### Verilog LMTV PLI Tasks

Logic Modeling to Verilog (LMTV) is a PLI application that interacts with the SmartModel simulation model and defines a number of tasks that can be included within the Verilog testbench. Some useful PLI tasks that work with the Developer's Kit models are described below:

#### \$lm\_monitor\_enable

This PLI task enables one or more window elements for access. Windows allow you to view and possibly change a model's internal registers. To enable all windows of model, specify only the instance name without any window names.

Usage:

```
$lm_monitor_enable(<instance_name>, <optional_window_name1>,
<optional_window_name2>)
```

#### \$lm\_status

This PLI task gives the status for the model and all of its windows in a model instance. This is useful to see all the possible windows that exist.

Usage:

```
$lm_status(<instance_name>)
```

#### \$lm\_help

This PLI task displays the usage for all LMTV PLI tasks

Usage:

```
$lm_help
```

For more information on other commands see:

<http://www.synopsys.com/products/lm/doc/smartmodel.html>

### PPC405 Model

For the PPC405, several HDL wrappers built around the SmartModel simulation model make it necessary to append additional hierarchy to the hierarchical name of the PPC405 instance when using an LMTV command. The amount of additional hierarchy is dependent upon the simulator being used. For ModelSim and Verilog-XL, append '.IPPC405\_SWIFT.I1' to the instance name. To enable all windows of the PPC405, use the following command:

```
$lm_monitor_enable(<hier_path_to_PPC405>.IPPC405_SWIFT.I1);
```

After enabling, to access a specific window, append to the instance name:

```
'.<window_name>'
```

For example, to access the PPC405 LR register:

```
wire LR = <hier_path_to_PPC405>.IPPC405_SWIFT.I1.LR;
```

The following is a list of available windows for the Xilinx PPC405 SmartModel simulation model. The windows are equivalent to the PPC405 internal registers.

**Table 4-2: PPC405 SmartModel Register Windows**

| <b>Name</b>    | <b>Access</b> | <b>Width</b> |
|----------------|---------------|--------------|
| dcdAddr        | Read Only     | 32 bits      |
| dcdData        | Read Only     | 32 bits      |
| exeFull        | Read Only     | 1 bit        |
| exeAddr        | Read Only     | 32 bits      |
| exeAReg        | Read Only     | 32 bits      |
| exeBReg        | Read Only     | 32 bits      |
| exeResult      | Read Only     | 32 bits      |
| GPR0 ... GPR31 | Read Only     | 32 bits      |
| CR             | Read Only     | 32 bits      |
| CTR            | Read Only     | 32 bits      |
| DAC1           | Read Only     | 32 bits      |
| DAC2           | Read Only     | 32 bits      |
| DVC1           | Read Only     | 32 bits      |
| DVC2           | Read Only     | 32 bits      |
| XER            | Read Only     | 32 bits      |
| DBCR0          | Read Only     | 32 bits      |
| DBCR1          | Read Only     | 32 bits      |
| DBSR           | Read Only     | 32 bits      |
| DBDR           | Read Only     | 32 bits      |
| DCCR           | Read Only     | 32 bits      |
| DEAR           | Read Only     | 32 bits      |
| DCWR           | Read Only     | 32 bits      |
| ICCR           | Read Only     | 32 bits      |
| SLER           | Read Only     | 32 bits      |
| SU0R           | Read Only     | 32 bits      |
| SGR            | Read Only     | 32 bits      |
| ZPR            | Read Only     | 32 bits      |
| PID            | Read Only     | 32 bits      |
| CCR0           | Read Only     | 32 bits      |
| ICDBDR         | Read Only     | 32 bits      |
| IAC1           | Read Only     | 32 bits      |
| IAC2           | Read Only     | 32 bits      |

Table 4-2: PPC405 SmartModel Register Windows (Continued)

| Name  | Access    | Width   |
|-------|-----------|---------|
| IAC3  | Read Only | 32 bits |
| IAC4  | Read Only | 32 bits |
| LR    | Read Only | 32 bits |
| EVPR  | Read Only | 32 bits |
| SPRG0 | Read Only | 32 bits |
| SPRG1 | Read Only | 32 bits |
| SPRG2 | Read Only | 32 bits |
| SPRG3 | Read Only | 32 bits |
| SPRG4 | Read Only | 32 bits |
| SPRG5 | Read Only | 32 bits |
| SPRG6 | Read Only | 32 bits |
| SPRG7 | Read Only | 32 bits |
| SRR0  | Read Only | 32 bits |
| SRR1  | Read Only | 32 bits |
| SRR2  | Read Only | 32 bits |
| SRR3  | Read Only | 32 bits |
| MSR   | Read Only | 32 bits |
| PVR   | Read Only | 32 bits |
| ESR   | Read Only | 32 bits |
| PIT   | Read Only | 32 bits |
| TBH   | Read Only | 32 bits |
| TBL   | Read Only | 32 bits |
| TSR   | Read Only | 32 bits |
| TCR   | Read Only | 32 bits |

The reference platforms in the PPC405 Developer's Kit contain several files that can be modified to display the state of the registers shown in Table 4-2. Files specific to the *PPC405 Embedded Reference System* are described in Table 4-3.

Table 4-3: Files in the \$V2PRO/platforms/embedded/sim/testbench/verilog/ Directory

| File         | Description                                                                                                 |
|--------------|-------------------------------------------------------------------------------------------------------------|
| testbench.v  | This file controls the window-enabling                                                                      |
| view_reg.v   | This file accesses the various registers and displays changes to them during simulation                     |
| sim_params.v | This file has a wire named <b>reg_MsgOn</b> that can be set to turn on or off the display of these messages |

## Rocket I/O Transceiver Model

For the Rocket I/O transceiver, there are several wrappers around the SmartModel simulation model making it necessary to append additional hierarchy to the hierarchical name of the primitive instance when using a LMTV command. The amount of additional hierarchy is also dependent upon the simulator being used. For ModelSim and Verilog-XL, append '.gt1.gt\_swift\_1.I1' to the instance name.

## Invoking the Simulator

### Include Wrapper Files

The wrapper files located in \$LMC\_HOME/../../wrappers/<simulator> declare modules that are instantiated by **unisim** or **simplim** library element. These files *must* be included in the simulation to use the models.

For ModelSim, replace <simulator> with:

```
mtiverilog
```

For Verilog-XL, replace <simulator> with:

```
verilogxl
```

For example:

```
% verilog -y $LMC_HOME/../../wrappers/verilogxl...
```

### Include PLI for Verilog-XL Only

For Verilog-XL, if the SWIFT PLI has not already been compiled into the binary Verilog executable, you must include the following option on the Verilog-XL command line:

```
+loadpli1=swiftpli:swift_boot
```

For example:

```
% verilog +loadpli1=swiftpli=swift_boot -y $LMC_HOME/../../wrappers/verilogxl...
```

## Further Reading

For further information on SmartModel simulation models, see the Synopsys SmartModel Library Documentation at: <http://www.synopsys.com/products/lm/doc/smartmodel.html>



# **Volume 5: Software Development Tools**

## ***Virtex-II Pro™ Platform FPGA Developer's Kit***

March 2002 Release







The Xilinx logo shown above is a registered trademark of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

"Xilinx" and the Xilinx logo are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, XC5210 are registered Trademarks of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Bencher, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Bencher, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, Nano-Blaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, Rocket I/O, SelectI/O, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM      IBM Logo      PowerPC      PowerPC Logo      Blue Logic      CoreConnect      CodePack

All other trademarks are the property of their respective owners.

Xilinx does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 2002 Xilinx, Inc. All Rights Reserved.

# *Introduction to Software Development Tools*

---

**TBD**



# *Using Data2BRAM*

---

## **Abstract**

Virtex-II Pro™ devices combine PowerPC® CPUs and FPGA fabric into one integrated circuit. In the past, hardware and software development efforts were fairly isolated from one another. With the Virtex-II Pro device, the CPU can now execute code in FPGA Block RAM, or from external RAM/ROM. This presents new challenges to product development efforts. This chapter discusses how the Data2BRAM tool simplifies the process of hardware and software development.

## **Introduction**

Data2BRAM is a very versatile and powerful tool for hardware and software development. It provides many differing features which are targeted to solve specific problems. The tool is designed to provide a bridge between the hardware and software worlds.

Hardware engineers must simulate complex systems. These complex systems have memory that is used by the hardware to accomplish a specific task. The memory might be a FIFO storage element, a DSP coefficient table, or perhaps code/data storage for a CPU. Modern FPGAs such as the Virtex, Virtex-II and Virtex-II Pro families offer substantial amounts of on chip Block RAM, or BRAM. The FPGA hardware engineer must choose where to put the memory of the system -- inside or outside the FPGA. For example, a Virtex-II Pro designer might choose to use BRAMs to store some PowerPC code, build FIFOs and store DSP values. Other PowerPC code and data would then reside in external memory. Because part of the code and/or data resides in the FPGA and part resides outside the FPGA, tools must exist that can properly segment the code for simulation. Data2BRAM solves this problem.

Software engineers prefer to execute their code on a real system, and be able to debug it. When software engineers are confronted with a code problem, they follow the age-old process: debug, edit, compile, link, load. This process is often repeated a large number of times a day. When some of the code resides in the FPGA's BRAM, and some in an external SRAM, interesting problems can result. For example, the download tool must now be able to write to both the BRAM and the SRAM in order to be able to load the new code in. Once the software application is ready to ship, it must be permanently stored somewhere in the system. If some of that code resides in FPGA BRAMs, the configuration bits of the FPGA may have to be preloaded with the proper elements of the software. Data2BRAM also solves these kinds of problems.

The Data2BRAM tool offers features that will greatly accelerate the design process. The tool can be used in two different modes: Stand-alone and Library. When used as a stand-alone tool, a command line interface is provided to access all of the features. This mode is most often used by software engineers when debugging code, and by hardware engineers when simulating the design. The library version of Data2BRAM is called by the Xilinx FPGA Implementation tools. This mode of operation is used primarily when the hardware

engineer is creating the FPGA logic via synthesis and the implementation tools. The library mode of Data2BRAM is not really noticed -- it just quietly does its job as part of the FPGA tool flow.

The remainder of this document will discuss some of the basic features of Data2BRAM. How the tool does its job is described in a primer using a set of examples. A short tutorial section on running Data2BRAM (in stand-alone mode) will also be included. Lastly, a reference section is provided which illustrates the command line options, file formats, and other useful information.

## Feature Summary

Data2BRAM can be used in a few different ways. The following summarizes the major features that the tool provides:

### Simulation Features

In functional and/or timing simulation, the stand-alone Data2BRAM tool can

- use **.mem**, **.elf** and/or **.drf** file inputs to produce verilog **.v** files which will initialize the contents of BRAM models for simulation
- use **.mem**, **.elf** and/or **.drf** file inputs to produce verilog or VHDL **.mem** files which will initialize the contents of external memory device models for simulation
- combine the creation of BRAM and external memory device models initialization files for simulation

### Software Debug Features

Software engineers may use the stand-alone Data2BRAM tool to

- use **.elf** or **.drf** files to initialize memory contents for software simulation
- quickly modifies the **.bit** files to update the BRAMs, outputting a new **.bit** file without the software engineer required to run Xilinx FPGA implementation tools

### FPGA Implementation Features

The Xilinx ISE 4.1i tools use the built-in library version of Data2BRAM tool to

- allow **.mem**, **.elf** and/or **.drf** files to initialize BRAM contents in the FPGA bitstream

## A Primer on How Data2BRAM Works

If you are already familiar with the Data2BRAM tool, you may wish to skip over this primer. For the purposes of this section, the stand-alone version of Data2BRAM tool will be used unless otherwise specified. Note that the functionality described is embedded in the tool whether it be hidden 'under the hood' in the FPGA implementation tools, or used as a stand-alone tool. The following discussion is designed to provide a reasonably detailed understanding of how the tool does its various functions. Greater detail can be found in the **Technical Reference for Data2BRAM**, page 1213.

The following points will be covered in this tutorial: **General Concepts**; **Simulating Designs With Data2BRAM**; **Software Debugging Using Data2BRAM**; and **FPGA Implementation Tools Interaction**.

### General Concepts

In general, Data2BRAM accepts an input file of data and produces a (set of) output file(s) that map the input file data to a plurality of memory devices. These output files are used by other tools to 'load' the contents of the memory devices for that tool's use. The power of Data2BRAM lies in its ability to accept several differing input files and producing differing output files.

#### Input / Output File Types

For example, Data2BRAM can accept Executable Load Format (.elf) files, DWARF format (.drf) files, .bit files, and verilog .mem files as input formats. The output formats of Data2BRAM are Xilinx .bit and .ucf, verilog .v and .mem, as well as VHDL .vhd files. The tool requires an additional input file called a Block Memory Map, or .bmm file, as well. The .bmm file contains a human readable textual description of how various memories are organized in a system. Figure 2-1 shows the basic input /output formats of the tool.

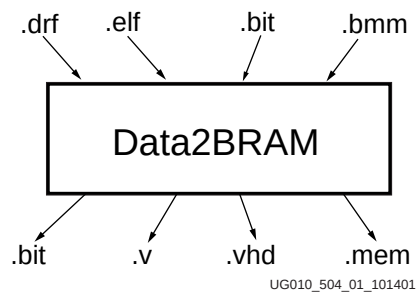


Figure 2-1: Basic Input & Output Files of Data2BRAM

## The Required .bmm File

The **.bmm** file contains a set of keywords which help Data2BRAM understand how the memory of the system is built. It has constructs to define contiguous regions of memory space, and constructs to define the organization of the memory device(s) within each of the contiguous regions. The keyword which begins the definition of a contiguous region is **ADDRESS\_BLOCK**. The keyword which ends the definition of a contiguous region is **END\_ADDRESS\_BLOCK**. Below is a small snippet from a **.bmm** file:

```
ADDRESS_BLOCK my_brams RAMB16
 BUS_BLOCK [0xFFFFC000:0xFFFFFFFF]
 top/my_mem/RAM0 [0:7];
 top/my_mem/RAM1 [8:15];
 top/my_mem/RAM2 [16:23];
 top/my_mem/RAM3 [24:31];
 top/my_mem/RAM4 [32:39];
 top/my_mem/RAM5 [40:47];
 top/my_mem/RAM6 [48:55];
 top/my_mem/RAM7 [56:63];
 END_BUS_BLOCK;
END_ADDRESS_BLOCK;
```

The **.bmm** file also needs the ability to describe how the memory devices are organized within the memory space defined by the **ADDRESS\_BLOCK**. Consider the case where a 64-bit data bus exists, and the hardware uses byte wide memory devices to populate the bus. This requires a total of eight memory devices to fill the 64-bit data bus. Data2BRAM needs to understand which data bits connect the byte wide memory devices to the bus. The tool needs to know because each of the eight memory devices must to be loaded with the data unique to their position on the data bus. In the example, the bus is 64 bits wide, but there are only byte wide RAMs. Label them RAM0 through RAM7. Assume that RAM0 only connects to bits [0:7] of the data bus. In this case, the data contents of RAM0 must hold the data for logical addresses 0x00, 0x08, 0x10, 0x18, 0x20, etc. Note that every eighth byte is contained in RAM0. The **.bmm** file uses another set of constructs to define how the memory devices are organized within the **ADDRESS\_BLOCK**. The keyword which starts the definition of memory device organization is **BUS\_BLOCK**. The keyword which ends the definition of the memory devices is **END\_BUS\_BLOCK**. The above code snippet illustrates an example usage of these two keywords in a **.bmm** file.

In the **BUS\_BLOCK** the individual memory devices are defined one at a time. The above snippet illustrates the example of the 64-bit data bus that connects to eight memory devices. Note that each line defines a specific instance name in the design. This is used by Data2BRAM to correctly identify the memory devices in verilog and/or VHDL simulations. It is also used to produce the proper bit stream if the memory devices are FPGA BRAMs. In addition to the instance name, each line within the **BUS\_BLOCK** defines the exact bit position(s) that the memory device will hold. Data2BRAM calculates the total size of the data bus for this **ADDRESS\_BLOCK** by looking at the number of bits contained within the **BUS\_BLOCK**.

The lines in the **.bmm** file with keywords also have attributes which follow them. For example, the **ADDRESS\_BLOCK** keyword has a name and memory device type associate with the block. This information is used by Data2BRAM to ensure that all contiguous memory spaces are uniquely identified, and to identify the specific type of memory devices which are being used in the **ADDRESS\_BLOCK**. Similarly, the **BUS\_BLOCK** keyword has an address range associated with it. This is used by Data2BRAM to know how much memory space is being defined within the **BUS\_BLOCK**. In addition, for FPGA BRAMs, the tool checks that the memory space defined matches what it calculated for the memory space size. This error checking can be important to ensure that each memory device gets properly loaded.

Remember that the **ADDRESS\_BLOCK** keyword defined which type of memory device will be used for this contiguous range of memory space. In the above **.bmm** snippet, the

memory device was a RAMB16 memory device. This is a 16 Kb BRAM device. The content within the `BUS_BLOCK` describes the exact physical layout of the memory devices. It begins with a physical instance name of each device, and ends with a listing of which bits of the bus that device is connected to. This information is used by Data2BRAM to identify how wide the data bus is which is connected to the memory devices. In the above `.bmm` snippet, each line in the `BUS_BLOCK` section defines a separate instance, thus there are eight instances. Each instance must be the same width, which in this case is eight bits. Data2BRAM reads the `ADDRESS_BLOCK` section of our example, and finds a `BUS_BLOCK` which defines 64 bits of data width. The tool knows that each of the eight RAMB16s will be organized as 2K x 8 bits. It figures this out because the `BUS_BLOCK` defined each RAMB16 as 8-bit data path (16Kb / 8 = 2K x 8).

Data2BRAM identified in the above `.bmm` file snippet that eight RAMB16s were used. It also calculated that each of those would be 2K x 8 wide. Since it is told (via the `.bmm` file) that the data bus width is 64-bits, Data2BRAM can calculate that the total memory space defined in this `ADDRESS_BLOCK` is 128Kb, organized as 2K x 64 or 16KB. Once Data2BRAM has calculated the size of the memory devices, it will compare the memory space defined in the `BUS_BLOCK` keyword to make sure it matches the memory space calculated.

**NOTE:** Data2BRAM will only check the memory space size for BRAM models. This is because it has no way to know the memory space of external memory models such as DDR SDRAM, Flash ROM, or SRAM.

The `.bmm` file format allows for multiple `ADDRESS_BLOCK` to define different regions of memory. For example, to specify where BRAM, and SRAM are located in the memory map. The `.bmm` file can have an unlimited number of `ADDRESS_BLOCK` entries. The only caveat is that each `ADDRESS_BLOCK` must be contiguous, without gap memory space.

Additionally, the `.bmm` file can have multiple `BUS_BLOCK` within a single `ADDRESS_BLOCK`. This means that the memory can be made up of 'banks' of memory devices all of the same width. Consider needing to build a 64KB memory out of RAMB16 type BRAMs. This would require a total of 32 BRAMs. If the data bus is organized as 64-bits, then there are a few choices to make in how the BRAMs can be hooked up. For instance, the BRAMs can be organized into four banks as 4 banks x 8 BRAMs/bank x 8 bits/BRAM. Another possibility could be two banks organized as 2 bank x 16 BRAMs/bank x 4 bits/BRAM. Both of these possibilities can be represented using multiple `BUS_BLOCK` representations. This has a very important consequence on Data2BRAM because it affects how each BRAM has its contents initialized. Note that the x8 BRAMs are organized differently in their internal bit structure than the x4 BRAMs. Thus, Data2BRAM will properly initialize the bits if the `.BMM` file matches how the memory devices are actually connected in the FPGA.

There are no hard and fast rules about how the memory subsystem should be organized. This will vary due to many factors. However, it is very important that the `.bmm` file reflect how each memory space in the system is layed out. If it does not, then the memory devices such as BRAMs, and DDR SDRAMs will not have their data content properly initialized.

Data2BRAM does not care if the multiple `ADDRESS_BLOCK` each represent different data bus widths. That is, `.bmm` file can properly define an 8 bit wide data bus in one `ADDRESS_BLOCK` while allowing a 64 bit wide data bus in another `ADDRESS_BLOCK`. Data2BRAM will correctly create the relevant output files based upon the `.bmm` file.

**NOTE:** Data2BRAM does not currently permit "holey" address space. For example, putting an eight-bit RAM on a 64-bit bus such that it is only available in one byte lane is not allowed in `.bmm` syntax. If the `.bmm` indicates the memory device(s) is (are) eight bits wide, then that memory space is treated as being byte accurate addressable. Similarly, if a `.bmm` file shows a 32-bit wide bus block, then that memory space is treated as being 32-bits wide for the entire range.

## The MEMORY Keyword in a .bmm File

The `ADDRESS_BLOCK` keyword line contains a unique name for the `ADDRESS_BLOCK`, as well as an additional keyword which defines the kind of memory device for this block. The valid keywords that Data2BRAM currently accepts in this position are: `RAMB4`, `RAMB16`, and `MEMORY`. The `RAMB4` and `RAMB16` define two different flavors of FPGA BRAMs. The first



defines a 4Kb BRAM used in Spartan and Virtex families while the second defines a 16Kb BRAM used in the Virtex-II and Virtex-II Pro families.

The **MEMORY** key is used to indicate to Data2BRAM that this **ADDRESS\_BLOCK** contains external memory devices, of an unknown memory array size. This keyword is used whenever memory devices such as SRAM or Flash are used in the system. Data2BRAM will no longer be able to compare the size of the memory space to that specified in the **.bmm** file because it does not know how big the devices are. However, the **.bmm** file still tells Data2BRAM all it really needs to know. The **ADDRESS\_BLOCK** defines as many **BUS\_BLOCK** (e.g. banks) of memory as are required to properly reflect the organization of the memory device models. Each **BUS\_BLOCK** (if more than one in the **ADDRESS\_BLOCK**) is looked at to verify they all specify the same number of data bus bits. (i.e. each **BUS\_BLOCK** specifies 64-bits)

The **MEMORY** keyword allows Data2BRAM to figure out how to correctly output the initialization information for simulation of external memory models such as Flash memory. One very important aspect of the memory keyword is that unlike BRAM4 or BRAM16 which have a direct 1:1 correspondence to the physical BRAM, memory models of external devices vary dramatically in how they are built. For example, some 16-bit DDR SDRAM memory device models are organized as banks of byte wide memory, say four banks of eight bits each. Others device models of the same size RAM might be organized as two banks of 16-bit wide memories. It is important to understand how the third party memory model is constructed so that it can be properly indicated in the **.bmm** file.

When external memory devices are to be initialized by Data2BRAM from **.mem**, **.elf**, and/or **.drf** files, it is important to know which output file is to be associated with which memory mode. The below **.bmm** snippet illustrates an example of an external 64-bit wide Flash ROM memory space as it might appear in the **.bmm** file. Note that the Flash devices are each 16-bits wide, and that an address range of 0x18000000 to 0x19FFFFFF has been defined.

```
ADDRESS_BLOCK FLASH_memory MEMORY
 BUS_BLOCK [0x18000000:0x19FFFFFF]
 testbench/flash_mem3 [63:48] OUTPUT = flash3.mem;
 testbench/flash_mem2 [47:32] OUTPUT = flash2.mem;
 testbench/flash_mem1 [31:16] OUTPUT = flash1.mem;
 testbench/flash_mem0 [15:0] OUTPUT = flash0.mem;
 END_BUS_BLOCK;
END_ADDRESS_BLOCK;
```

The main difference between this **.bmm** snippet and the previous one is that the **MEMORY** keyword is used. Note also that each instance line within the **BUS\_BLOCK** has an **OUTPUT =** keyword. The **OUTPUT =** keyword tells Data2BRAM what file name should be associated with the specified data bits. This information is used by Data2BRAM to make sure that only the correct bits of data associated with the memory device are to be output to the given file name. In the above snippet, **flash\_mem0** will be written with the contents of every fourth set of 16-bits of data. If the **.bmm** file did not permit the ability to tell Data2BRAM which bits go with which output file, there would be no ability to output simulation data for external memory devices. It should also be noted that the above snippet shows the data bus bits as being **[15:0]** syntax, where the previous snippet had **[7:0]**. These do have meaning in the **.bmm** file. Please see **How Data2BRAM Handles Bitwise Twisting of Data**, page 1215 for more information on data bit ordering in **.bmm** files.

The Data2BRAM tool essentially ignores the instance information in the **BUS\_BLOCK** because it is not required to produce the output file. However, the data bus size must be set to match that of the physical device model. Some third-party memory models require an unusual interleaving of files to build a single image of that one memory device. Thus, it is possible that two or four **.mem** files may be required to 'fill' the memory model. SDRAM devices, including DDR SDRAM devices are typically made of several banks of memory. These devices will inherently require multiple **BUS\_BLOCK** if they are to be properly

initialized. Care must be taken to understand how the memory simulation device model work before creating the **.bmm** file.

In the Embedded Reference System of the Virtex-II Pro Developer's Kit, several different kinds of memory are used. The Embedded Reference System contains BRAM in varying organizations, as well as DDR SDRAM, ZBT SRAM, Flash ROM, and Asynchronous SRAM. The BRAMs utilize Xilinx FPGA components, and are thus modeled inside the Xilinx simulation library. All of the external memories are from third parties. The DDR model is organized differently from the ZBT. The ZBT is organized differently than the Flash. The Flash models are organized different from the SRAM. As you can see, representing all of these memory types, as well as memory organizations is fully within the scope of the **.bmm** file and Data2BRAM. Please see , [Appendix A - Example Data2BRAM .bmm File](#) for a complete list of the Virtex-II Pro Developer's Kit, Embedded Reference System memory\_init.bmm file.

## Types of Output Files

Data2BRAM has several differing file types it can create. When simulating a design, an engineer requires that the memory contents of the design be initialized to certain values. Consider a simulation of a Virtex-II Pro device which has some of its PowerPC code in BRAM and some in external SRAM. The simulator has no direct means of knowing what PowerPC code there is in these models without initializing them to the values contained in the software object file.

### Verilog .v Files

The Data2BRAM tool allows the creation of verilog **.v** files for BRAMs in a FPGA. These **.v** files contain `defparam` records which cause the verilog simulator to initialize the contents of referenced BRAMs. Data2BRAM will produce a single **.v** file that contains all the initialization records for the BRAMs specified in the **.bmm** file. The verilog **.v** file produced by Data2BRAM is simply included in the top level of the design. This works because the `defparam` records reach down into the rest of the verilog source. Please see , [Appendix B - Example Data2BRAM .v File Output](#) for a sample of a verilog **.v** file output from Data2BRAM.

**Note:** **.v** file can be specified for any memory type, but will only be produce for RAMB4 and RAMB16s. The MEMORY type will inherently cause **.mem** files to be produced.

### .mem Files

Data2BRAM can also produce verilog **.mem** files. These files are used in simulation along with the BRAMs **.v** file (if present). Data2BRAM creates the **.mem** file so a memory device external to the FPGA can have its device model contents initialized to the correct values from the input file(s). The **.mem** file is then read in at the beginning of simulation, thereby initializing the memory of the system. An example of a **.mem** file for an eight bit SRAM can be found in , [Appendix C - Example Data2BRAM .mem File Output](#).

**Note:** VHDL **.mem** files require that a VHDL wrapper be written to read in the **.mem** file. It VHDL **.mem** output from Data2BRAM is identical to the verilog **.mem** output.

Data2BRAM can also be used to generate images for EEPROMs or Flash memories. In the past, the **.elf** file had to be broken down into four interleaved image files for each of the byte wide EEPROMs in a 32 bit system. Data2BRAM can automatically generate the interleaved files (**.mem** output) from the **.elf** file. This is useful when other tools for this purpose are not otherwise available.

### .ucf Files

When customers need to ensure that the BRAM contents are initialized to known values at the end of FPGA configuration, Data2BRAM can produce **.ucf** files. The **.ucf** file is used by the Xilinx FPGA Implementation Tools to set user constraints. This includes the ability to define the initialized contents of BRAMs. The **.ucf** file that Data2BRAM produces can be combined with the other **.ucf** file that the customer would normally use. If the **.ucf** files are not combined, the Xilinx Implementation Tools can generate the BRAM **.ucf** file internal to

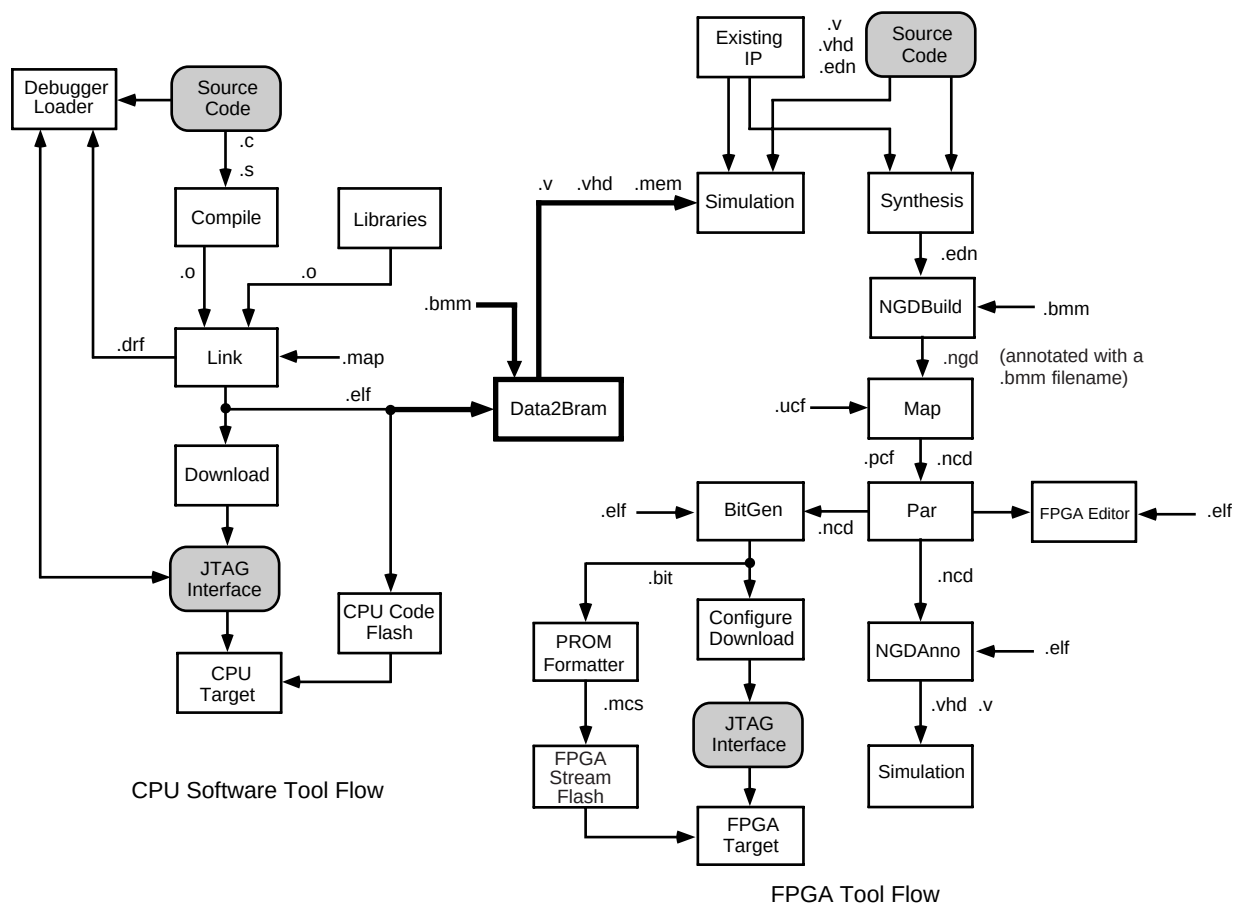
the tools. The implementation tools do this by calling Data2BRAM as a library. Please see , **Appendix D - Example Snippet of Data2BRAM .ucf File** for a sample.

## How Data2BRAM Fits in with Hardware and Software Flows

The Virtex-II Pro family offers PPC405 CPUs with large amounts of BRAM. In some systems, all of the PowerPC code may be contained within these BRAMs. Other systems might choose to put interrupt handling code in BRAM to ensure low latency. In either case, when the software code has to be placed in the FPGA's BRAM, a new set of challenges emerge. For instance, how does the FPGA bitstream get updated with the software contents? This problem has not been faced before because FPGAs did not contain CPUs. Now that they do, the hardware and software tool chains must interact with one another. The Data2BRAM tools acts like a bridge between the hardware and software worlds.

Figure 2-2 illustrates what the typical software and hardware flow look like. Notice that Data2BRAM is the bridge spanning between them. This is because it takes the input files from the software side of the world and generates output files that the hardware side can understand.

The software flow begins with source code which is compiled with the libraries. The



x604\_06\_090501

Figure 2-2: Software and Hardware Flows Using Data2BRAM

compiled code is then linked. During the linking stage, the software engineer can choose to output Executable Load Format (.elf) files or DWARF (.drf) files which contain additional debugging information. Generally, the software engineer can test the code by downloading the debug version of the code to the target hardware platform. This is often done via JTAG. If the software engineer has completed their debugging, (or has to ship the code!), they will use the non-debug version of the code in the .elf file to produce a ROM

image. The ROM image is typically stored in Flash ROM or some other non-volatile storage.

The hardware flow also begins with source files, typically Verilog or VHDL. These source files are combined with hardware IP functions, which can be thought of much like software libraries. The synthesis step “compiles” the hardware into an object form of the hardware called an EDIF file (.edn). The EDIF file is used by the Xilinx Implementation tools to create a finished FPGA. This is a multi-step process, but the end result of the process (BitGen) is roughly analogous to the software linking process.

The EDIF file that came from the synthesis tool goes through a preparatory tool called NGDDBuild. Following NGDDBuild, MAP and PAR are run. These tools map the EDIF file to FPGA resource, and then Place And Route the design. The hardware engineer may optionally choose to look at how the hardware got built using the FPGA Editor tool. This step is roughly analogous to a software engineer using a debugger to see what PowerPC instructions the C code gets turned into. One major difference is that with the FPGA Editor or FloorPlanner tools, the hardware engineer can graphically force the hardware code to reside in specific areas of the FPGA. The hardware engineer will now typically do a timing simulate of the design. ( The functional simulation is left off this flow diagram to keep it simpler). The NGDAnno tool is used to produce the back annotated files required for the simulator.

Once the hardware engineer is satisfied that the design works in simulation, they will generate the FPGA bitstream using the BitGen tool. This step produces .bit files which can be downloaded to the FPGA directly, or formatted to fit into a non-volatile storage device. The downloading process with the .bit file is analogous to the software process of downloading the .drf or .elf file to the CPU target. Similarly, the preparation of data for the nonvolatile storage is very similar.

Note that the hardware and software flows have a great degree of symmetry even though there are differences in the steps. In general the flow for hardware and software are similar: Edit, Compile, Link, Load, Test. The hardware flow offers simulation that can precisely identify issues -- if they're being looked for. Software may also offer simulation through the use of an Instruction Set Simulator (ISS). Software engineers typically use ISSs when they do not yet have target hardware to run on. However, with the advent of the Virtex-II Pro family, target hardware may be very quickly created. Hardware engineers now have “libraries” of IP which can rapidly build a nearly completed system. It is therefore possible that software engineers can debug entirely on the Virtex-II Pro based system.

In the hardware flow, the Xilinx Implementation Tools are used to build the FPGAs bitstream. Part of the FPGAs bitstream may contain the information needed to initialize the contents of the BRAMs used for PowerPC code execution. In order for the bitstream to contain the proper bits in the proper place, the Data2BRAM tool is called (in library form) several times throughout the FPGA implementation process. For example, it is called by NGDDBuild in order to parse the .bmm file and provide information to the downstream tools.

Data2BRAM is used in the flow whenever simulation of software is required. It is also used when the contents of the BRAMs need to become part of the bitstream. It can also be used to allow a software engineer to quickly turn the .elf file into a modified FPGA .bit file for testing in real hardware. These many facilities make the tool indispensable to the hardware and software flow.

## Simulating Designs With Data2BRAM

The Xilinx Implementation tools, in concert with the Virtex-II Pro Developer's Kit and a third party simulator offer the ability to simulate the PowerPC using a PPC405 SWIFT model. This model of the CPU permits cycle accurate hardware and software simulation of the PPC405 CPU in the FPGA. Consider the case of a hardware engineer who has a Virtex-II Pro design which has a PowerPC CPU, some BRAMs to store some of the CPU code in , and an some external DDR SDRAM. If the software engineer has segmented the code such

that the Interrupt Service Routine resides in the FPGA's BRAM, while the rest of the system's software code resides in the DDR, how can the hardware engineer simulate the design?

When the software engineer compiles and links the C source code, the software development tools they use will produce an **.elf** file. (See **Types of Output Files**, page 1209 and **How Data2BRAM Fits in with Hardware and Software Flows**, page 1210 for further details). The **.elf** file contains the PowerPC object code, and the require data of the compiled program. Unfortunately, hardware simulators do not read in **.elf** files, thus something must be done to the **.elf** file to convert it to a format that the simulator can understand.

Remember that the Data2BRAM tool is able to input a file in a particular form, say **.elf**, and convert it to another form? Data2BRAM reads in the **.elf** file, along with the BRAM Memory Map or **.bmm** file. The **.bmm** file contains a 'decoder ring' that tells the Data2BRAM tool what kind of memory, where the memory starts and ends, and what organization the memory contains. Using the **.bmm** 'decoder ring', Data2BRAM can write out Verilog **.v** and **.mem** files.

In our example system, two basic types of memory exist: FPGA BRAM, and DDR SDRAM. Each of these has a simulation model associate with it. In this case, the simulation model for the DDR requires a verilog **.mem** file. The **.mem** file is a human readable textual hexadecimal format that is used to preload the contents of the DDR once simulation begins. On the otherhand, the BRAM simulation model requires a verilog **.v** file which contains defparam statements to fill the contents of the BRAM model. Notice that Data2BRAM reads in a single **.elf** file, but can produce multiple types of output files, depending upon requirements. These features of Data2BRAM are only one aspect of its power.

The example system has several BRAMs instantiated with some particular organization. It also has a particular organization of DDR SDRAM devices. Data2BRAM must be able to properly reorganize its output files to ensure the data is correctly positioned within each individual memory device model. The **.bmm** file must be written in such a way as to properly reflect how each memory device is hooked up. This is where the real complexity begins. For example, the Virtex-II Pro PPC405 requires a 64 bit bus. A Virtex-II Pro BRAM can be organized as x1, x2, x4, x8, x16, or x32. If the hardware engineer wanted to fill the PPC405's 64 bit bus using a single bank of BRAMs, it could be done as 64 copies of x1 BRAMs, 32 copies of x2 BRAMs, etc. The 64-bit Processor Local Bus numbers its data bits as [0:63] instead of the more usual [63:0] -- and how these bits are hooked up to the memories matters a great deal because it affects how the data from our **.elf** file will be stored in each BRAM.

The example system also contains a set of DDR SDRAMs. These memories are external to the FPGA, but the system can not be simulated completely without the DDR being filled properly from the **.elf** file. The **.bmm** file will specify how the DDR is organized, for example as 32 bits wide. It will also define an address range for the memory space occupied by the DDR SDRAMs. When creating the portion of the **.bmm** file for the DDRs, the simulation model of the DDR memory device must be carefully looked at. If the simulation model of the DDR uses several **.mem** files to load the DDRs contents, then the **.bmm** file must reflect the organization and breakdown of those files. For example, some 16-bit wide DDRs SDRAMs actually have eight **.mem** files associated with each device. The DDR device may have its own internal banking which uses two **.mem** files of eight bits each per bank to load. If there are two devices in a system, then the **bmm** file must be constructed to build all 16 **.mem** files! The **.bmm** file can handle this by a combination of ordering within the **BUS\_BLOCK** as well as by using multiple **BUS\_BLOCK**.

Once the proper definition of **.bmm** file is created, and the software has produced an **.elf** file, then Data2BRAM can be run. When Data2BRAM is run, it is given command line options to tell it what to do. For example, where the **.elf** and **.bmm** files are; where to put the output file(s); what kind of output files to generate, etc.



When Data2BRAM has been run, with the proper switches and input files, it will generate all of the required output files. The simulation can now begin. When the simulator first starts up, it compiles all of the source files, including the BRAMs .v file. Once that has been completed, the simulator will load all of the .mem files into the various memory elements at time 0. This preloads the contents of all the memories of the system so that simulation can now take place.

## Software Debugging Using Data2BRAM

TBD

## FPGA Implementation Tools Interaction

TBD

## Tutorial On Running the Stand-Alone Data2BRAM

This tutorial assumes that the Virtex-II Pro Developer's Kit has been installed into a directory pointed at by the environment variable \$V2PRO. You can follow the directions provided to watch Data2BRAM do its job.

**NOTE:** This section will be completed in a later revision of this document. The stand-alone version of Data2BRAM is called within **make** in the Virtex-II Pro Developer's Kit. The **make** facility runs a script which in turn calls Data2BRAM. As a result, Data2BRAM is run in the background. For information on how to run, simply type **data2bram** on the command line of your shell. The on-screen information provides a measure of help. You can also look at the log files from **make**, to see exactly how data2bram is being called.

## Technical Reference for Data2BRAM

The following sections are designed to give some additional details on Data2BRAM. They include a command line reference, information on data bus twisting, the modified Backus-Naur Form for the .bmm file syntax, etc.

### Data2BRAM Command Line Reference

The following text is displayed when the Data2BRAM tool is run with no command line options. This built-in command line help is always available.

```
Version 1.11b October 4, 2001 (C) Copyright 2000-2001, Xilinx.
```

```
data2bram <-bm FILENAME [.bmm]> |
 <<[-bm FILENAME [.bmm]]> <-bd FILENAME [<.elf>|<.mem>] [<tag
TagName>]>
 <-o <u|v|h|m> FILENAME [.ucf|.v|.vhd|.mem]>
 <-p PARTNAME>
 -i> |
 <<-bd FILENAME [.elf]> -d> |
 -h
```

-bm FILENAME

Name of the input BMM file. If the file extension is missing, a .bmm file extension is assumed. If this is option not unspecified, the ELF root file name with a .bmm extension is assumed. If only this option is given, then the BMM file is merely syntax checked and any errors are reported.

-bd FILENAME

Name of the an input ELF or MEM file. If the file extension is

missing, .elf is assumed. If a TagName is given, only the address space of the same name within the BMM file will be used for translation. All other input file data outside of the TagName address space will be ignored.

If no further options are specified, "-o u FILENAME" functionality is assumed.

-o u|v|h|m FILENAME

Name of an output file. The string preceding the file name indicates which file formats are to be output. No spaces can separate the file type characters, but can appear in any order. As many, or as few file type characters can be used at once. The file type characters mean:

'u' = UCF file format, a .ucf file extension.  
'v' = Verilog file format, a .v file extension.  
'h' = VHDL file format, a .vhd file extension.  
'm' = MEM file format, a .mem file extension.

The FILENAME applies to all specified output file types. If the file extension is missing, the appropriate file extension will be added to specified output file types. If the file extension is specified, the appropriate file extension will be added to the remaining file formats.

-p PARTNAME

Name of the target Virtex part. If this is unspecified, a 'xcv50' part is assumed. Use the -h option to obtain the full supported part name list.

-i

Ignore data outside the address space defined in the BMM file.

-d

Dump the contents of the input ELF file as formatted hex records.

-h

Print this text, plus support part name list.

Supported Virtex-II PRO family devices:

xc2vp50 xc2vp20 xc2vp7 xc2vp4 xc2vp2

Supported Virtex-II family devices:

xc2v10000 xc2v8000 xc2v6000 xc2v4000 xc2v3000 xc2v2000  
xc2v1500 xc2v1000 xc2v500 xc2v250 xc2v80 xc2v40

Supported Virtex-e family devices:

xcv3200e xcv2600e xcv2000e xcv1600e xcv1000e xcv600e  
xcv400e xcv300e xcv200e xcv100e xcv50e xcv812e xcv405e

Supported Virtex family devices:

xcv1000 xcv800 xcv600 xcv400 xcv300 xcv200 xcv150 xcv100  
xcv50

Supported Spartan-II family devices:

xc2s200 xc2s150 xc2s100 xc2s50

## How Data2BRAM Handles Bitwise Twisting of Data

The format of the **.bmm** file permits the specification of how every data bit of every memory element is used in the system. Consider the example depicted in **Figure 2-3**, which shows a 64KB memory built from 32 BRAMs for PPC405 usage. The memory is organized as four banks of 64 bits x 2K, which equals 16KB per bank. Every bank consist of eight BRAMs which are organized a 2K x 8-bits each. Note that all banks are identical -- a requirement of Data2BRAM.

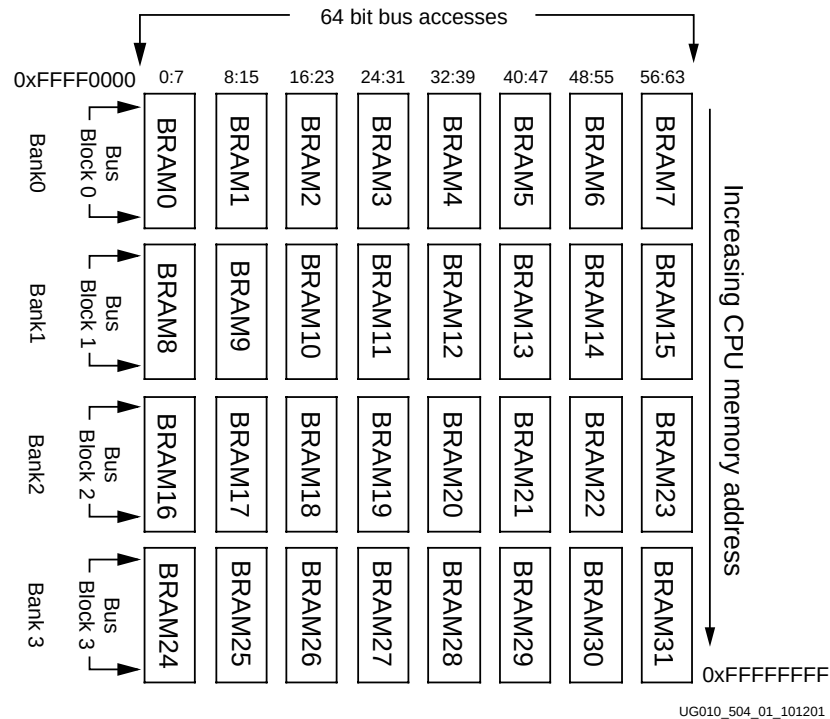


Figure 2-3: Example Layout of a 64KB Memory for PPC405 using BRAMs

**Figure 2-3** illustrates four banks, Bank0 through Bank3, as well as the 32 BRAMs, BRAM0 through BRAM31. Notice that the CPU's 64-bit data bus is labeled across the top in 8-bit chunks, and that each set of bitlanes is common to the four banks. For example, BRAM0, BRAM8, BRAM16, and BRAM24 all are hooked up to data bus bit[0:7]. Each Bank of the memory structure will be represented as a separate `BUS_BLOCK`. Each BRAM within the bank will be referenced within the `BUS_BLOCK`. The following snippet from a **.bmm** file illustrates how the whole `ADDRESS_BLOCK` would be built for this memory array:

```
ADDRESS_BLOCK my_brams RAMB16
 BUS_BLOCK [0xFFFF0000:0xFFFF3FFF]
 top/my_mem/BRAM0 [0:7];
 top/my_mem/BRAM1 [8:15];
 top/my_mem/BRAM2 [16:23];
 top/my_mem/BRAM3 [24:31];
 top/my_mem/BRAM4 [32:39];
 top/my_mem/BRAM5 [40:47];
 top/my_mem/BRAM6 [48:55];
 top/my_mem/BRAM7 [56:63];
 END_BUS_BLOCK;

 BUS_BLOCK [0xFFFF4000:0xFFFF7FFF]
 top/my_mem/BRAM8 [0:7];
 top/my_mem/BRAM9 [8:15];
 top/my_mem/BRAM10 [16:23];
```



```

 top/my_mem/BRAM11 [24:31];
 top/my_mem/BRAM12 [32:39];
 top/my_mem/BRAM13 [40:47];
 top/my_mem/BRAM14 [48:55];
 top/my_mem/BRAM15 [56:63];
 END_BUS_BLOCK;

 BUS_BLOCK [0xFFFF8000:0xFFFFBFFF]
 top/my_mem/BRAM16 [0:7];
 top/my_mem/BRAM17 [8:15];
 top/my_mem/BRAM18 [16:23];
 top/my_mem/BRAM19 [24:31];
 top/my_mem/BRAM20 [32:39];
 top/my_mem/BRAM21 [40:47];
 top/my_mem/BRAM22 [48:55];
 top/my_mem/BRAM23 [56:63];
 END_BUS_BLOCK;

 BUS_BLOCK [0xFFFFC000:0xFFFFFFFF]
 top/my_mem/BRAM24 [0:7];
 top/my_mem/BRAM25 [8:15];
 top/my_mem/BRAM26 [16:23];
 top/my_mem/BRAM27 [24:31];
 top/my_mem/BRAM28 [32:39];
 top/my_mem/BRAM29 [40:47];
 top/my_mem/BRAM30 [48:55];
 top/my_mem/BRAM31 [56:63];
 END_BUS_BLOCK;

 END_ADDRESS_BLOCK;

```

In the above **.bmm** file snippet, four **BUS\_BLOCK** are defined within a single **ADDRESS\_BLOCK**. Since [Figure 2-3](#) shows the BRAMs as being a contiguous set of memory, all 32 BRAMs can be defined within this single **ADDRESS\_BLOCK**. The **.bmm** snippet indicates that the memory will start at address 0xFFFF0000 and end at address 0xFFFFFFFF, a total of 64KB of memory. Note that each **BUS\_BLOCK** defines 64-bits of space, and that each BRAM has a line in each of the **BUS\_BLOCK**. Each **BUS\_BLOCK** specified which addresses are valid for that **BUS\_BLOCK**. The sum of these provides the 64KB of memory.

The data bus on the PPC405 CPU is numbered as [0:63], where bit 0 represents the Most Significant Bit (MSB) and 63 represents the Least Significant Bit (LSB). These are just numbers and do not inherently define where the bytes are. In the case of the PPC405, Bits [0:7] contain the data byte at address  $n+0$ , while Bits [56:63] contain the data byte at address  $n+7$ . Bit 7 and bit 63 are the LSBs of data byte  $n+0$  and  $n+7$  respectively, while bit 0 and 56 represent the MSBs of those bytes. The point of this is that in all cases, the MSB is on the right and the LSB is on the left -- regardless of what the bit numbers might seem to say. Notice that [Figure 2-3](#) is numbered using the PPC405 numbering convention.

Recall that each of the instance lines in a **BUS\_BLOCK** represent a physical BRAM. Data2BRAM will always interpret the bit lane definitions the same way: The MSB of the memory device is on the right and the LSB of the memory device is on the left. This has nothing to do with how those bits are numbered, rather, how they are interpreted.

Consider the decimal number 123456. The digit "6" is the least significant digit while the "1" is the most significant digit. If we number the "digits" like we do bits in a data bus, it doesn't matter if we number our six total digits as digit0 to digit5 or digit5 to digit0. On the other hand, if we swap the *content* of the digits to be 654321, we have a very different number! In our decimal number system, we treat the number on the left as being most significant. Data2BRAM treats every instance record in a **BUS\_BLOCK** the same. The fact that it is a binary digit instead of a decimal digit has no effect. Thus, Data2BRAM when reading the **.bmm** file assumes that the left most digit of the bit lane declaration within the **BUS\_BLOCK** will be most significant to the memory device. Most byte-wide memory

devices have data bus labeled [7:0], where bit position 7 is the most significant, and 0 the least. Accordingly, we tend to treat the labeling of memory devices to also affect their data content.

If a byte-wide memory device is looked at in greater detail, it would typically consist of a bunch of verilog or VHDL code that surrounds a register array. The surrounding code allows the simulation model to act like the real device would, i.e. respond to read and write transactions. The register array for the simulation model will often be treated as an array of eight bits registers. This array is loaded from the **.mem** file at time 0 in the simulation. The bits in the **.mem** file are just normal unsigned bytes of data. The least significant bit of the first byte in the **.mem** file get stuffed into the least significant register of the first byte in the register array. Similarly, the most significant bit gets loaded to the most significant register. The process of continuing the loading of bits in the array continues until all are stuffed in. This means that the register array now contains a perfect image of what was in the **.mem** file. In all cases what is loaded in the **.mem** file gets stuffed as one to one into the memory array.

This is well and good if the data bus bits are truly hooked up such that the LSB of the bus is the LSB of the memory. If they are swapped, then unusual things can happen. **Figure 2-4** below illustrates two byte-wide BRAMs, BLOCK\_RAM\_A and BLOCK\_RAM\_B. On the left side of the diagram, BLOCK\_RAM\_A is shown to have the data bus DBUS[7:0] hooked up to the BRAM's data pins DI[7:0]. Notice that DBUS[7] is connected to DI[7]. In contrast, BLOCK\_RAM\_B has DBUS[0:7] tied to DI[7:0] such that DBUS[0] is connected to DI[7].

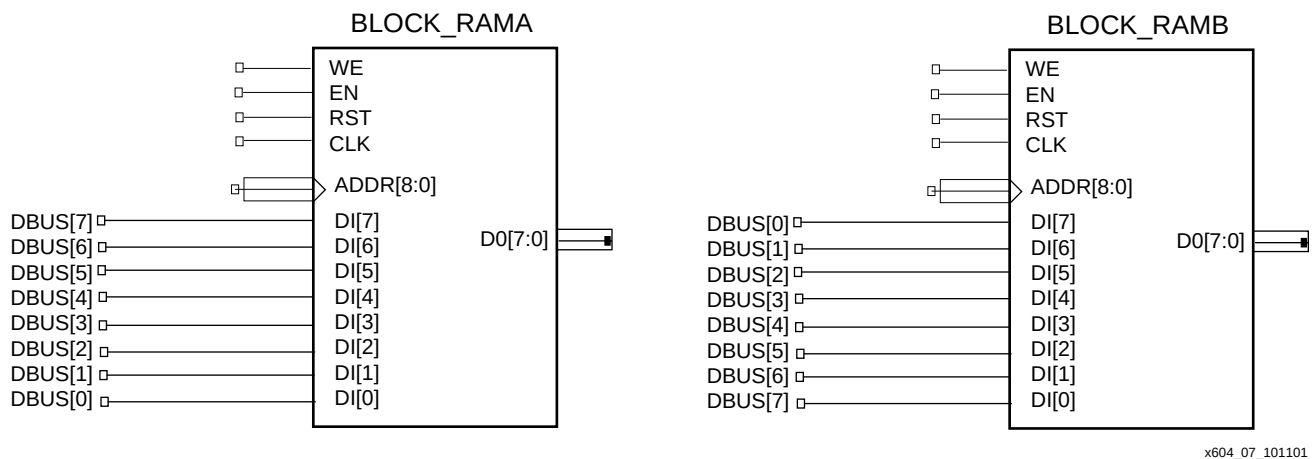


Figure 2-4: Example of Bitwise Swapping on BRAMs

The question is: How would a **.bmm** file represent each case? The answer will likely surprise you. The fact is that it would take four **.bmm** file snippets to represent the possible combinations. Why? The answer lies in the discussion about what bit number of the DBUS is MSB and which is LSB. Consider the following **.BMM** snippet:

```
ADDRESS_BLOCK RAM_BLOCKA RAMB16
BUS_BLOCK [0xFFFFC000:0xFFFFC7FF]
RAM_BLOCKA/block_ram0 [7:0];
END_BUS_BLOCK;
```

Data2BRAM will take this to mean that bit 0 is the least significant bit of the data bus and that bit 7 is the most significant. It also means that Data2BRAM expects that data bus bit 0 is connected physically to BRAM data bit 0. Remember that our byte-wide BRAM has DI[7:0] where 0 is least significant.

The **.bmm** can also be represented (legally in **.bmm** syntax) as:

```
ADDRESS_BLOCK RAM_BLOCKA RAMB16
BUS_BLOCK [0xFFFFC000:0xFFFFC7FF]
RAM_BLOCKA/block_ram0 [0:7];
```

```
END_BUS_BLOCK;
```

This syntax will not cause an error from Data2BRAM, but it also will not work in simulation! The reason it will not work is that the **.bmm** file has told Data2BRAM that the hardware is hooked up as follows: Data bus Bit 0 is tied to BRAM data bit 7, etc. This is an error in the **.bmm** file, or in the hardware hookup -- it is a matter of perspective. Said differently, the **.bmm** file does not match what the hardware does. This would only be caught in simulation, or in real operation of the FPGA *if* the BRAMs are preloaded. If the hardware is writing a location and then reading it back, no error will result. If the hardware reads a location that was initialized by Data2BRAM though, the data read back would be *bit reversed*.

The situation with BLOCK\_RAMB is nearly identical. In the below **.bmm** snippet, the BRAMs match the way that the hardware was connected.

```
ADDRESS_BLOCK RAM_BLOCKB RAMB16
BUS_BLOCK [0xFFFFC000:0xFFFFC7FF]
RAM_BLOCKB/block_ram0 [0:7];
END_BUS_BLOCK;
```

Notice that this looks just like the previous example! The difference is that now the hardware and **.bmm** file actually match each other. Accordingly, this **.bmm** file will allow Data2BRAM to generate a proper simulation file.

The last example to consider, as might be guessed, will not correctly work in simulation. The following **.bmm** snippet is presented for completeness of the discussion:

```
ADDRESS_BLOCK RAM_BLOCKA RAMB16
BUS_BLOCK [0xFFFFC000:0xFFFFC7FF]
RAM_BLOCKA/block_ram0 [7:0];
END_BUS_BLOCK;
```

This example once again shows that the actual physical data bus is hooked up differently than the logical one represented in the **.bmm** file.

What does all of this mean then? The following points can be derived:

- The number bit interpretation is very important to make sure that LSB of BRAM device is hooked to LSB of the data bus -- regardless of how each is numbered.
- The **.bmm** file must be constructed to reflect this LSB to LSB correspondence.
- Failure to properly connect LSB to LSB will result in errant behavior -- but that behavior may be difficult to detect depending upon the circumstances
- Care must be taken in the construction of the **.bmm** file to make sure that the bit widths and positions all match each other.

## Tool Behavior in Xilinx ISE 4.1i

This section describes how Data2BRAM can be utilized in the Xilinx Software Flow. This flow allows the hardware designer to associate BRAM with a **.bmm** file. Data2BRAM then performs the correlation between the **.bmm** and **.elf** file, and creates the BRAM initialization strings. Access to Data2BRAM in the software flow is done by utilizing different software options in NGDBuild, BitGen, NGDAnno, and FPGA Editor. [Figure 2-2, page 1210](#) illustrates the software flow and the file dependencies. The following sections describe each option and its effect in the software flow.

### NGDBuild Flow

**Option: -bm**

**Syntax: -bm <file\_name>[.bmm] switch**

**Usage:** This option allows you to specify the name and the path of the **.bmm** file. If the **.bmm** file is the same name as the design name and is located in the same directory as the design netlist, the **.bmm** file is loaded by default. If the **.bmm** file is added to an ISE project, ISE tracks changes to the **.bmm** file and re-implements the design when necessary. The

Xilinx Design Manager/Flow Engine (DMFE) does not track changes to the **.bmm** file. In this case, the user is responsible to restart the Xilinx software flow themselves.

**Functionality:** NGDBuild creates the “BMM\_FILE” property in the NGD file. No BRAM validation occurs. No **.bmm** file syntax checking and validation occurs by NGDBuild. The syntax can be checked by running stand-alone Data2BRAM.

**NOTE:** The **-bm** switch is supported in ISE. Design Manager users are required to use the Template Manager and Customize to set this option.

## MAP and PAR Flow

There are no command line or functionality changes to MAP or PAR. However, incorrectly connected BRAM components are trimmed by MAP. You are responsible for correctly connecting the BRAM components. Check the MAP report “Section 5 - Removed Logic” to see if any BRAM components were removed from the design

## BitGen Flow

**Option:** **-bd**

**Syntax:** **-bd <elf\_filename> switch**

**Usage:** The **-bd** switch specifies the path and file name of the **.elf** file used to populate the BRAMs specified in the **.bmm** file. The address information contained in the **.elf** file allows Data2BRAM to determine which ADDRESS\_BLOCK to place the data.

**Functionality:** BitGen passes the **-bd** switch with the **<elf\_filename>** to Data2BRAM. Data2BRAM processes the **.bmm** file specified during NGDBuild and the **.elf** file is used to create the BRAM initialization strings for each of the constrained BRAMs. The initialization strings are then used to update the **.ncd** file before the **.bit** file is created.

Placement information of the BRAM is provided by the **.ncd** file to Data2BRAM. This information is used to create the **<bramfilename>\_bd.bmm** file. This file contains the placement information for all BRAM constrained or unconstrained. This is necessary to enable the use of the stand-alone Data2BRAM program. This file must be given to the software engineer for use in the stand-alone flow.

**NOTE:** The **-bm** switch is supported in ISE. Design Manager users are required to use the Template Manager and Customize to set this option.

## NGDAnno Flow

**Option:** **-bd**

**Syntax:** **-bd <elf\_filename> switch**

**Usage:** The **-bd** switch specifies the path and file name of the **.elf** file used to populate the BRAMs specified in the **.bmm** file. The address information contained in the **.elf** file allows Data2BRAM to determine which ADDRESS\_BLOCK to place the data.

**Functionality:** NGDAnno passes the **-bd** switch with the **<elf\_filename>** to Data2BRAM. Data2BRAM processes the **.bmm** file specified during NGDBuild and the **.elf** file is used to create the BRAM initialization strings for each of the constrained BRAMs. The initialization strings are then used to update the **.ncd** file before the **.bit** file is created.

Placement information of the BRAM is provided from the **.ncd** file to Data2BRAM. This information is used to create the **<bramfilename>\_bd.bmm** file. This file contains the placement information for all BRAM constrained or unconstrained. This is necessary to enable the use of the stand-alone Data2BRAM program. This file must be given to the software engineer for use in the stand-alone flow.

**NOTE:** The **-bm** switch is supported in ISE. Design Manager users are required to use the Template Manager and Customize to set this option.

## FPGA Editor Flow

**Option:** **-bd**

**Syntax:** `-bd <elf_filename> switch`

**Usage:** The `-bd` switch specifies the path and file name of the `.elf` file used to populate the BRAMs specified in the `.bmm` file. The address information contained in the `.elf` file allows Data2BRAM to determine which ADDRESS\_BLOCK to place the data.

**Functionality:** In the 4.1i software release, BRAMs specified in the `.bmm` file will not be marked READ-ONLY in FPGA Editor. Any changes made in FPGA Editor to the contents of the BRAM mapped in the `.bmm` file will not be retained even if the user writes out the NCD. In the future, BRAMs specified in the `.bmm` file is made READ-ONLY. The `.elf` file contents must be changed at the source.

**NOTE:** The `-bm` switch is supported in ISE. Design Manager users are required to use the Template Manager and Customize to set this option

## Known Limitations

### Trimming of BRAMs in MAP

If BRAMs are improperly specified in the `.bmm` file, they will be trimmed by MAP. When Data2BRAM is run, it will generate errors. Check the map report file to see if any BRAMs have been trimmed in the design.

### VHDL Simulation

Data2BRAM currently does not produce VHDL simulation files for the initialization of BRAMs. It does produce `.mem` files which can be used in VHDL simulations if a wrapper is written to read in the `.mem` file.

### FPGA Editor BRAM values

The FPGA Editor tool does not currently call the library version of Data2BRAM. Because of this, the initialization contents of the BRAMs may not be accurately reflected when using FPGA Editor.

### No CPU Physical Address Checking in Data2BRAM

The hardware designer is responsible for ensuring that the memories specified in the `.bmm` file are mapped in the real hardware to the locations specified. Data2BRAM will only generate what the `.bmm` file tells it. The hardware designer should therefore carefully check that the hardware and `.bmm` file memory maps actually match.

## Future Enhancements

TBD

## Modified Backus-Naur Form for .bmm File Syntax

```

Address_block_keyword ::= "ADDRESS_BLOCK";

End_address_block_keyword ::= "END_ADDRESS_BLOCK";

Bus_block_keyword ::= "BUS_BLOCK";

End_bus_block_keyword ::= "END_BUS_BLOCK";

LOC_location_keyword ::= "LOC";

PLACED_location_keyword ::= "PLACED";

MEM_output_keyword ::= "OUTPUT";

BRAM_location_keyword ::= LOC_location_keyword |
 PLACED_location_keyword ;

Memory_type_keyword ::= "RAMB4" |
 "RAMB16" |
 "MEMORY";

Number_range ::= "[" NUM ":" NUM "]";

Name_path ::= IDENT ("/" IDENT) * ;

BRAM_instance_name ::= Name_path ;

MEM_output_spec ::= MEM_output_keyword "="
 Name_path [".mem"] ;

BRAM_location_spec ::= BRAM_location_keyword "="
 ("R" NUM "C" NUM) | ("X" NUM "Y" NUM) ;

Bit_lane_def ::= BRAM_instance_name Number_range
 [BRAM_location_spec |
 MEM_output_spec] ";" ;

Bus_block_def ::= Bus_block_keyword Number_range
 (Bit_lane_def) +
 End_bus_block_keyword ";" ;

Address_block_def ::= Address_block_keyword IDENT
Memory_type_keyword (Bus_block_def) +
 End_address_block_keyword ";" ;

```

## Appendix A - Example Data2BRAM .bmm File

```
//-----
//
// $Revision: 1.5 $
// $Date: 2001-10-13 00:38:06+00 $
//
//-----
// BRAM Initialization File - Configuration File
//-----
//
// THESE DESIGNS ARE PROVIDED "AS IS" WITH NO WARRANTY
// WHATSOEVER AND XILINX SPECIFICALLY DISCLAIMS ANY
// IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR
// A PARTICULAR PURPOSE, OR AGAINST INFRINGEMENT.
//
// THESE DESIGNS ARE XILINX CONFIDENTIAL MATERIAL
// PROVIDED UNDER NDA.
//
// Copyright (c) 2001 Xilinx, Inc.
// All rights reserved
//
//-----
// Filename: memory_init.bmm
//
// Description:
// This module defines the path, memory map, and organization of the
// BRAM memory used in this reference design. This file is used in
// conjunction with the Data2BRAM program to initialize the BRAM
// contents for simulation.
//
//-----
// Author: KD
// History:
// KD 5/17/01 -- EA 2 Release
// KD 10/09/01 -- EA 3 Release
//
//-----

////////////////////////////////////
// PLB BRAM Address space 0xFFFF8000 - 0xFFFFFFFF (32 KB)
////////////////////////////////////

ADDRESS_BLOCK plb_bram_controller RAMB16
 BUS_BLOCK [0xFFFF8000:0xFFFFFFFF]
 top_ip/ip_wrapper/bram_block/block_ram0 [63:60];
 top_ip/ip_wrapper/bram_block/block_ram1 [59:56];
 top_ip/ip_wrapper/bram_block/block_ram2 [55:52];
 top_ip/ip_wrapper/bram_block/block_ram3 [51:48];

 top_ip/ip_wrapper/bram_block/block_ram4 [47:44];
 top_ip/ip_wrapper/bram_block/block_ram5 [43:40];
 top_ip/ip_wrapper/bram_block/block_ram6 [39:36];
 top_ip/ip_wrapper/bram_block/block_ram7 [35:32];

 top_ip/ip_wrapper/bram_block/block_ram8 [31:28];
 top_ip/ip_wrapper/bram_block/block_ram9 [27:24];
 top_ip/ip_wrapper/bram_block/block_ram10 [23:20];
 top_ip/ip_wrapper/bram_block/block_ram11 [19:16];

 top_ip/ip_wrapper/bram_block/block_ram12 [15:12];
 top_ip/ip_wrapper/bram_block/block_ram13 [11:8];
```



```

 top_ip/ip_wrapper/bram_block/block_ram14 [7:4];
 top_ip/ip_wrapper/bram_block/block_ram15 [3:0];
 END_BUS_BLOCK;
END_ADDRESS_BLOCK;

///
// Data Side OCM Address Space 0x40000000 - 0x40001FFF (8KB)
///

ADDRESS_BLOCK dsocm RAMB16
 BUS_BLOCK [0x40000000:0x40001FFF]
 top_cpu/DSBRAM/u3 [31:24];
 top_cpu/DSBRAM/u2 [23:16];
 top_cpu/DSBRAM/u1 [15:8];
 top_cpu/DSBRAM/u0 [7:0];
 END_BUS_BLOCK;
END_ADDRESS_BLOCK;

///
// Instruction Side OCM Address Space 0x50000000 - 0x50000FFF (4KB)
///

ADDRESS_BLOCK isocm RAMB16
 BUS_BLOCK [0x50000000:0x50000FFF]
 top_cpu/ISBRAM/u2 [63:32];
 top_cpu/ISBRAM/u1 [31:0];
 END_BUS_BLOCK;
END_ADDRESS_BLOCK;

///
// OPB Configuration ROM Address Space 0xE0000000 - 0xE00007FF (2KB)
///

ADDRESS_BLOCK CROM RAMB16
 BUS_BLOCK [0xE0000000:0xE00007FF]
 top_ip/ip_wrapper/opb_crom/block_ram0 [31:0];
 END_BUS_BLOCK;
END_ADDRESS_BLOCK;

///
// DDR Memory Address Space 0x00000000 - 0x01FFFFFF (32MB)
///

ADDRESS_BLOCK DDR_memory MEMORY

 BUS_BLOCK [0x00000000:0x007FFFFFFF]
 testbench/dds1/Bank0 [31:16] OUTPUT = ddr1_bank0.mem;
 testbench/dds0/Bank0 [15:0] OUTPUT = ddr0_bank0.mem;
 END_BUS_BLOCK;

 BUS_BLOCK [0x00800000:0x00FFFFFFF]
 testbench/dds1/Bank1 [31:16] OUTPUT = ddr1_bank1.mem;
 testbench/dds0/Bank1 [15:0] OUTPUT = ddr0_bank1.mem;
 END_BUS_BLOCK;

 BUS_BLOCK [0x01000000:0x017FFFFFFF]
 testbench/dds1/Bank2 [31:16] OUTPUT = ddr1_bank2.mem;
 testbench/dds0/Bank2 [15:0] OUTPUT = ddr0_bank2.mem;
 END_BUS_BLOCK;

 BUS_BLOCK [0x01800000:0x01FFFFFFF]
 testbench/dds1/Bank3 [31:16] OUTPUT = ddr1_bank3.mem;

```



```

 testbench/DDR0/Bank3 [15:0] OUTPUT = ddr0_bank3.mem;
 END_BUS_BLOCK;

 END_ADDRESS_BLOCK;

 //
 // SRAM Memory Address Space 0x10000000 - 0x101FFFFF (2MB)
 //

 ADDRESS_BLOCK SRAM_memory MEMORY

 BUS_BLOCK [0x10000000:0x101FFFFF]
 testbench/sram3/mem2 [63:56] OUTPUT = sram_byte7.mem;
 testbench/sram3/mem1 [55:48] OUTPUT = sram_byte6.mem;
 testbench/sram2/mem2 [47:40] OUTPUT = sram_byte5.mem;
 testbench/sram2/mem1 [39:32] OUTPUT = sram_byte4.mem;
 testbench/sram1/mem2 [31:24] OUTPUT = sram_byte3.mem;
 testbench/sram1/mem1 [23:16] OUTPUT = sram_byte2.mem;
 testbench/sram0/mem2 [15:8] OUTPUT = sram_byte1.mem;
 testbench/sram0/mem1 [7:0] OUTPUT = sram_byte0.mem;
 END_BUS_BLOCK;

 END_ADDRESS_BLOCK;

 //
 // FLASH Memory Address Space 0x18000000 to 0x19FFFF (32 MB)
 //

 ADDRESS_BLOCK FLASH_memory MEMORY

 BUS_BLOCK [0x18000000:0x19FFFFFF]
 testbench/flash_mem3 [63:48] OUTPUT = flash3.mem;
 testbench/flash_mem2 [47:32] OUTPUT = flash2.mem;
 testbench/flash_mem1 [31:16] OUTPUT = flash1.mem;
 testbench/flash_mem0 [15:0] OUTPUT = flash0.mem;
 END_BUS_BLOCK;

 END_ADDRESS_BLOCK;

 //
 // ZBT Memory Address Space 0x30000000 - 0x300FFFFF (1MB)
 //

 ADDRESS_BLOCK ZBT_memory MEMORY

 BUS_BLOCK [0x30000000:0x300FFFFF]
 testbench/zbt1/bank3 [63:56] OUTPUT = zbt_byte7.mem;
 testbench/zbt1/bank2 [55:48] OUTPUT = zbt_byte6.mem;
 testbench/zbt1/bank1 [47:40] OUTPUT = zbt_byte5.mem;
 testbench/zbt1/bank0 [39:32] OUTPUT = zbt_byte4.mem;
 testbench/zbt0/bank3 [31:24] OUTPUT = zbt_byte3.mem;
 testbench/zbt0/bank2 [23:16] OUTPUT = zbt_byte2.mem;
 testbench/zbt0/bank1 [15:8] OUTPUT = zbt_byte1.mem;
 testbench/zbt0/bank0 [7:0] OUTPUT = zbt_byte0.mem;
 END_BUS_BLOCK;

 END_ADDRESS_BLOCK;

```

## Appendix B - Example Data2BRAM .v File Output

The following is a small section from the memory\_init.v file for the Embedded Reference Design. Note that this section lists the defparams for a single BRAM (64 entries). Also note that the head of the BRAM contains data while the rest has been set to zero.

```
// Verilog initialization records.
//
// Data2BRAM version 1.11b October 4, 2001
// (C) Copyright 2000-2001, Xilinx.
//
// Created on 10/14/01 05:43 pm, from:
//
// Map file - /home/bob/v2pdk/platforms/embedded/sys/memory_init.bmm
// Data file -
//
// Address space 'plb_bram_controller' [0xFFFF8000:0xFFFFFFFF],
// 32768 bytes in size.
//
// Bus width = 64 bits, bit lane width = 4 bits,
// number of bus blocks = 1.

// BRAM 0 in address block [0xFFFF8000:0xFFFFFFFF], bit lane [63:60]
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_00 =
256'h8347389354843777837783433387349748844548834597879977898758458799;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_01 =
256'h4543943942244454753488578334784782597783397787439778334789779974;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_02 =
256'h9799778754457787997488459784254399748847875349647896478738354596;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_03 =
256'h99487897978BA79948789779787787879787ABB7879787899787977858439839;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_04 =
256'h00000000000000000000A01FFF060FFF04878979787A79948489979948457A77;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_05 =
256'h0000000000000000000000000000000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_06 =
256'h0000000000000000000000000000000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_07 =
256'h0000000000000000000000000000000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_08 =
256'h0000000000000000000000000000000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_09 =
256'h0000000000000000000000000000000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_0A =
256'h0000000000000000000000000000000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_0B =
256'h0000000000000000000000000000000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_0C =
256'h0000000000000000000000007300000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_0D =
256'h0000000000000000000000000000000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_0E =
256'h0000000000000000000000000000000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_0F =
256'h0000000000000000000000000000000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_10 =
256'h0000000000000000000000000000000000000000000000000000000000000000;
defparam top_ip.ip_wrapper.bram_block.block_ram0.INIT_11 =
256'h0000000000000000000000000000000000000000000000000000000000000000;
```

[illegible]

March 2002 Release  
**Virtex-II Pro™ Platform FPGA Developer's Kit**

## Appendix C - Example Data2BRAM .mem File Output

The below is a sample **.mem** file output by Data2BRAM. This sample is taken from the Virtex-II Pro Developer's Kit, Embedded Reference Design. This particular **.mem** file can be found at \$V2PRO/platforms/embedded/sys/mem/sram\_byte0.mem. Notice that the file only contains bytes of data. This is because the SRAM was defined as being a byte-wide SRAM. Also notice that the @00000000 corresponds to the relative address into the SRAM. In other words, the SRAM is mapped at CPU Address 0x10000000, but the SRAM's contents start at its internal address 0. Lastly, notice that the **.mem** file leaves the remaining contents of the SRAM undefined. Only the actual contents which were used in the **.elf** file that produced this **.mem** are listed. The rest will be undefined if the CPU were to try to read the contents of the SRAM before writing to the SRAM.

```
// Mem file initialization records.
//
// Data2BRAM version 1.11b October 4, 2001
// (C) Copyright 2000-2001, Xilinx.
//
// Created on 10/14/01 06:08 pm, from:
//
// Map file - /home/bob/v2pdk/platforms/embedded/sys/memory_init.bmm
// Data file -
//
// Address space 'SRAM_memory', bit lane [7:0].
//
// Bus width = 64 bits, bit lane width = 8 bits,
// number of bus blocks = 1.

@00000000
A6 1C 08 00 00 2C 00 10 21 01 C4 04 FC 20 A6 1C
00 59 78 08 0D 10 B1 3E 78 08 1B 08 00 A6 78 F0
0C 78 0C AB 7E 78 00 C9 1C 25 04 04 FC 20 F8 08
20 BE 00 04 0C 40 50 A6 FC FC F8 00 A6 00 10 08
E0 1C 78 0C 78 10 12 0C 00 08 78 E5 14 ED 78 08
78 78 13 18 B5 78 08 78 7D 08 85 78 13 3E 1C 08
47 18 08 00 08 78 0D 0C 08 14 3E 78 08 14 3E 78
10 06 06 05 3C 28 2C 00 28 13 02 10 13 12 01 13
78 11 04 78 00 14 00 14 20 08 13 13 78 30 28 13
13 78 08 08 13 78 8D 10 0B 00 A6 78 E8 14 78 08
78 78 3E 08 D8 2D 78 0C 78 00 A6 78 E8 14 78 78
0C 14 C5 B4 00 08 0C 08 7D 04 FC 20 A6 24 08 10
10 00 00 08 78 78 FE 00 04 FC 20 14 08 00 78 0D
AC 3E 04 FC 20 14 08 00 78 0E AC 3E 04 FC 20 1C
08 00 10 0C 0C 04 FC 20 14 08 78 0C 00 00 78 E8
78 78 0C 08 AC FC 20 14 08 0C 00 00 78 E8 78 08
78 0E 0C 0C 78 00 78 E0 78 08 78 10 AC 78 00 78
E8 78 78 0C 08 AC FC 20 14 08 0C 2C 00 78 00 18
18 20 00 6C 0A 00 18 40 00
```

The following snippet of a **.ucf** file produced by Data2BRAM illustrates a single BRAM instance in the **.ucf** file. Note that the **.ucf** file is used by Xilinx Implementation Tools to set user constraints for the tools. One of these constraints allows the ability to initialize the BRAMs to known value after configuration. Data2BRAM uses this mechanism to program the BRAMs for the Xilinx Implementation Tools.

[illegible]

**1230**

```
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_31 =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_32 =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_33 =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_34 =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_35 =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_36 =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_37 =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_38 =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_39 =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_3A =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_3B =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_3C =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_3D =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_3E =
0000000000000000000000000000000000000000000000000000000000000000;
INST top_ip/ip_wrapper/bram_block/block_ram0 INIT_3F =
0000000000000000000000000000000000000000000000000000000000000000;

// BRAM 1 in address block [0xFFFF8000:0xFFFFFFFF], bit lane [59:56]
```





# GNU Software Tool Chain

---

## Overview

GNU software is used to develop software for the Virtex-II Pro™ family of FPGAs. This introduction concentrates on the GNU C compiler (GCC), the GNU binary utilities (binutils), the GNU debugger (GDB), and the GNU make program.

From the GNU Project website, <http://www.gnu.org>:

The GNU Project was launched in 1984 to develop a complete Unix-like operating system which is free software: the GNU system. (GNU is a recursive acronym for GNU's Not Unix; it is pronounced 'guh-NEW'.) Variants of the GNU operating system that use the Linux kernel are now widely used; though these systems are often referred to as "Linux," they are more accurately called *GNU/Linux systems*.

As a prerequisite for the development of the GNU system, many different software packages—compilers, assemblers, linkers, debuggers, libraries, and other tools—had to be programmed.

See <http://www.gnu.org/software/software.html#TOCDescriptionsOfGNUSoftware> for a complete list of packages available as free software under the terms of the GNU GPL (<http://www.gnu.org/copyleft/gpl.txt>).

## Related Documentation

GNU software documentation is available from built-in **help** pages, as **man** pages, **info** pages, and the Internet. Documentation about programming in C is available in books and on the Internet.

### Built-in Help Pages

Most of the GNU utilities have built-in help pages that can be invoked with the **--help** parameter.

Example 1: See the built-in help for the **objdump** utility.

```
$ powerpc-eabi-objdump --help
```

Example 2: See the built-in help for GCC.

```
$ powerpc-eabi-gcc --help
```

### Man Pages

Man pages are installed on your system with the installation of the GNU software, and are accessible by using the **man** program. For example, for information about GCC, issue the command:

```
$ man powerpc-eabi-gcc
```

The **man** program then displays the requested information. For the **man** program to find the corresponding **man** page, the environment variable MANPATH must be set appropriately.

To check the current value of the MANPATH, execute

```
$ printenv MANPATH
```

To include the **man** pages shipped with the GNU software in your MANPATH, execute

```
$ setenv MANPATH $V2PRO/tools/$V2PRO_HOST_OSTYPE/gnu/man:$MANPATH
```

in a **csh** or

```
$ export MANPATH=$V2PRO/tools/$V2PRO_HOST_OSTYPE/gnu/man:$MANPATH
```

in a **bash**.

## Info Pages

**Info** pages are often more informative than **man** pages. Unlike **man** pages, **info** pages use cross-references to make navigation and finding information easier. However, some packages do not offer info pages and the info program may not be installed on all systems.

**Info** pages are installed on a system during the installation of the GNU software and are accessible by using the **info** program. For example, for information about GCC, issue the command:

```
$ info powerpc-eabi-gcc
```

The **info** program then displays the requested information. For the **info** program to find the corresponding **info** page, the environment variable INFOPATH must be set appropriately.

To check the current value of the INFOPATH, execute

```
$ printenv INFOPATH
```

To include the **info** pages shipped with the GNU software in your INFOPATH, execute

```
$ setenv INFOPATH $V2PRO/tools/$V2PRO_HOST_OSTYPE/gnu/man:$INFOPATH
```

in a **csh** or

```
$ export INFOPATH=$V2PRO/tools/$V2PRO_HOST_OSTYPE/gnu/man:$INFOPATH
```

in a **bash**.

## Internet Pages

Use <http://www.gnu.org/software/software.html#TOCDescriptionsOfGNUSoftware> as a starting point. The GNU software is very well documented on the Internet.

For documentation about a specific package, consult [Table 3-1](#).

**Table 3-1: Online GNU Software Documentation**

| Package                                                           | Link                                                                                                    |
|-------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| GCC: GNU C compiler                                               | <a href="http://www.gnu.org/software/gcc/onlinedocs">http://www.gnu.org/software/gcc/onlinedocs</a>     |
| Binutils: GNU binary utilities (assembler, linker, objdump, etc.) | <a href="http://sources.redhat.com/binutils/docs-2.10">http://sources.redhat.com/binutils/docs-2.10</a> |
| Make: controls code generation                                    | <a href="http://www.gnu.org/software/make/make.html">http://www.gnu.org/software/make/make.html</a>     |

## Books About Programming in C

There are many good books about the C and C++ programming languages. A few of these are listed below:

Books about C

Kernigham Brian, Ritchie Dennis. 1988. *The C Programming Language*. Prentice Hall

Wang Paul. 1992. *An Introduction to ANSI C on Unix*. Wadsworth.

Kumar Ram, Agrawal Rakesh. 1992. *Programming in ANSI C*. West Publishing Company.

Books about C++

Stroustrup Bjarne. 1991. *The C++ Programming Language*. Addison-Wesley.

Lafore Robert. 1995. *Object-Oriented Programming in C++*. Waite Group Press.

## Online Documents About Programming in C

The Internet offers plenty of documentation about how to program in C and C++. One of the best approaches to finding documentation online is to use a search engine (such as Google, <http://www.google.com>) and search on "introduction to programming in C."

## Terminology

Table 3-2 lists the software terms commonly used in this document.

Table 3-2: Software Terminology

| Term                       | Meaning                                                                                                                                                                                                                                                                                          |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Source code                | The source code is the human readable form of a user executable. It is written in a high-level programming language like C or C++ or in low-level assembly language.                                                                                                                             |
| Object code                | The object code is the machine-readable form of a program. A compiler or an assembler translates source code into object code.                                                                                                                                                                   |
| Source file / object file  | File that contains source/object code. The name of the source file ends with the extension <b>.c</b> for C files, <b>.h</b> for C header files, and <b>.asm</b> or <b>.S</b> for assembly files.                                                                                                 |
| ELF file / user executable | The user executable is the software program written by the user (in C or assembly) that can be executed by the processor. The compiler and linker process one or multiple object files and generate the executable. The user executable is an ELF file, which carries the <b>.elf</b> extension. |
| Library                    | A collection of object files packed into one file. A library simplifies the use of standard functions.                                                                                                                                                                                           |
| Assemble                   | The process that translates an assembly source file into an object file.                                                                                                                                                                                                                         |
| Disassemble                | The process that displays the assembly code contained in an object file or in an executable.                                                                                                                                                                                                     |
| Compile                    | The process that translates a source file into an object file.                                                                                                                                                                                                                                   |
| Link                       | The process that generates a user executable from one or more object files.                                                                                                                                                                                                                      |
| Map file or linker script  | The mapfile (or linker script) tells the linker how to combine object files and where to map them in the memory space.                                                                                                                                                                           |

Table 3-2: Software Terminology (Continued)

| Term           | Meaning                                                                                                                                               |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| Make           | A program used to automate the creation of a user executable from source code.                                                                        |
| Makefile       | Tells Make how the executable is built from the source files.                                                                                         |
| Cross-Compiler | A tool (such as GNU) capable of creating a user executable for use on a platform architecture other than the platform on which the tool is being run. |

## Tool Flow

Figure 3-1 shows the necessary steps to build an executable that writes "Hello world" to a UART.

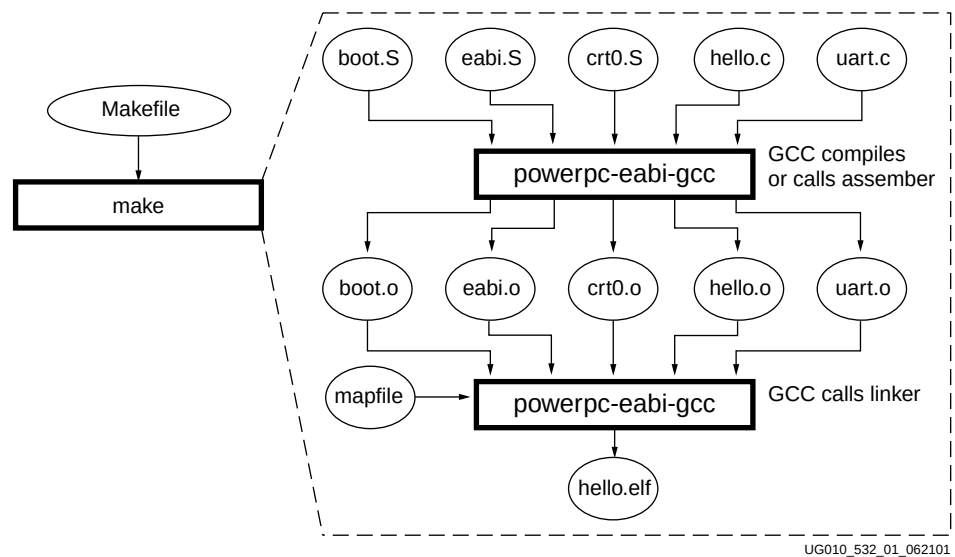


Figure 3-1: Steps for Building Hello.elf

The user initiates the build process by calling

```
$ make
```

from the command line. The program **make** reads the Makefile. The Makefile contains all the steps and instructions to build the executable **hello.elf** from the source files **boot.S**, **eabi.S**, **crt0.S**, **hello.c**, and **uart.c**. During the build process the intermediary object files **boot.o**, **eabi.o**, **crt0.o**, **hello.o**, and **uart.o** are generated.

## Make

**Make** is a utility that automates the build process. It reads the commands to perform from the Makefile and executes them step-by-step. **Make** has the ability to skip over commands that need not be executed since the result of the command already exists. This behavior speeds up the build process.

The Makefile contains targets. When **make** is called without parameters, the target **all** is executed. A specific target can be executed by giving its name as a parameter to the **make** program.

Example 1: Call **make**, build the target **all** in the current directory, and use the Makefile:

```
$ make
```

Example 2: Call **make**, build first the target **clean** and then the target **all** in the current directory, and use the Makefile:

```
$ make clean all
```

Example 3: Call **make**, build the target **all** in the current directory, and use the file named **makefile.mak** as the Makefile:

```
$ make -f makefile.mak
```

Example 4: Call **make**, build the target **clean** in the /tmp directory, and use the Makefile (in the /tmp directory):

```
$ make -C /tmp
```

## GCC

**GCC** acts as a compiler, translating files written in C into a form that can be understood and executed by the target processor. **GCC** also functions as the user interface to the GNU assembler and to the GNU linker, calling the assembler and the linker with the appropriate parameters.

Example 1: Compile all C source files (\*.c), assemble all assembler source files (\*.S), and generate the executable **hello.elf**. In this example, **GCC** compiles the C source files, initiates the assembler to assemble the assembler source files, and calls the linker to generate the executable **hello.elf**. The linker uses a built-in map file since no other map file is specified.

```
$ powerpc-eabi-gcc -o hello.elf *.c *.S
```

Example 2: Compile all C source files (\*.c) and assemble all assembler source files (\*.S) to object files (\*.o). Link the object files and generate the executable **hello.elf**. In this example, the intermediary object files are generated.

```
$ powerpc-eabi-gcc -c *.c *.S
$ powerpc-eabi-gcc -o hello.elf *.o
```

Example 3: Compile all C source files (\*.c) and assemble all assembler source files (\*.S) and generate the executable **hello.elf**. **GCC** compiles the C source files, initiates the assembler to assemble the assembler source files, and calls the linker to generate the executable **hello.elf**. The linker uses the specified mapfile as linker script.

```
$ powerpc-eabi-gcc -Wl,-T,mapfile -o hello.elf *.c *.S
```

Compiler flags, shown in [Table 3-3](#), influence the code generation of **GCC**.

**Table 3-3: Compiler Flags**

| Compiler Flag | Influence                                                                                                                                                                                                                                    |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -c            | Generate intermediary object files.                                                                                                                                                                                                          |
| -o <name>     | Name the output file to name. This can be the name of the executable or of the object file.                                                                                                                                                  |
| -gdwarf       | Generate debugging information in the dwarf format. This debugging information is included in the object files and in the executable. Do not use with one of the -O optimization levels since optimized code might not be possible to debug. |
| -Ox           | Generate optimized code. X is a place holder for the optimization level. Possible optimization levels range from 0 to 3 or s. Higher numbers indicate optimization for higher speed. S indicates optimization for smaller space.             |
| -mcpu=403     | Generate code for the IBM 403 processor. Unfortunately, GCC does not yet offer a compiler switch to generate code for the 405. This compiler switch also turns on code generation for floating point emulation in software.                  |

Table 3-3: Compiler Flags

| Compiler Flag | Influence                                                                                                       |
|---------------|-----------------------------------------------------------------------------------------------------------------|
| -msdata=eabi  | Place small data in the small data areas following the EABI specification.                                      |
| -Wall         | Generate all possible compiler warnings.                                                                        |
| -Wl           | Pass a parameter to the linker, e.g. -Wl,-T, mapfile tells the linker to use the file mapfile as linker script. |
| -lc           | Use the C library during linking.                                                                               |

## Binutils

**Binutils** is a collection of binary utilities. It includes the portable GNU assembler (**as**), the archive utility for creating, modifying and extracting from libraries (**ar**), the GNU linker (**ld**), and the utility to display information from object files and executables (**objdump**).

## AS

The GNU assembler, **as**, processes assembly source code into object code. Normally, **as** is not executed manually but is invoked by GCC during compilation.

**Note:** Consult additional documentation about **as** for stand-alone use.

## AR

The GNU archiver, **ar**, is a utility to create, modify, and extract from libraries. A library consists of a set of object files combined into one single file.

Example 1: Create the library **hello.a** consisting of all object files:

```
$ powerpc-eabi-ar -r hello.a boot.o eabi.o crt0.o hello.o uart.o
```

Example 2: Extract the object files **boot.o** and **crt0.o** from the library **hello.a**.

```
$ powerpc-eabi-ar -x hello.a boot.o crt0.o
```

## LD

The GNU linker, **ld**, links object files and libraries into an executable. Normally, **ld** is not executed manually but is invoked by GCC.

**Note:** Consult additional documentation about **ld** for stand-alone use.

Example 1: Use **GCC** to link a set of object files (**\*.o**) into the executable **hello.elf**. Tell the linker to use the specified mapfile as linker script.

```
$ powerpc-eabi-gcc -Wl,-T,mapfile -o hello.elf *.o
```

Example 2: Use **GCC** to link a set of object files (**\*.o**), the board support package (**bsp.a**) and the C library into the executable **hello.elf**. Tell the linker to use the specified mapfile as linker script.

```
$ powerpc-eabi-gcc -Wl,-T,mapfile -o hello.elf *.o -lc bsp.a
```

The map file (or linker script) tells the linker how to combine object files and where to map them in the memory space. Below is a fairly complete example of a map file:

```
1. STACKSIZE = 0x400;
2. MEMORY
3. {
4. bram :ORIGIN = 0xfffffe000, LENGTH = 0x2000-4
5. boot :ORIGIN = 0xfffffffffc, LENGTH = 4
6. }
7.
8. ENTRY(_boot)
9.
```

```

10. SECTIONS
11. {
12. .text :{ *(.text) } > bram
13. .data :{ *(.data) } > bram
14. .stack :
15. {
16. . = ALIGN(8);
17. . = . + STACKSIZE;
18. } > bram
19. __stack = ADDR(.stack) + SIZEOF(.stack);
20. .boot :{ *(.boot) } > boot
21. }

```

In line 1, a constant of 1 KB for the stack size is defined. Lines 2 to 6 specify the memory map. In this example, the memory consists of two regions. The first region is named **bram**, starts at address 0xFFFFE000 and has a size of 8188 bytes. The second region is named **boot**, starts at address 0xFFFFF000 (the location of the reset vector on the PPC405) and has a size of 4 bytes. In line 8, the linker is told that the entry point to the program is named **\_boot**. This name must exist in one of the object files (e.g. in **boot.o**). The lines 10 to 21 tell the linker how the different sections of the object files have to be mapped into memory. In line 12, the code is filled into the memory region named **bram**, and in line 13 the same is done for the data. In lines 14 to 19, space for the stack is reserved. First, in line 16, the stack is aligned to an 8-byte boundary. This is done by aligning the dot (location pointer) to 8 byte. In line 17, the dot is increased by the size of the stack. In line 18, the stack is mapped into the memory region named **bram**. Finally, in line 19, a constant is defined that points to the top of the stack. This constant can be accessed from the initialization code (e.g. **crt0.o**) to set up the stack pointer register. In line 20, the section containing the boot code is mapped to the memory region named boot.

## OBJDUMP

GNU **objdump** is a utility used to display information from object files and executables. It allows the user to inspect file headers, to browse the contents of an object file or an executable, and to disassemble the object code. If **objdump** is used on an executable, the mapping of the sections into memory can be inspected.

Example 1: Display the header information of the object file **hello.o** and the executable **hello.elf**.

```

$ powerpc-eabi-objdump -x hello.o
$ powerpc-eabi-objdump -x hello.elf

```

Example 2: Disassemble the executable **hello.elf**.

```

$ powerpc-eabi-objdump -d hello.elf

```

Example 3: Show all information about the object file **hello.o**

```

$ powerpc-eabi-objdump -s hello.o

```

## GDB

GDB is a source-level debugger for C, C++, Assembly, and Fortran. Consult additional documentation for more information about **gdb**.

## Cross-Compilation and the powerpc-eabi Target

GNU software is normally distributed as source code. For your convenience, **GCC**, **binutils**, the **newlib** C library, and **GDB** have been cross-compiled from the source code for a powerpc-eabi target and are ready to use.

The powerpc-eabi target follows the EABI specification. The programmer of an application must be aware of this in a few points:

- Small data sections: The compiler references small data relative to two registers.



- The register R13 has to point to the readable and writeable **.sdata** and **.sbss** data sections.
- The register R2 has to point to the read-only **.sdata2** data section.
- **\_\_eabi**: The compiler inserts a call to **\_\_eabi** into the **main()** function. This is the place where registers R2 and R13 can be set up.
- **Stack**: The stack has to be set up on an 8-byte boundary.

More information on the EABI standard can be found in the following two documents:

[http://www.chips.ibm.com/products/powerpc/library/eabi\\_app.pdf](http://www.chips.ibm.com/products/powerpc/library/eabi_app.pdf)

<http://www.esofta.com/pdfs/ppceabi.pdf>

## Building GNU Software

This section gives detailed instructions on how to build the **binutils**, **GCC**, the **newlib** C library, and **GDB** in a user's home directory.

Assumptions:

- The shell is **sh**
- The environment variable **HOME** exists and points to the user's home directory.
- The directory **\${HOME}/gnu** exists
- The directory **\${HOME}/gnu/build** exists

### Extract Sources

```
$ cd ${HOME}/gnu/build/
$ bzip2 -cd binutils-010116.tar.bz2 | tar xvf -
$ bzip2 -cd gdb+dejagnu.tar.bz2 | tar xvf -
$ gtar xvzf gcc-2.95.2-6-src.tar.gz
$ gtar xvzf newlib-1.9.0.tgz
```

### Define "Build Target" and "Build Install" Directories

```
$ target=powerpc-eabi
$ prefix=${HOME}/gnu
$ i=${prefix}/bin
```

### Create Build and Install Directories

```
$ mkdir ../powerpc-eabi build-binutils-010116 build-gcc-2.95.2-6 build-
newlib-1.9.0 build-gdb+dejagnu
```

### Copy "Target Include" Files (for Library) to Installation Directory

```
$ cd newlib-1.9.0/
$ cp -d -p -r -v newlib/libc/include ${prefix}/${target}
```

### Configure and Build **binutils**

```
$ cd ../build-binutils-010116/
$../binutils-010116/configure --target=$target --prefix=$prefix -v
$ gmake all install
```

### Configure and Build **GCC** (Part 1)

```
$ cd ../build-gcc-2.95.2-6/
$../gcc-2.95.2-6/configure --target=$target --prefix=$prefix -v
Note: The next command will complete with errors due to missing target libraries.
$ gmake all install CC_FOR_TARGET=$i/${target}-gcc
AS_FOR_TARGET=$i/${target}-as LD_FOR_TARGET=$i/${target}-ld
```

```
AR_FOR_TARGET=$i/${target}-ar RANLIB_FOR_TARGET=$i/${target}-ranlib
NM_FOR_TARGET=$i/${target}-nm
$ make install-gcc CC_FOR_TARGET=$i/${target}-gcc
AS_FOR_TARGET=$i/${target}-as LD_FOR_TARGET=$i/${target}-ld
AR_FOR_TARGET=$i/${target}-ar RANLIB_FOR_TARGET=$i/${target}-ranlib
NM_FOR_TARGET=$i/${target}-nm
```

## Configure and Build **newlib**

```
$ cd ../build-newlib-1.9.0/
$../newlib-1.9.0/configure --target=$target --prefix=$prefix -v
$ make all install CC_FOR_TARGET=$i/${target}-gcc
AS_FOR_TARGET=$i/${target}-as LD_FOR_TARGET=$i/${target}-ld
AR_FOR_TARGET=$i/${target}-ar RANLIB_FOR_TARGET=$i/${target}-ranlib
NM_FOR_TARGET=$i/${target}-nm
```

## Configure and Build **GCC (Part 2)**

```
$ cd ../build-gcc-2.95.2-6/
```

**Note:** The next command may complete with errors for some unknown reason.

```
$ make all install CC_FOR_TARGET=$i/${target}-gcc
AS_FOR_TARGET=$i/${target}-as LD_FOR_TARGET=$i/${target}-ld
AR_FOR_TARGET=$i/${target}-ar RANLIB_FOR_TARGET=$i/${target}-ranlib
NM_FOR_TARGET=$i/${target}-nm
```

**Note:** The next command is only necessary if the previous command did not complete without error.

```
$ make install CC_FOR_TARGET=$i/${target}-gcc
AS_FOR_TARGET=$i/${target}-as LD_FOR_TARGET=$i/${target}-ld
AR_FOR_TARGET=$i/${target}-ar RANLIB_FOR_TARGET=$i/${target}-ranlib
NM_FOR_TARGET=$i/${target}-nm
```

## Configure and build **GDB**

```
$ cd ../build-gdb+dejagnum/
$../gdb+dejagnum-20010114/configure --target=$target --prefix=$prefix -v
$ make all install CC_FOR_TARGET=$i/${target}-gcc
AS_FOR_TARGET=$i/${target}-as LD_FOR_TARGET=$i/${target}-ld
AR_FOR_TARGET=$i/${target}-ar RANLIB_FOR_TARGET=$i/${target}-ranlib
NM_FOR_TARGET=$i/${target}-nm
```



# Volume 6: Reference Systems

## *Virtex-II Pro™ Platform FPGA Developer's Kit*

March 2002 Release





The Xilinx logo shown above is a registered trademark of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

"Xilinx" and the Xilinx logo are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, XC5210 are registered Trademarks of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Bencher, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Bencher, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, Nano-Blaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, Rocket I/O, SelectI/O, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM      IBM Logo      PowerPC      PowerPC Logo      Blue Logic      CoreConnect      CodePack

All other trademarks are the property of their respective owners.

Xilinx does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 2002 Xilinx, Inc. All Rights Reserved.

# *Introduction to Reference Systems*

---

## Summary

This chapter briefly describes each of the reference systems provided with the Virtex-II Pro™ Developer's Kit. It offers an overview of each system, discusses the various elements that the system contains, and illustrates what can be learned from each. The reference systems are intended to familiarize users with the Virtex-II Pro product and its features. While they do not contain all elements a user system might require, they provide a foundation for those who are just learning about the embedded PowerPC processor in Virtex-II Pro FPGAs.

## Introduction

The Virtex-II Pro Developer's Kit provides three complete reference systems; designs that contain a combination of known working hardware and software elements that, together, create an entire system. Each example reference system is designed to demonstrate a different element of the IBM® PowerPC® and/or Rocket I/O™ Multi-Gigabit Transceiver. Each system also offers an opportunity to learn about the Device Control Register (DCR) bus.

The three reference systems are:

- Simple PPC405 Reference System
  - Demonstrates the basics of the Processor Local Bus (PLB)
- PPC405 Packet Processing Engine (PPE) Reference System Using Rocket I/O Transceivers
  - Demonstrates the basics of the PPC405 On-Chip Memory (OCM)
  - Demonstrates the basics of the Rocket I/O Transceiver
- Embedded PPC405 Reference System
  - Demonstrates the On-Chip Peripheral Bus (OPB)
  - Demonstrates the combined elements of the PLB and OCM interfaces from the other systems

## Reference System Information

This section is an overview of the features and uses of each reference system. Although the information contained in each reference system chapter is not exhaustive, it covers the basic requirements to effectively use PPC405 and Rocket I/O technology. All of the reference systems provided in the Developer's Kit have instructions on how to simulate, synthesize, and run the designs through the Xilinx Implementation Tools (e.g. ISE 4.2i) for the Virtex-II Pro family.

Each reference system chapter contains sections about:

- Hardware used in the system

- HDL file organization
- Simulation and verification
  - Using SWIFT
  - Using Bus Functional Model (BFM)
- Synthesis and implementation
- Software applications that interoperate with the system
- How to run the software applications
- Directory structure of each system

## Simple PPC405 Reference System

The Simple PPC405 Reference System represents one of the simplest systems that can be built with Virtex-II Pro. It is designed to familiarize the user with the design flow, to teach the basic PLB requirements, and to instruct the user in running Bus Functional Model (BFM) and SWIFT model simulations.

The Simple PPC405 Reference System consists of a PPC405 CPU that uses the 64-bit version of the Processor Local Bus (PLB) to connect to its other elements. The system contains a BRAM memory interface and a 16450 UART peripheral, which are slaves on the PLB. A PLB arbiter in the system permits the CPU (the master) to talk to the memory and/or peripherals.

The Simple PPC405 Reference System offers a quick glance at a basic system by demonstrating basic PLB transactions and allowing users to run simulations of actual PowerPC code executing in the PPC405 SWIFT model. While this design could have real-world applications, it is intended mainly to illustrate how the PLB operates and to familiarize users with the design flow.

The user can choose between a SWIFT model simulation of the PowerPC and a Bus Functional Model simulation by editing a single file. The SWIFT model simulation permits the execution of actual PPC405 code. In contrast, the Bus Functional Model simulation emulates the PLB and Device Control Register (DCR) bus transaction behavior of the PPC405 by executing commands listed in a Bus Functional Language (BFL) file.

## PPC405 PPE Reference System Using Rocket I/O Transceivers

The PPC405 Packet Processing Engine (PPE) Reference System Using Rocket I/O Transceivers builds on the Simple PPC405 Reference System by exposing the user to three additional elements: the Rocket I/O Transceiver, the On-Chip Memory buses of the PowerPC, and the Device Control Register Bus. It contains three packet processors in a ring topology to demonstrate how multiple, concurrent CPUs can be used to form a larger system. Additionally, by illustrating some typical packet headers and data containers, the system shows how packet processing is accomplished.

The chapter provides greater detail about the processor block, the On-Chip Memory interfaces, the DCR bus, and associated registers in the design. It also offers a more thorough description of the system software by showing an example of booting the PPC405 and running housekeeping operations.

The provided testbench instantiates three of the packet processors to make a complete packet processing system. To run synthesis or place and route, only one instance of the packet processor is used.

## Embedded PPC405 Reference System

The Embedded Reference System, the most complex of the three reference systems, is an example of a completely embedded computer. It provides a wide variety of memory interfaces on three differing buses, as well as various peripherals such as IIC, General

Purpose I/O (GPIO), UARTs, Configuration ROM (CROM), PCI interface, Character LCD controller, and a memory-mapped DCR bus bridge.

The Embedded PPC405 Reference System combines the elements of a typical embedded system by taking advantage of the architectural features of the PPC405, such as separated Instruction-Side and Data-Side OCM interfaces. It illustrates the use of a typical segmented bus design where the higher-speed elements (such as memory) are differentiated from the lower-speed elements (such as GPIO) through the use of bus arbiters and bridges.

This system provides an excellent example of the various elements a user might use to run a Real-Time Operating System (RTOS). The example software provided with this reference system is currently designed to operate only as a stand-alone application.

The Embedded PPC405 Reference System provides additional study of the PLB, OCM, and DCR buses. In addition, it affords the opportunity to see how OPB-based devices are used in a system. Step-by-step instructions are provided to help the user through the design flow and to target a Virtex-II Pro device.

Users can modify the Embedded PPC405 Reference System to add and subtract peripherals, as well as to change the software for their own custom-designed systems. These designs can be fully simulated, synthesized, and run through PAR to produce a bitstream for Virtex-II Pro devices.

## Further Reading

The Virtex-II Pro™ Developer's Kit contains a wealth of valuable information that will assist you in your design efforts. The documentation contained within the eight volumes is organized to assist you in quickly finding relevant materials. To obtain the most recent revision of this documentation, please see

[http://support.xilinx.com/xlnx/xil\\_tt\\_product.jsp?sProduct=Virtex-II+Pro](http://support.xilinx.com/xlnx/xil_tt_product.jsp?sProduct=Virtex-II+Pro).

## Documentation Provided by Xilinx

*Virtex-II Pro™ Platform FPGA Handbook*

*Virtex-II Pro™ Platform FPGA Developer's Kit Documentation:*

*Volume 1, Advance Product Specification*

*Volume 2, PowerPC® 405 Processor*

*Volume 3, Rocket I/O™ Transceiver User Guide*

*Volume 4, Design Flow*

*Volume 5, Software Development Tools*

*Volume 6, Reference Systems*

*Volume 7, Hardware IP Specifications*

*Volume 8, Software IP and Applications*

## IBM® CoreConnect™ Documentation

The Virtex-II Pro Developer's Kit integrates seamlessly with the IBM CoreConnect Toolkit. This toolkit is not included with the Developer's Kit, but is required if bus functional simulation is desired. The toolkit provides a number of features which enhance design productivity and allow you to get the most from the Developer's Kit. To obtain the toolkit, you must be a licensee of the IBM CoreConnect Bus Architecture. Licensing CoreConnect provides access to a wealth of documentation, Bus Functional Models, Hardware IP, and the toolkit.

Xilinx provides a Web-based licensing mechanism that allows you to obtain the CoreConnect toolkit from our website. To license CoreConnect, use an Internet browser to access [http://www.xilinx.com/ipcenter/processor\\_central/register\\_coreconnect.htm](http://www.xilinx.com/ipcenter/processor_central/register_coreconnect.htm).



Once your request has been approved (typically within 24 hours), you will receive an e-mail granting access to a protected website. You may then download the toolkit.

If you prefer, you can also license CoreConnect directly from IBM.

If you would like further information on CoreConnect Bus Architecture, please see IBM's CoreConnect website at <http://www.ibm.com/chips/products/coreconnect>.

Once you have licensed the CoreConnect toolkit, and installed it with the Developer's Kit, the following documents will be available to you in the following locations:

## IBM CoreConnect Bus Architecture Specifications

*IBM CoreConnect Processor Local Bus (PLB) Architecture Specification*  
see **\$CORECONNECT/published/corecon/64bitPlbBus.pdf**

*IBM CoreConnect On-chip Peripheral Bus (OPB) Architecture Specification*  
see **\$CORECONNECT/published/corecon/OpbBus.pdf**

*IBM CoreConnect Device Control Register (DCR) Bus Architecture Specification*  
see **\$CORECONNECT/published/corecon/DcrBus.pdf**

## IBM CoreConnect Toolkit Documentation

*PLB Bus Functional Model Toolkit - User's Manual*  
see **\$CORECONNECT/published/corecon/PlbToolkit.pdf**

*OPB Bus Functional Model Toolkit - User's Manual*  
see **\$CORECONNECT/published/corecon/OpbToolkit.pdf**

*DCR Bus Functional Model Toolkit - User's Manual*  
see **\$CORECONNECT/published/corecon/DcrToolkit.pdf**

*CoreConnect Test Generator - User's Manual*  
see **\$CORECONNECT/published/corecon/ctg.pdf**

**Note:** \$CORECONNECT is an environment variable that is created when installing the Developer's Kit or CoreConnect Toolkit.

## Software Development Documentation

There are many sources of documentation available for those who wish to learn more about Software Development. It is recommended that a web search be conducted using a favorite search engine for keywords such as "PowerPC+Software+Development". Alternatively, a technical bookstore should be able to provide many valuable resources. The books listed below are a very small fraction of those available.

### Books About Programming in C

There are many good books about the C and C++ programming languages. A few of these are listed below:

#### Books about C

Kernigham Brian, Ritchie Dennis. 1988. *The C Programming Language*. Prentice Hall

Wang Paul. 1992. *An Introduction to ANSI C on Unix*. Wadsworth.

Kumar Ram, Agrawal Rakesh. 1992. *Programming in ANSI C*. West Publishing Company.

#### Books about C++

Stroustrup Bjarne. 1991. *The C++ Programming Language*. Addison-Wesley.

Lafore Robert. 1995. *Object-Oriented Programming in C++*. Waite Group Press.

## Online Documents About Programming in C

The Internet offers plenty of documentation about how to program in C and C++. One of the best approaches to finding documentation online is to use a search engine (such as Google, <http://www.google.com>) and search on "introduction to programming in C."



## Simple PPC405 Reference System

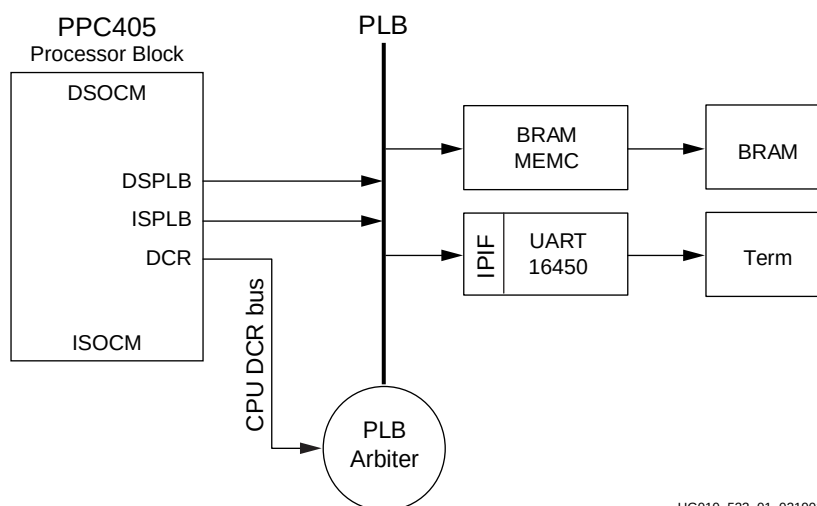
### Introduction

The Simple PPC405 Reference System is an example of a small Virtex-II Pro™ based system. An IBM CoreConnect™ infrastructure connects the CPU to memory and a peripheral using the Processor Local Bus (PLB) and Device Control Register (DCR) buses to build a system. To keep the size of the design small, an On-Chip Peripheral Bus (OPB) is not used. This document describes the contents of the reference design and provides information about how the design is organized and implemented. It also discusses verification methods, including software-driven and bus model-driven simulations. A complete design cycle incorporating simulation, synthesis, and FPGA implementation is described. The information presented introduces many aspects of the Simple PPC405 Reference System, but the user should refer to additional specific documentation for more detailed information about the software, peripherals, interface protocols, and capabilities of the FPGA.

### Hardware

#### Overview

Figure 2-1 provides a high-level view of the hardware contents of the Simple PPC405 Reference System. This design demonstrates a system that uses PLB and DCR devices. The PLB connects BRAM memory and the UART peripheral to the CPU. Refer to the PLB and DCR CoreConnect Architecture Specifications for more information.



UG010\_523\_01\_021902

Figure 2-1: High-Level Hardware View of Simple PPC405 Reference System

## Processor Local Bus (PLB)

The PLB connects the CPU to high-performance devices such as memory controllers. The PLB protocol supports higher-bandwidth transactions and has a feature set that better supports memory operations from the CPU. Highlights of the PLB protocol include synchronous architecture, independent read/write data paths, and split transaction address/data buses. The reference design includes a 64-bit PLB infrastructure with 64-bit master and slave devices attached.

The PLB devices in the reference system include:

- PLB Masters
  - CPU Instruction-Side PLB Interface (Master ID = 0)
  - CPU Data-Side PLB Interface (Master ID = 1)
- PLB Slaves
  - 16450 UART
  - BRAM Controller
- PLB Arbiter
  - 8 Master, 64-bit Xilinx PLB Arbiter
- PLB Bus Logic
  - The specification for the PLB protocol requires that a *PLB Slave Bus* be created. This is accomplished by using some additional logic to OR together the outputs of the slave devices. This sample system supports up to nine PLB slave devices, but can easily be expanded to support additional slaves.

In general, all PLB devices are optimized around the Virtex-II Pro architecture and make use of pipelining to improve maximum clock frequencies and reduce logic utilization. Refer to the accompanying documentation for each device for more information about its design.

The UART does not require the high performance available over PLB, but is connected to the PLB to demonstrate a system without OPB. In smaller systems, the extra overhead incurred by using PLB interfaces for slower peripherals is outweighed by the benefit of not having to instantiate an OPB Bridge and OPB Arbiter. The UART is implemented by replacing the IPIF module in a normal OPB-based UART with a PLB-based IPIF. This illustrates how IPIF-based designs have greater flexibility to support different bus interfaces without much redesign.

## Device Control Registers (DCR)

DCR offers a very simple interface protocol and is used for accessing control and status registers in various devices. It allows for register access to various devices without loading down the high speed buses. The CPU contains a DCR Master interface that is accessed through special Move-To-DCR and Move-From-DCR instructions. The only DCR device in the reference system is the bus error address and status registers of the PLB Arbiter.

## Other Devices

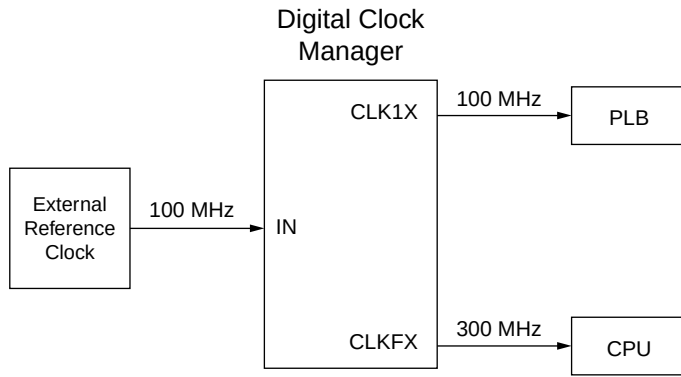
This design does not have an interrupt controller, so the UART interrupt output is fed into the CPU's non-critical interrupt input. The critical interrupt input on the CPU is tied off to logic 0.

## Clock/Reset Distribution

Figure 2-2 illustrates use of the Digital Clock Manager (DCM) for generating clocks in the design. A 100-MHz input reference clock is used to generate the main 100-MHz PLB clock. The CPU clock is multiplied up from the PLB clock to 300 MHz. Since each of these clocks

are generated from the same 100-MHz input, they are all phase-aligned to each other. This synchronous phase alignment is required by the CPU.

The CPU clock can run at any integer multiple of the PLB clock, up to the maximum CPU clock frequency. During reset, internal clock synchronizers in the CPU detect the phase alignment of the PLB and CPU clocks and adjust for it automatically.



UG010\_523\_02\_062501

**Figure 2-2: Clock Generation**

After a system reset or at FPGA startup, a debounce circuit holds the FPGA in reset until the DCM has locked onto its reference clock. Once the DCM is locked and the clocks remain stable for several cycles (may take several microseconds in simulation), the reset condition is released to allow the system logic to begin operating. (For example, the CPU will begin fetching instructions a few cycles after reset is released.) Since the reset net is a high-fanout signal, it may not be able to reach all the logic in the design within one clock cycle. User IP blocks should be designed to take into account the possible skew in the global reset and still start up properly. Alternatively, the global reset can be registered locally in each IP block to generate a synchronous reset signal.

The design implements the three levels of reset supported by the PPC405:

- Core reset
- Chip reset
- System reset

The core reset only affects the processor while the chip reset clears all the logic on the FPGA. The system reset is designed to reset the entire system including the FPGA and external devices connected to the FPGA. The CPU provides an internal special-purpose register that allows software to request that one of the three resets be performed.

**Figure 2-3** shows how the reset logic in the Simple PPC405 Reference System is implemented. The chip and core reset request lines from the CPU are driven to the corresponding reset nets through shift registers to ensure that minimum pulse width requirements are met. The system reset request line from the CPU drives an external pin as an open-drain signal (with resistive pullup) so that other external devices can sense or drive the system reset. Note that the system reset has the effect of resetting the DCMs in the FPGA, while the chip reset does not affect the DCMs. The reset logic in the Simple PPC405 Reference System is an example implementation of the PPC405 reset architecture. Designers should set the scope, boundaries, and effects of resets as appropriate to their designs.

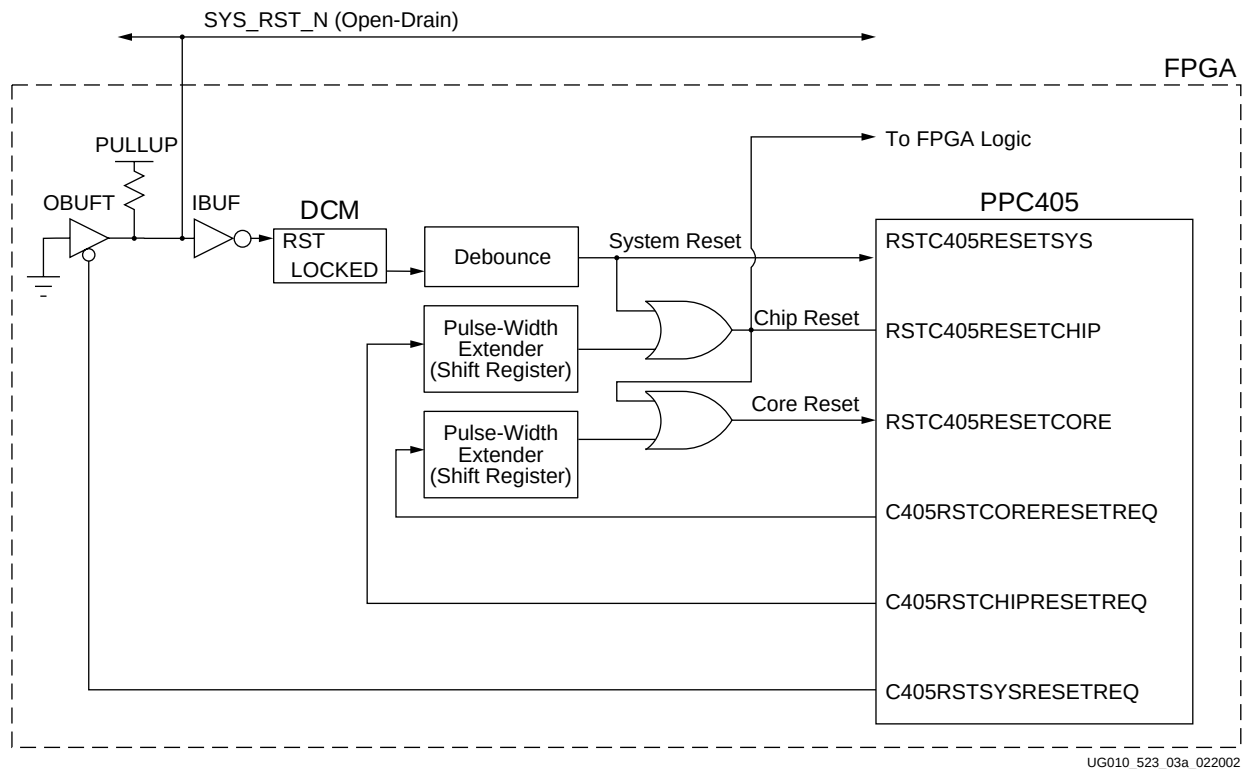


Figure 2-3: Simple Reference System Reset Logic

## HDL Organization

Figure 2-4 diagrams the organization of the higher-level HDL files that build up the system and testbench environment.

All of the peripherals, the memory controller, the CoreConnect infrastructure, and the other user logic are instantiated in the **ip\_wrapper.v** file. External I/O signals, clock/reset connections, and processor block connections are propagated up to the next level file **top.v**. The **top.v** file instantiates the clock/reset logic, processor block and **ip\_wrapper.v** file. The module ports of **top.v** represent the external I/O signals of the FPGA. Therefore, all files in the hierarchy from the **top.v** level and below make up the Virtex-II Pro FPGA design and are synthesized/routed into an FPGA.

For simulation, the file **testbench.v** instantiates the FPGA as the device under test and includes behavioral models for the FPGA to interact with. In addition to behavioral models for clock oscillators and external peripherals, the testbench also instantiates the CoreConnect bus monitors to observe the PLB and DCR buses for protocol violations. The testbench can also preload the BRAM memories in the system for purposes such as loading software for the CPU to execute. The **sim\_params.v** file is designed to be user modified to customize various simulation options. These options include message display options, maximum simulation time, and clock frequency. Users should edit this file to reflect their simulation preferences.

An additional HDL file called **global\_params.v** defines the global constants and parameters used throughout the reference design. This information may include memory map definitions, device configuration values, or any other information that is better organized in a global context.

Some of the testbench code accesses signals internal to the design, using hierarchy path names to reach into the design without changing any of the port interfaces. It is important that the design source files used for simulation match the source files for synthesis. Therefore, port interfaces should not be different, or else inconsistencies can result.

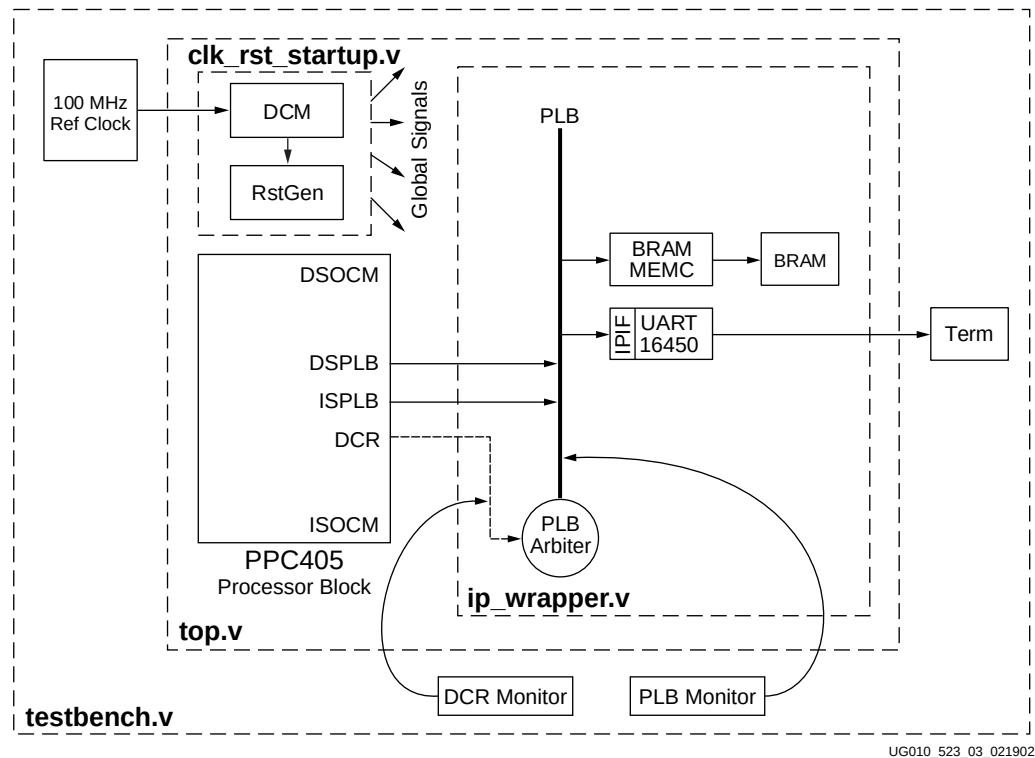


Figure 2-4: Organization of Higher-Level HDL Files

## Simulation and Verification

### SWIFT and BFM CPU Models

The Simple PPC405 Reference System demonstrates two different simulation methods to help verify designs utilizing the PPC405 CPU. One method is to utilize a full simulation model of the CPU based on the actual silicon. The second method is to employ Bus Functional Models (BFMs) to generate processor bus cycles from a command scripting language. The two methods offer different trade-offs between behavior in real hardware, ease of generating bus cycles, and the amount of real time required to simulate a given clock cycle.

A SWIFT Model of the CPU can be utilized to simulate the CPU executing software instructions. In this scenario, the executable binary images of the software are preloaded into memory, from which the CPU can boot up and run the code. Though this is a relatively slow way to exercise the design, it more accurately reflects the actual behavior of the system. The SWIFT Model is most useful for helping to bring up software and for correlating behavior in real hardware with simulation results. The reference design demonstrates the SWIFT Model simulation flow by allowing the user to write a C program compiled into an executable binary file. This executable (in .elf format) is then converted into BRAM initialization commands using a tool called Data2BRAM.

When a simulation begins and reset is released, the CPU SWIFT Model will fetch the instructions from BRAM (mapped to the boot vector) and begin running the program. The user can then observe the bus cycles generated by the CPU or any other signal in the



design. For debugging purposes, the values of the CPU's internal program counter, general-purpose registers, and special-purpose registers are available for waveform display during simulation. An option bit in `sim_params.v` can be set to cause the testbench to display a status message any time an internal CPU register changes.

Generating a desired sequence of bus operations from the CPU may require a lot of software setup or simulation time to produce. For early hardware bring-up or IP development, Bus Functional Models can be used to speed up simulation cycles and avoid having to write software. A model of the CPU is available in which two PLB master BFM's and one DCR BFM are instantiated to drive the CPU's PLB/DCR ports. These BFM's are provided in the CoreConnect toolkits and allow the user to generate bus operations by writing a script written in the Bus Functional Language (BFL). The Simple PPC405 Reference System provides a sample BFL script that exercises many of the peripherals in the system. Refer to the CoreConnect toolkit documentation for more information.

Since the CPU SWIFT Model and BFM model both have the same set of port interfaces, the user can switch between the two simulation methods by compiling the appropriate set of files, without having to modify the system's design source files. The user may, however, need to modify the testbench to take into account which model is being used.

## Behavioral Models

The Simple PPC405 Reference System includes some behavioral models to help exercise some of the devices and peripherals in the FPGA. A terminal interface is connected to the UART to send and receive serial data. The terminal allows a user to interact with the simulation in real time. Characters sent out by the UART are displayed on a terminal, while characters typed into the terminal program are serialized and sent to the UART. A simple file I/O mechanism passes data between the hardware simulator and the terminal program.

## Synthesis and Implementation

The Simple PPC405 Reference System can be synthesized and placed/routed into a Virtex-II Pro FPGA. A basic set of timing constraints for the design is provided to allow the design to go through place and route. After a successful place and route, it is possible to run a simulation of the design using a back-annotated timing model of the FPGA with a SWIFT or BFM version of the processor block.

## Design Flow Environment

A flow engine provides an environment to help manage the design flow for the Simple PPC405 Reference System. This engine uses the utility program, called **make**, and a set of PERL scripts to allow the user to perform tasks such as running simulations, synthesizing a design, or implementing it on an FPGA with a simple set of commands. The design flow tool is implemented with a generic architecture to allow it to be adapted for use on a variety of different designs. For more information, refer to *Flow Documentation*.

# File Listings

The files and directories specific to the Simple PPC405 Reference System are located below the **\$V2PRO/platforms/simple** directory and are shown in **Figure 2-5** and listed in the tables that follow. Source files for the IP modules and software applications are kept in the **\$V2PRO/source** directory. The **\$V2PRO/source** directory generally contains files that can be shared by different designs. The **File Structure for Simple PPC405 Reference System** section diagrams the full file directory structure for this design. Refer to the appropriate instructions for installation and setup information. Note that the following tables only list files in the **\$V2PRO/platforms/simple** directory that are present after installation. After running simulation, synthesis, or place and route, additional files may be created. Directory path names are shown separated by the "/" character as is the UNIX convention. For Windows, the "\" should be used to separate directory paths.

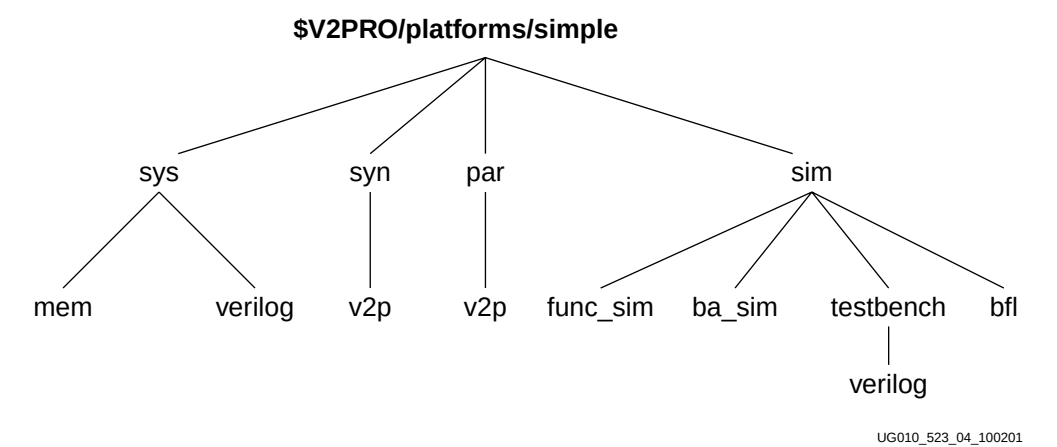


Figure 2-5: Map of Directories Below \$V2PRO/platforms/simple

Table 2-1: Files in the \$V2PRO/platforms/simple Directory

| File     | Description                                                                                          |
|----------|------------------------------------------------------------------------------------------------------|
| Makefile | Redirection file to allow the centralized design flow scripts to be invoked from different locations |
| flow.cfg | File used to configure the design flow scripts                                                       |

Table 2-2: Files in the \$V2PRO/platforms/simple/sys Directory

| File          | Description                                                       |
|---------------|-------------------------------------------------------------------|
| bram_init.bmm | Data2BRAM configuration file describing the system memory devices |

Table 2-3: Files in the \$V2PRO/platforms/simple/sys/verilog Directory

| File              | Description                                                              |
|-------------------|--------------------------------------------------------------------------|
| src.lst           | List of source files for the design                                      |
| bram_block.v      | Wrapper holding PLB connected block RAMs                                 |
| clk_rst_startup.v | Clock, reset, startup logic for the design                               |
| global_params.v   | Centralized location for the global parameters used within the design    |
| ip_wrapper.v      | Wrapper file around all the IP in the design. Contains PLB and DCR logic |
| plb_bus_logic.v   | OR logic for PLB to connect PLB slaves to the PLB arbiter                |
| top.v             | Top-level file for Virtex-II Pro FPGA                                    |

Table 2-4: Files in the \$V2PRO/platforms/simple/sim/func\_sim Directory

| File        | Description                                                                              |
|-------------|------------------------------------------------------------------------------------------|
| Uart1.input | Character to be sent to UART 1 from the terminal (set first line to "@0 0" for no input) |
| wave.do     | MTI command to display a sample set of interesting signals for functional simulation     |

Table 2-5: Files in the \$V2PRO/platforms/simple/sim/bfl Directory

| File                    | Description                                                               |
|-------------------------|---------------------------------------------------------------------------|
| test_simple_ref_des.bfl | Sample BFL script to exercise the devices in the system during simulation |

Table 2-6: Files in the \$V2PRO/platforms/simple/sim/ba\_sim Directory

| File        | Description                                                                                     |
|-------------|-------------------------------------------------------------------------------------------------|
| Uart1.input | Character to be sent to UART 1 from the terminal (set first line to "@0 0" for no input)        |
| wave.do     | MTI command to display a sample set of interesting signals for back-annotated timing simulation |

Table 2-7: Files in the \$V2PRO/platforms/simple/sim/testbench/verilog Directory

| File         | Description                                                                         |
|--------------|-------------------------------------------------------------------------------------|
| sim_params.v | Centralized location for parameters and settings that affect simulation             |
| testbench.v  | Testbench source file                                                               |
| view_reg.v   | Behavioral code to display internal CPU register contents whenever they are updated |

Table 2-8: Files in the \$V2PRO/platforms/simple/par Directory

| File    | Description                                             |
|---------|---------------------------------------------------------|
| top.ucf | User constraints file used by FPGA implementation tools |

Table 2-9: Files in the \$V2PRO/platforms/simple/syn Directory

| File         | Description                                 |
|--------------|---------------------------------------------|
| synplify.opt | Optional synthesis project file information |

## Instructions for Running Functional Simulations

The Simple PPC405 Reference System comes with SWIFT and BFM based simulation examples. This section describes the necessary steps for running these simulations using functional timing with the original source code. It assumes the Virtex-II Pro Design Kit is properly installed and that the **\$V2PRO/platforms/simple/flow.cfg** file is properly edited to correctly call the user's simulation tools. The user may also modify the **sim\_params.v** file to customize various simulation parameters.

### SWIFT Simulations

The steps to run a SWIFT simulation are described below:

1. Go to the main directory of the design and edit the **flow.cfg** to select the software application you wish to run:  

```
$ cd $V2PRO/platforms/simple
$ <text editor> flow.cfg
```

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

The *<text editor>* can be a program like Emacs, vi, or WordPad. In the **flow.cfg** file, set the PPC\_MODEL to "swift". After editing the **flow.cfg** file, save it.

2. Invoke the design flow engine to start the simulation:  

```
$ make func_sim
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Simple** → **Make Functional Simulation**)

The flow engine begins by compiling and linking the application software (as necessary) to generate a binary executable program file (in **.elf** format). The **.elf** file is copied locally, then **data2bram** is invoked to generate the necessary BRAM

initialization command. The simulator tool is then launched along with a terminal for the UART.

**Note:** For ModelTech (MTI) ModelSim, the scripts copy a corresponding NT or Solaris version of the **modelsim.ini** configuration file into the local directory. This file can be found in the following directory:

**\$V2PRO/tools/\$V2PRO\_HOST\_OSTYPE/xilinx/data/modelsim.ini**

3. Run the simulation:

In the MTI window, type **run -all**. You can interact with the simulation using the terminal program. On the terminal, the words "Hello world" followed by a carriage return will appear as the program executes in the simulation. You can also type into the terminal (after **Hello world** is displayed) and have the characters echoed back after some delay. Keep in mind that the program is running in simulation, so it may take many seconds of real time for the characters to get transmitted and received. Pressing the **Esc** (escape) key will exit the terminal and stop the simulation after a few seconds.

For Verilog XL, the simulation begins immediately. You can interact with the terminal for the **hello\_uart** and **hello** programs as described above. If the " 'define VIRSIM " command in **sim\_params.v** is set, the waveforms for all the signals in the design are stored for the **virsim** program to display. You may need to edit the testbench to invoke your own Verilog XL waveform capture and display program.

It is normal to see some warnings from the PLB monitor or behavioral memory models during reset, but the PLB/DCR monitors should not report any protocol errors during simulation (warnings and notes may occur depending on the circumstances).

## BFM Simulations

The steps to run a BFM simulation are described below:

1. To run a BFM simulation, edit the **flow.cfg** file to change the processor block model to BFM:

```
$ cd $V2PRO/platforms/simple
$ <text editor> flow.cfg
```

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

Change the line "PPC\_MODEL = swift" to "PPC\_MODEL=bfm". After editing the **flow.cfg** file, save it.

2. Invoke the design flow engine to start the simulation:
 

```
$ make func_sim
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Simple** → **Make Functional Simulation**)

This script invokes the BFC from the IBM CoreConnect toolkit on the **.bfl** script in **\$V2PRO/platforms/simple/sim/bfl**. This generates a Verilog command file that is "compiled in" by the Verilog simulator to control the processor block PLB and DCR BFM masters. The simulator tool is then launched along with terminals for the UARTs and LCD.

3. Run the simulation:

In the MTI window, type **run -all**. For Verilog XL, the simulation begins immediately. The BFL script performs a set of memory write/read tests to all memory devices in the system. It then sends out "Hello" over UART1 after about 9  $\mu$ s.

The simulation stops when the BFL script is finished executing or when an error occurs. If the simulation completes successfully, the simulator displays the following message after about 32  $\mu$ s:

**Synch 31 received... Simulation Completed**

If an error is detected, either in the form of a protocol violation reported by a bus monitor or a read comparison error, the simulation stops and an error message is displayed.

It is a useful exercise to view the simulator's waveform display and correlate the commands in the **.bfl** script with the bus transaction waveforms over PLB and DCR.

## Instructions for Synthesizing the Design

The Simple PPC405 Reference System can be synthesized into FPGA primitive components. This section describes the necessary steps for synthesizing the design. It assumes the Virtex-II Pro Design Kit is properly installed and that the **\$V2PRO/platforms/simple/flow.cfg** file has been properly edited to correctly call the user's synthesis tool.

1. Invoke the design flow engine to start the synthesis:  
\$ cd \$V2PRO/platforms/simple  
\$ make synth

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Simple** → **Make Synthesis**.)

This script creates a project file for the synthesis tool and invokes it. For Synplify, the GUI is launched.

2. If using Synplify, Click the Run button to start the synthesis process.  
**Note:** Any changes saved to the Synplify project file (**top.prj**) will be overwritten the next time "make synth" is called. You should transfer any changes in the project file to the **synplify.opt** file.
3. After completion, close the Synplify GUI.

You can edit the **\$V2PRO/platforms/simple/syn/synplify.opt** file to add or modify any of the synthesis options. You can also edit the **\$V2PRO/platforms/simple/flow.cfg** file to change the target part. (The default part is xc2vp7fg456-7).

## Instructions for FPGA Implementation

After synthesis, the Simple PPC405 Reference System can be targeted into an FPGA using the Xilinx implementation tools. This section describes the necessary steps for implementing the design. It assumes the Xilinx ISE Software is properly installed.

1. Invoke the design flow engine to start the implementation process:  
\$ cd \$V2PRO/platforms/simple  
\$ make fpga

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Simple** → **Make FPGA**)

The script then copies the synthesis output files to the **\$V2PRO/platforms/simple/par/v2p** directory and runs through the implementation tools (**ngdbuild**, **map**, **par**, **trce**). Any necessary netlists (i.e., **.edf**, **.edn**, **.ngc**, **.ngo**) for devices in the system are also copied for **ngdbuild** to process. The timing constraints used by the implementation tools are copied from **\$V2PRO/platforms/simple/par/top.ucf** along with any BRAM initialization commands passed from **data2bram**.

2. With a successfully routed design, you can generate a bitstream:  
\$ cd \$V2PRO/platforms/simple  
\$ make bit

This generates a bitstream for the targeted Virtex-II Pro device.

## Instructions for Back-Annotated Timing Simulation

A placed and routed FPGA design can be simulated with full timing back-annotation. This section describes the necessary steps for simulating the back-annotated design using BFM or SWIFT models of the processor block. In general, the steps are identical to those of a functional simulation except that the "make ba\_sim" command is used.

### SWIFT Simulations

The steps to run a SWIFT simulation are described below:

1. Go to the main directory of the design and edit the **flow.cfg** to select the software application you wish to run:  
\$ cd \$V2PRO/platforms/simple  
\$ <text editor> flow.cfg

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

Make sure that PPC\_MODEL = swift. After editing the **flow.cfg** file, save it.

2. Invoke the design flow engine to start the simulation:  
\$ make ba\_sim

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Simple** → **Make Timing Simulation**)

The flow engine compiles/links the software, runs **data2bram**, and invokes the Xilinx tools for back-annotation (**ngdanno**, **ngd2ver**). It then launches the simulator tool along with a terminal for the UART.

3. Run the simulation:

In the MTI window, type **run -all**. For VerilogXL, the simulation begins immediately.

**Troubleshooting Hint:** Slow down the clocks or edit the back annotated simulation file (**\$V2PRO/platforms/simple/sim/ba\_sim/top.v**) and comment out the **\$sdf\_annotate** command. This causes the simulation to ignore timing, but allows a functional test on the routed FPGA design.

### BFM Simulations

The steps to run a BFM simulation are described below:

1. To run a BFM simulation, edit the **flow.cfg** file to change the processor block model to BFM:  
\$ cd \$V2PRO/platforms/simple  
\$ <text editor> flow.cfg

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

Change the line "PPC\_MODEL = swift" to "PPC\_MODEL=bfm". After editing the **flow.cfg** file, save it.

2. Invoke the design flow engine to start the simulation:  
\$ make ba\_sim

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Simple** → **Make Timing Simulation**)



This script runs the Xilinx tools for back-annotation (**ngdanno**, **ngd2ver**) and then invokes the BFC on the **.bfl** test script. It then launches the simulator tool along with a terminal for the UART.

3. Run the simulation:

In the MTI window, type **run -all**. For Verilog XL, the simulation begins immediately. The test ends successfully when the following is returned:

Synch 31 received... Simulation Completed

## Memory Map

This section diagrams the system memory map for the Simple PPC405 Reference System. The memory map shown in **Table 2-10** reflects the default location of the system devices as they are set in the **global\_params.v** file.

*Table 2-10: PLB Device Memory Map*

| Device                    | Address Boundaries |          | Size  |
|---------------------------|--------------------|----------|-------|
|                           | Upper              | Lower    |       |
| PLB Slv1 - UART1 (16450)  | A000101F           | A0001000 | 32 B  |
| PLB Slv0 - PLB BRAM Space | FFFFFFFF           | FFFF8000 | 32 KB |

## Extending or Modifying the Design

The Simple PPC405 Reference System is a good starting point from which a user can add, remove, or modify components in the system. Since the IP in the design is attached to the PLB/DCR CoreConnect infrastructure, adding or removing devices is a fairly straightforward process that is explained in the CoreConnect documentation. Below is an overview for making various changes to the system.

### Adding or Removing PLB Devices

To remove a PLB slave:

- Delete the module instantiation for that piece of IP
- Delete all corresponding external I/O ports
- Tie all PLB slave output signals to zero
- Remove any global parameters used exclusively by that slave (such as the base address)

Since the two masters in the reference design come from the CPU, they may not be removable, but in general, a PLB master is removed in the same way as a PLB slave. The device, its external I/O, and associated global parameters are removed and the PLB signals it normally drives must be tied off to zero.

To add a PLB slave:

- Instantiate the device
- Connect its external I/O to the top level
- Set its configuration parameters (i.e., base address)
- Add the slave outputs to the terms of the OR-gate that drives the PLB slave bus

The reference design has the logic in place for up to nine PLB slaves, with only two PLB slave slots used. The extra slave inputs which are tied off to logic 0 can be connected to the new slave. If larger numbers of slaves are needed, the OR logic can be expanded.



The PLB arbiter used in the reference design supports up to eight PLB masters, of which only two are instantiated. The other six PLB master ports are tied off to logic 0. The user can instantiate a new master and connect it to one of the unused PLB master ports in the arbiter.

An eight-master PLB arbiter is used in the reference design to make it easier to add PLB masters. However, different versions of the PLB arbiter are available which support a smaller number of PLB masters. These smaller arbiters are faster and use less logic resources than a large arbiter with some of its inputs tied off.

The bus logic in the PLB infrastructure supports up to nine PLB slaves. The bus logic is instantiated in the `plb_bus_logic.v` file and uses FPGA-optimized logic structures to build the large fan-in OR-gates more efficiently. The user may modify this file to implement logic that is more efficient for the exact number of PLB slaves in the system. Adding or removing PLB devices may also require adding or removing local signals. Any changes to the memory map must also be passed on to the PLB bus monitor.

## Adding or Removing DCR Devices

The user can add or subtract from the DCR bus very easily by modifying the data bus chain and OR'ing together the corresponding acknowledges back to the masters (as described in the DCR specification).

## Behavioral Models/Testbenches

Whenever new IP devices or BFM's are added, the testbenches should be updated. This may include new device models or edits to the connections of some of the bus monitors. The sample BFL script supplied with the design can be used as a template for building custom test scripts.

## Modifying Software

The `hello_uart` program can be used as a template for learning about writing and compiling software to run on the PPC405. The user can make small modifications to the `hello_uart` program (for example, changing the **Hello world** message) and take it through the process of building and simulating software. The tool flow can then be extrapolated for the design of new programs with greater complexity. A short outline of the steps for modifying and recompiling the `hello_uart` program are provided below. (Refer to **Volume 8, Chapter 3, PPC405 Software: Demonstration and Test Applications**, which provides detailed information about designing and compiling software to run on the CPU).

1. Copy `$V2PRO/source/sw` tree to your own area:  
`$ cp -Rp $V2PRO/source/sw <My_Apps_Dir>`
2. Change directory to the location of the source file in your own area:  
`$ cd <My_Apps_Dir>/apps/hello_uart`  
 Edit `hello.c` (change the **Hello world** message)
3. Go to the Simple PPC405 Reference System directory  
`$ cd $V2PRO/platforms/simple`
4. Edit the `flow.cfg` file to point to the location of the software program  
`$ <text editor> flow.cfg`  
`set SW_MAKE = make -C <My_Apps_Dir>/apps/hello_uart hello_debug_v2pdk`  
`set SW_ELF = <My_Apps_Dir>/apps/hello_uart/v2pdk/debug/hello.elf`
5. Run the simulation  
`$ make func_sim`

Your new message should be displayed on the terminal. The diagram in **Figure 2-6** summarizes the tool flow and source file relationships for the functional simulation of the

**hello\_uart** program as described in this document and the corresponding software documentation.

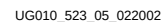


**Figure 2-6: Design Flow for Functional Simulation**

# File Structure for Simple PPC405 Reference System

```

\-----platforms
|
| \-----simple
| |
| | \-----par
| | |
| | | \-----v2p
| | |
| | | \-----sim
| | | |
| | | | \-----ba_sim
| | | | \-----bfl
| | | | \-----func_sim
| | | | \-----testbench
| | | | |
| | | | | \-----verilog
| | | | | \-----vhdl
| | |
| | | \-----syn
| | | |
| | | | \-----v2p
| | |
| | | \-----sys
| | | |
| | | | \-----mem
| | | | \-----verilog
| | | | \-----vhdl
|
| \-----source
| |
| | \-----hw
| | |
| | | \-----verilog
| | | |
| | | | \-----ipif_slv_sram_simple
| | | | \-----plb_arbiter
| | | | \-----plb_bram_cntlr
| | | | \-----plb_uart
| | | | \-----uart16550
| | |
| | | \-----sim_models
| | | |
| | | | \-----other
| | | | |
| | | | | \-----uart_rcvr
| | | |
| | | | \-----Xilinx
| | | | |
| | | | | \-----proc_block_BFM
| | | | | \-----verilog
| | | | | |
| | | | | | \-----smartmodel
| | |
| | | \-----sw
| | | |
| | | | \-----apps
| | | | |
| | | | | \-----hello_uart
| | | |
| | | | \-----libs
| | | | |
| | | | | \-----bsp
| | | |
| | | | \-----mapfiles
|
| \-----tools
| |
| | \-----cygwin
| | |
| | | \-----xilinx
| | | |
| | | | \-----bin
| | | | \-----data
| | |
| | | \-----solaris
| | | |
| | | | \-----xilinx
| | | | |
| | | | | \-----bin
| | | | | \-----data

```

## PPC405 PPE Reference System Using Rocket I/O™ Transceivers

### Introduction

The PPC405 Packet Processing Engine (PPE) Reference System Using Rocket I/O™ Transceivers addresses the need in the digital communications market for high-speed data transfer. Serial protocols can be controlled by complex logic, but are more simply handled by a microprocessor. This reference system is an example of a high-speed, serial-link packet processing engine implemented in Virtex-II Pro™ FPGAs. As shown in **Figure 3-1**, the reference system connects three PPC405-driven Rocket I/O multi-gigabit transceiver ports, each embedded within a single FPGA. The PPC405 application software required to run the serial transmission demonstration in a simulation environment is included.

The design uses a serial frame construct, consisting of control, headers, and payload/data bitstreams grouped according to a typical serial protocol such as Fibre Channel or Gigabit Ethernet. Each port has a Port\_ID that the application code reads to determine the function provided by each port. For example, Port A begins transmitting a 512-byte frame while Port B and Port C wait for incoming frames to process. If the port is not the destination, the frame is passed to the next port. Otherwise, an acknowledge frame is sent. Each serial port has a differential transmit output, TXP and TXN, and a differential serial input, RXP and RXN.

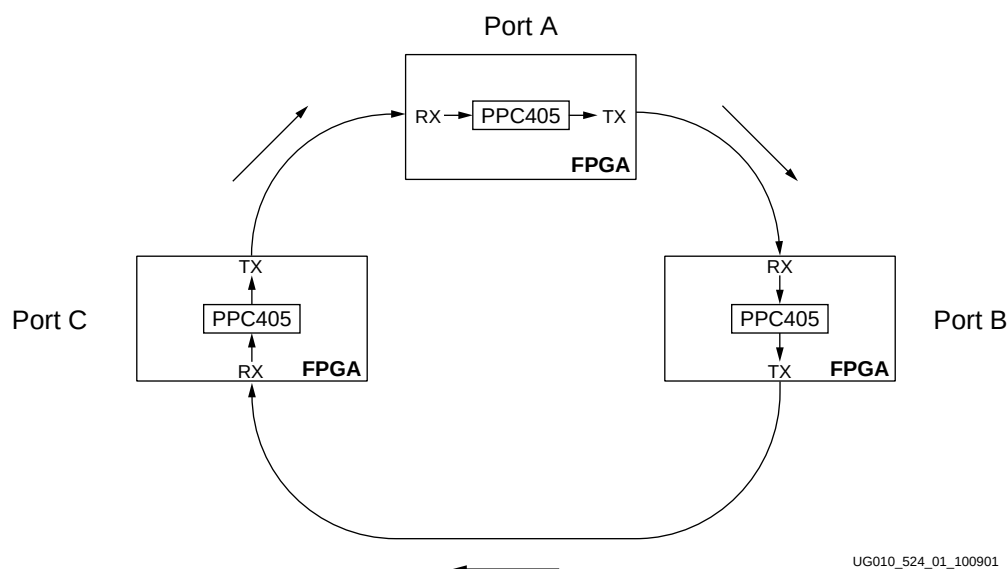


Figure 3-1: Serial Packet Flow

## Hardware

### Overview

Figure 3-2 provides a high-level view of the PPE system. Each port contains a PPC405 processor block, data-side block SelectRAM™ (DSBRAM) memory, instruction-side block SelectRAM (ISBRAM) memory, Rocket I/O transceiver, and packet processing interface logic. All registers used to control the application are implemented on the Device Control Register (DCR) bus from the PPC405 block.

The processor memory is contained within the block SelectRAM (BRAM) structure of the FPGA fabric. The PPC405 has an On-Chip Memory Controller (OCM) interface that supports 64 KB of DSBRAM and 128 KB of ISBRAM.

The design includes processor initialization code (Assembly language) and a "C" language routine to transmit and receive frames on the serial loop in a simulation environment.

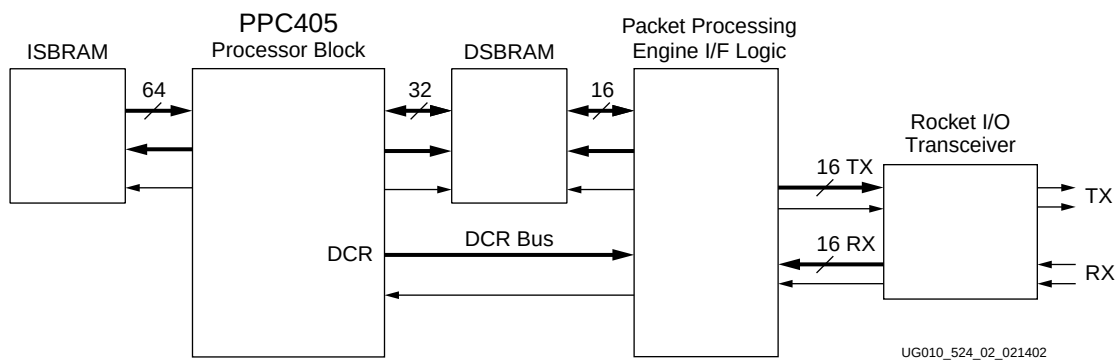


Figure 3-2: High-Level Hardware View of the PPE Reference System

### Processor Block Interfaces

The processor block contains the PPC405 core, a data-side OCM controller, an instruction-side OCM controller, reset control logic, and an internal DCR bus.

The processor block has the following interfaces:

- Data-side OCM to data-side BRAM
- Instruction-side OCM to instruction-side BRAM
- Clock and power management
- DCR interface
- External interrupts and reset control

For a complete description of the processor block interfaces, see the *PPC405 Processor Block Manual*.

## Instruction Side Block SelectRAM (ISBRAM)

The ISBRAM in this reference system is a 4-KB block SelectRAM memory organized as 512 x 64 bits. Two dual-port block SelectRAM components, **unisim** library part RAMB16\_S36\_S36, are used to construct the instruction-side memory.

The instruction-side OCM controller interfaces to the ISBRAM through a write-only port and a read-only port. The write-only port can be used to initialize the ISBRAM, allowing the user to write to the ISBRAM through the DCR interface. The read-only port is used to fetch two instructions (64 bits) per cycle.

## Data Side Block SelectRAM (DSBRAM)

The DSBRAM in this reference system is an 8-KB block SelectRAM memory organized as 2K x 32 bits. The PPE interface logic converts the 32-bit interface to a 16-bit interface for the Rocket I/O transceiver. Four block SelectRAM components, **unisim** library part RAMB16\_S9\_S9, are used to construct the data-side memory, which contains the 2-KB transmit buffer and the 2-KB receive buffer.

The data-side OCM controller interfaces to one side of the 32-bit DSBRAM through a read/write port. The other side of the DSBRAM is connected to the PPE interface logic. The data-side OCM controller supports byte, halfword, and word writes to the DSBRAM. It can be initialized by the processor or through the configuration bitstream.

The 8-KB DSBRAM memory is partitioned into segments to provide storage for firmware data, as well as transmit and receive buffers, as shown in [Table 3-1](#). The DSBRAM begins at address 0xF4000000.

**Table 3-1: DSBRAM Buffer Allocation**

| Address Offset | Usage                           |
|----------------|---------------------------------|
| 6 KB - 8 KB    | Transmit buffer                 |
| 4 KB - 6 KB    | Receive buffer                  |
| 0 KB - 4 KB    | Firmware data: stack, bss, etc. |

## Interrupts

The reference system does not use any interrupts. The two external interrupt inputs to the processor block are deasserted, as shown in [Table 3-2](#). The interrupt vector table is not initialized, since this requires an additional 8 KB of instruction memory.

**Table 3-2: Disabling the External Interrupt Interface**

| External Interrupt Input Signal | Signal Connection If Unused |
|---------------------------------|-----------------------------|
| EICC405EXTINPUTIRQ              | 0                           |
| EICC405CRITINPUTIRQ             | 0                           |

## Device Control Registers (DCR)

All microprocessor I/O are controlled through DCRs that the processor accesses either through *Move From DCR* (mfdcr) or *Move To DCR* (mtdcr) instructions. These are privileged-mode instructions.

## OCM Control Registers

The address for the data-side OCM (DSOCM) DCR registers is configured by connecting the TIEDSOCMADDR[0:7] inputs to the processor block to 0x06. The address for the instruction-side OCM (ISOCM) DCR registers is configured by connecting the TIEISOCMADDR[0:7] inputs to the processor block to 0x07.

The design configures the DSARC, DSCNTL, ISARC, and ISCNTL registers by connecting the OCM inputs to a constant value, as defined in [Table 3-3](#). The application code should not need to interface with the OCM DCR registers. The design does not use the ISINIT and ISFILL registers to write into ISBRAM. See the *PPC405 Processor Block Manual* for more information on the OCM controller.

Table 3-3: OCM DCR Register Summary

| Address (hex) | R/W | Register Name | Register Function                                                                                                                                                 |
|---------------|-----|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 01A           | R/W | DSARC         | Provides a decode address, compared against the 8 most significant bits of the OCM address. The design hard-wires the DSARCVAlUE OCM inputs to a hex value of F4. |
| 01B           | R/W | DSCNTL        | Enables the data-side OCM address compare function. The design hardwires the DSCNTLVALUE OCM inputs to a hex value of 83.                                         |
| 01C           | R/W | ISINIT        | Generates the 21-bit address to the write-only port of the instruction-side OCM. Not used in reference system.                                                    |
| 01D           | R/W | ISFILL        | Writes to the write only port of the instruction-side OCM. Not used in reference system.                                                                          |
| 01E           | R/W | ISARC         | Provides a decode address, compared against the 8 most significant bits of the OCM address. The design hardwires the ISARCVAlUE OCM inputs to a hex value of FF.  |
| 01F           | R/W | ISCNTL        | Enables the instruction-side OCM address compare function. The design hardwires the ISCNTLVALUE inputs to a hex value of 83.                                      |

## PPE Device Control and Status Registers

The processor controls the transmission and reception of serial frames through the PPE Control Registers, as defined in [Chapter 21, Packet Processing Engine Controller](#). The OCM controller forwards all non-OCM DCR accesses on to the FPGA. The FPGA asserts the DCR acknowledge signal to terminate the DCR bus cycle. The DCR cycle must complete within 64 processor clocks.

## Rocket I/O Transceiver Interface

The Rocket I/O transceiver interface consists of a 16-bit transmit bus, a 16-bit receive bus, a reference clock, and control signals. The design uses the 8B/10B encoder and 8B/10B decoder functions in the transceiver. Refer to [Chapter 21, Packet Processing Engine Controller](#) for a description of the interface signals. For detailed information about the Rocket I/O transceiver, see the *Virtex-II Pro Handbook* and [Volume 3, Rocket I/O™ Transceiver User Guide](#).

## Rocket I/O Attributes

There are 16 Rocket I/O transceiver primitives in the **unisim** library. These primitives have certain attributes that are set for specific protocol and transfer interfaces—for example, 8-, 16-, or 32-bit interfaces to the FPGA fabric. The PPE Reference System uses the

GT\_CUSTOM primitive, which allows the most flexibility. Its interface with the FPGA fabric is shown in the *Rocket I/O Transceiver Packet Processing Engine Controller* document.

## Clock/Reset Distribution

### Clocks

The design requires several clock domains including a processor clock, serial transceiver clock, OCM controller clock, and DCR module clock as shown in [Figure 3-3](#).

#### Processor and Processor-Side BRAM

The processor runs at 300 MHz. The PPC405 clock operates at a 2:1 clock ratio with respect to the instruction side and data side OCM controller clocks.

#### OCM and DCR Module Interface Clocks

The OCM and DCR module interface clocks are set up in 1:2 ratio to the processor core clock. To change this ratio, you must change the clock design in the **top** and **clk\_rst\_startup** modules. The data-side OCM and instruction-side OCM multicycle mode control register bits must be modified for various PPC405 to OCM clock ratios, as shown in [Table 3-4](#). The bits are in the DSCNTL and ISCNTL OCM DCR registers.

**Table 3-4: Adjustment of OCM Multicycle Mode Clock Ratio**

| Signal                               | Value (binary) | PPC405:OCM Clock Ratio |
|--------------------------------------|----------------|------------------------|
| DSOCMMCM[0:2]<br>or<br>ISOCMMCM[0:2] | 000            | Not Supported          |
|                                      | 001            | 1:1                    |
|                                      | 010            | Not Supported          |
|                                      | 011            | 2:1                    |
|                                      | 100            | Not Supported          |
|                                      | 101            | 3:1                    |
|                                      | 110            | Not Supported          |
|                                      | 111            | 4:1                    |

The Rocket I/O transmit and receive interface is 16 bits. The testbench module provides a 75 MHz clock for the OCM/DCR and PPC405 DCM circuit, as well as a 156 MHz clock for the PPE and Rocket I/O transceiver DCM circuit.



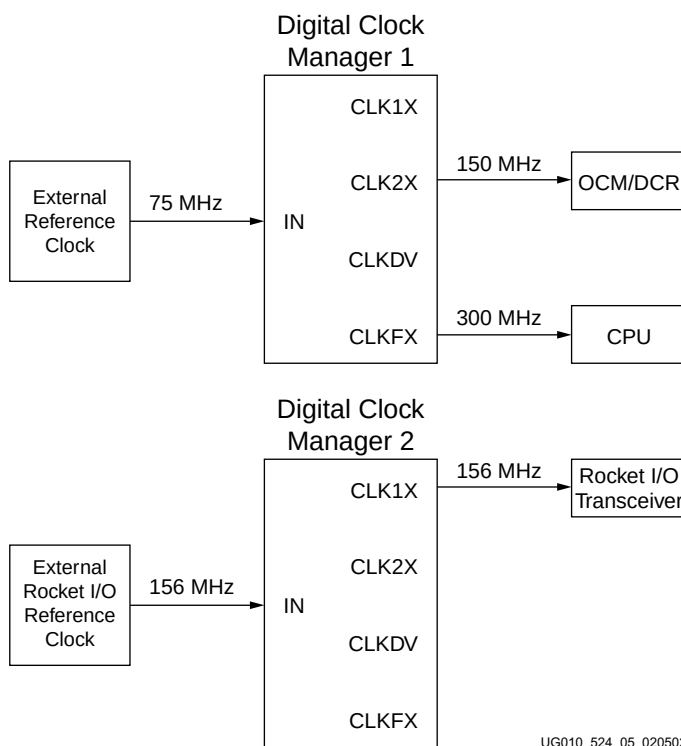


Figure 3-3: Clock Generation

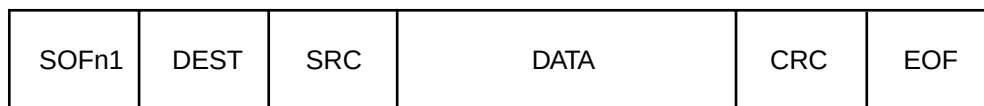
## Resets

Shortly after DCM Lock, the reset is deasserted to all devices. The processor requires a four clock minimum pulse on the reset input. The PPE logic uses the same reset as the 405 core. The Rocket I/O transceiver has two reset inputs. **TXRESET** is used for the transmit side and **RXRESET** is used for the receive side of the interface. The transceiver synchronizes the reset inputs to the reference clock. Following the deassertion of **TXRESET**, there is a latency until valid data is transmitted by the transceiver.

## Theory of Operation

### Frame Format

The reference system implements a frame, shown in Figure 3-4. The frame consists of a start-of-frame primitive, a header field, a data field, an optional CRC field, and an end-of-frame primitive.



UG010\_524\_03\_100901

Figure 3-4: Rocket I/O Transceiver Serial Frame Format

The start-of-frame (**SOFn1**) and end-of-frame (**EOF**) primitives are defined in Table 3-5. The frame header includes the following information:

- A 32-bit DESTINATION port address that identifies the receiver port information
- A 32-bit SOURCE port address that identifies the sender of the frame

## Frame Size

The frame size is 256 halfwords for this application.

## CRC

CRC is not implemented in the reference system.

## Framing Protocol Characters

When the port is not transmitting, an **IDLE** primitive signal, defined in Table 3-5, is transmitted. The design does not attempt to implement higher-level communications protocols such as port initialization, port reset, port discovery, physical addressing, or buffer credit management. The K28.5 character, referred to as a *comma*, is used for character and word boundary alignment. A minimum of six **IDLE**s are transmitted between frames.

Table 3-5: Ordered Set Primitives

| Framing Protocol Character | Ordered Set  | TXCHARISK[1:0]<br>or<br>RXCHARISK[1:0] | TXDATA[15:0]<br>or<br>RXDATA[15:0] |
|----------------------------|--------------|----------------------------------------|------------------------------------|
| <b>IDLE</b>                | K28.5, D21.4 | 10                                     | BC95                               |
|                            | D21.5, D21.5 | 00                                     | B5B5                               |
| <b>SOFn1</b>               | K28.5, D21.5 | 10                                     | BCB5                               |
|                            | D23.1, D23.1 | 00                                     | 3737                               |
| <b>EOFn</b>                | K28.5, D21.4 | 10                                     | BC95                               |
|                            | D21.6, D21.6 | 00                                     | D5D5                               |

## Half-Duplex Operation

The design supports half-duplex operation. (Full-duplex operation would require separate BRAMs for simultaneous receive and transmit operations.) The data-side OCM is partitioned into a transmit buffer and a receive buffer. The dual port BRAM does not support simultaneous reads and writes to different addresses on the same port. A receive FIFO is placed in the RX data path between the gigabit transceiver and the DSB RAM. The RX FIFO provides temporary storage for incoming frames if the transmit frame logic is accessing DSB RAM. The RX FIFO is 1K halfwords deep.

### Transmit Buffer

The transmit buffer holds a maximum of 2 KB. The **SOF** and **EOF** characters are not stored in the buffer. The **SOF** and **EOF** characters are constants that are multiplexed into the TX data stream by a state machine sequencer in the PPE interface logic module `pkt_proc`.

### Receive FIFO and Receive Buffer

Receive frames arrive asynchronously. To handle simultaneous transmit and receipt of frames, a receive FIFO is required to hold incoming frames until the frame transmit function completes. The PPE moves the receive packet from the FIFO to the receive buffer in the data-side OCM. The RX FIFO holds a maximum of 2 KB –2.

## Example Applications

This section refers to the term “software” with respect to programs running on the PPC405 processor blocks. It does not refer to implementation or place-and-route tools.

The PPE example application demonstrates software development for the PPC405 core. The application demonstrates the PPC405 interacting with control logic to implement a serial data transmission protocol using start-of-frame and end-of-frame delimiters.

The PPE example application is located in `$V2PRO/source/sw/apps/pkt_proc`. It is a stand-alone application and can be compiled and linked without references to any libraries.

### Initialization

The *PPC405 User Manual* recommends performing several tasks during the initialization of the PPC405. Only those features that need to be changed from their reset states need to be configured. Following is a summary of those tasks and how the PPE application implements them:

- Configure the following storage attribute control registers, if necessary:
  - Initialize the Storage Guarded Register (SGR) to configure guarded or unguarded storage.
  - Initialize the Storage Little-Endian Register (SLER) to configure storage byte ordering.
  - Initialize the Storage User-defined 0 Register (SU0R) to configure storage compression.

The PPE Application initializes the SGR register to 0x00000000. This turns off guarded storage for all of memory, allowing speculative prefetching. The SLER and SU0R registers are left in their reset state, allowing big-endian storage and no user-defined attributes.

Code snippet from `crt0.S`:

```
addis r3, 0,0x0000 /* Init SGR to turn off guarded */
ori r3,r3,0x0000 /* storage */
mtsgpr r3
```

- Before executing instructions as cacheable:
  - Invalidate the instruction cache.
  - Initialize the Instruction Cache Cacheability Register (ICCR) to configure instruction cacheability.

The instruction cache is flash-invalidated with the `iccci` instruction. The instruction cache is not turned on since the application is executing solely from instruction side and data side OCM.

Code snippet from `crt0.S`:

```
iccci 0,r3 /* Invalidates ENTIRE I-Cache */
isync
lis r3,0x0000 /* Init Cache Control Registers */
ori r3,r3,0x0000 /* All regions are non-cacheable */
mticcr r3
isync
```

- Before using storage access instructions:
  - Invalidate the data cache.
  - Initialize Core Configuration Register (CCR0) to determine if a store miss results in a line fill.
  - Initialize the Data Cache Write-Through Register (DCWR) to select write-back or write-through caching.

- Initialize the Data Cache Cacheability Register (DCCR) to configure data cacheability.

The data cache must be invalidated by looping through and invalidating each congruence class. This portion of the code is commented out in **crt0.S** to save on simulation cycles. The simulator performs this function for us. The CCR0 and DCWR are initialized for the sake of completeness.

Code snippet from **crt0.S**:

(NOTE: In the actual **crt0.S** file, the CCR0 register is initialized just before I-cache is invalidated.)

```

 addi r3, 0, 256 /* 256 Time to Loop */
 mtctr r3 /* Load the count register */
 addi r3, 0, 0 /* Clear R3 */
loop: dccci 0, r3 /* Invalidate congruence cache */
 addi r3, r3, 32 /* Move Forward one Cache Line */
 bdnz loop /* Loop back to do rest */

 addis r3, 0, 0x0000 /* No need for prefetching... */
 ori r3, r3, 0x0000 /* ...since we are operating... */
 /* ...completely out of OCM */

 mtspr ccr0, r3 /* Load Core Configuration Register */

 addis r3, 0, 0x0000 /* Init the DCWR for write back policy */
 ori r3, r3, 0x0000
 mtdcwr r3

 lis r3, 0 /* Init Cache Control Registers */
 ori r3, r3, 0x0000 /* All memory is non-cacheable */
 sync
 mtdccr r3
 isync

```

NOTE: In this example application, ISARC and ISCNTL are preconfigured by the hardware to permit the processor to boot from ISOCM.

- Before allowing interrupts (synchronous or asynchronous):
  - Initialize the Exception Vector Prefix Register (EVPR) to point to vector table.
  - Provide vector table with branches to interrupt handlers.

The PPE application does not initialize the EVPR or the vector table.

- Before enabling asynchronous interrupts:
  - Initialize timer facilities.
  - Initialize Machine State Register (MSR) to enable appropriate interrupts.

The MSR is initialized to 0x00000000.

- Initialize other processor features, such as the MMU, cache line locking, debug, and trace.

These features are not initialized, hence the processor runs in real mode.

- Initialize non-processor resources.
  - Initialize system memory as required by the operating system or application code.
  - Initialize off-chip system facilities.

OCM registers such as DSARC, ISARC, DSCNTL, and ISCNTL are initialized to set up the OCMs. The **bss**, **sbss** and **stack** are initialized as well.

- Start the execution of operating system or application code.

Branch to **main()**.

Table 3-6: Values of Certain Registers Out of Reset

| Register | Field  | Core Reset Value       | Comment                                  |
|----------|--------|------------------------|------------------------------------------|
| MSR      | APE    | 0                      | Auxiliary Processor exception disabled   |
|          | WE     | 0                      | Wait state disabled                      |
|          | CE     | 0                      | Critical interrupts disabled             |
|          | EE     | 0                      | External interrupts disabled             |
|          | PR     | 0                      | Supervisor mode                          |
|          | ME     | 0                      | Machine check interrupts disabled        |
|          | DE     | 0                      | Debug interrupts disabled                |
|          | IR     | 0                      | Instruction translation disabled         |
|          | DR     | 0                      | Data translation disabled                |
| CCR0     |        |                        | TBD                                      |
| DBCR0    | EDM    | 0                      | Debug mode and events disabled           |
|          | RST    | 00                     | No reset action                          |
| DBSR     | MRR    | Most recent reset type | Most recent reset type                   |
| ESR      | MCI    | 0                      | Machine check exception has not occurred |
| ICCR     | S0:S31 | 0x00000000             | Instruction cache is disabled            |
| PVR      | 0:31   |                        | Set to assigned PVR value for core       |
| SGR      | G0:G31 | 0xFFFFFFFF             | Storage is guarded                       |
| SU0R     | K0:K31 | 0x00000000             | Storage is not compressed                |
| SLER     | S0:S31 | 0x00000000             | Storage is big endian                    |
| TCR      | WRC    | 00                     | Watchdog timer reset disabled            |
| TSR      | WRS    | Copy of TCR[WRC]       | If reset caused by watchdog timer        |
|          |        | Undefined              | After power-up                           |
|          |        | Unchanged              | If reset not caused by watchdog timer    |

## Application Firmware

The PPE application firmware demonstrates the use of the PPC405 along with the DCR bus, data-side OCM, instruction-side OCM, and the high-speed serial I/O. The firmware does the following tasks:

- Identifies the port (Port A, B, or C) by reading the Rocket I/O status register.
- Places the Rocket I/O transceiver in serial loopback mode and performs a port diagnostic. If the diagnostic passes, the Rocket I/O serial transmit and receive functions are enabled.
- Sends out a 512-byte packet if it determines that it is Port A.
- Polls **Rx\_Tail\_Ptr** for incoming packets.
- Does one of the following (depending on the incoming packet's destination header

field):

- Retransmits the packet to the next node.
- Accepts the packet and sends out an acknowledgement packet.

### Confirming the Transmit and Acknowledge Packets

The PPE and Rocket I/O transceiver transmit IDLE characters the majority of the time. Locating the data in packets in the MTI waveform window is difficult. Although the MTI window messages verify correct operation, viewing the signal waveforms is required if more in-depth investigation is desired.

First, to find a *transmitted packet*, look in the PORT\_A interface section of the waveform window for the transmit signal to go High. This indicates the beginning of a packet transmission. You can verify the correct data by checking the **TXDATA[15:0]**. Find the **SOF**, shown in [Table 3-5, page 1273](#). Then the 16-bit **TXDATA**, in hexadecimal form, will be as follows (first hexadecimal character is the **TXCHARISK[1:0]** signal):

```

SOF
00000
00003 Destination PORT
00000
00001 Source PORT
00001 start incrementing data frame pattern
00203
00405
00607
.....
.....
0f4f5
0f6f7 end of ascending order
0ffff
0ffff end of data character
EOF

```

The *acknowledge packet* is sent from PORT\_C to PORT\_A. In the message window, check the time the PORT\_C starts to transmit. In the waveform window, look at the **RXDATA[15:0]** for PORT\_A at the same time. **RXDATA** should start with the **SOF** followed by the data. After the first 64 bits of data, the radix should be converted to ASCII for the middle portion of the data.

```

SOF
00000
00005
00000
00005 hex value of acknowledge packet
T_H_A_N_K___
Y_O_U___
D_A_T_A___
H_A_S___
B_E_E_N___
R_E_C_E_I_V_E_D___ ascii value of acknowledge packet
0ffff
0ffff
00000
00000
.....
.....
00000
00000 hex value of end of acknowledge packet
EOF
""

```

## Source Files Location

The source files needed to do a complete firmware build are located in `$V2PRO/source/sw/apps/pkt_proc` directory. Table 3-7 gives a brief overview of what each source file is for. Include files are located in `$V2PRO/source/sw/apps/pkt_proc/includes`.

Table 3-7: Application Firmware Modules

| File Name    | Purpose                                                                                                                                                                                                                 |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| boot.S       | Contains a section of code for the reset vector at 0xFFFF FFFC. This single instruction jumps to <code>_start</code> in <code>crt0.S</code> . The boot section is mapped to the reset vector during linking.            |
| eabi.S       | Contains code to initialize the EABI environment, i.e., setting the registers R2 and R13 to the correct values. The compiler inserts a call to <code>_eabi</code> at the beginning of the <code>main()</code> function. |
| crt0.S       | Contains code to initialize the <code>.bss</code> and <code>.sbss</code> data sections and to set up the stack. This file also contains code to initialize processor features listed in the Initialization section.     |
| ppc-asm.h    | Contains <i>register defines</i> specific to the PPC405.                                                                                                                                                                |
| pkt_proc.c   | Contains the <code>main()</code> function and declares global variables used by all modules.                                                                                                                            |
| initialize.c | Contains function calls to initialize all of the high-speed serial hardware.                                                                                                                                            |
| data.c       | Contains functions for packet processing. Also contains device driver functions to initialize and test the Packet Processing Engine hardware.                                                                           |
| includes.h   | Included in all <code>.c</code> files. Includes all other <code>.h</code> files. Contains <i>defines</i> and <i>macros</i> for all modules. Also, contains <i>extern global variable</i> declarations.                  |
| initialize.h | Contains <i>extern function</i> declarations for functions used by other modules.                                                                                                                                       |
| data.h       | Contains <i>extern function</i> declarations for functions used by other modules.                                                                                                                                       |
| Makefile     | Builds the PPE example application.                                                                                                                                                                                     |
| mapfile      | The mapfile is the linker script file. It defines the memory map and how sections are mapped within memory.                                                                                                             |

## Using the DCR to Debug in ModelSim

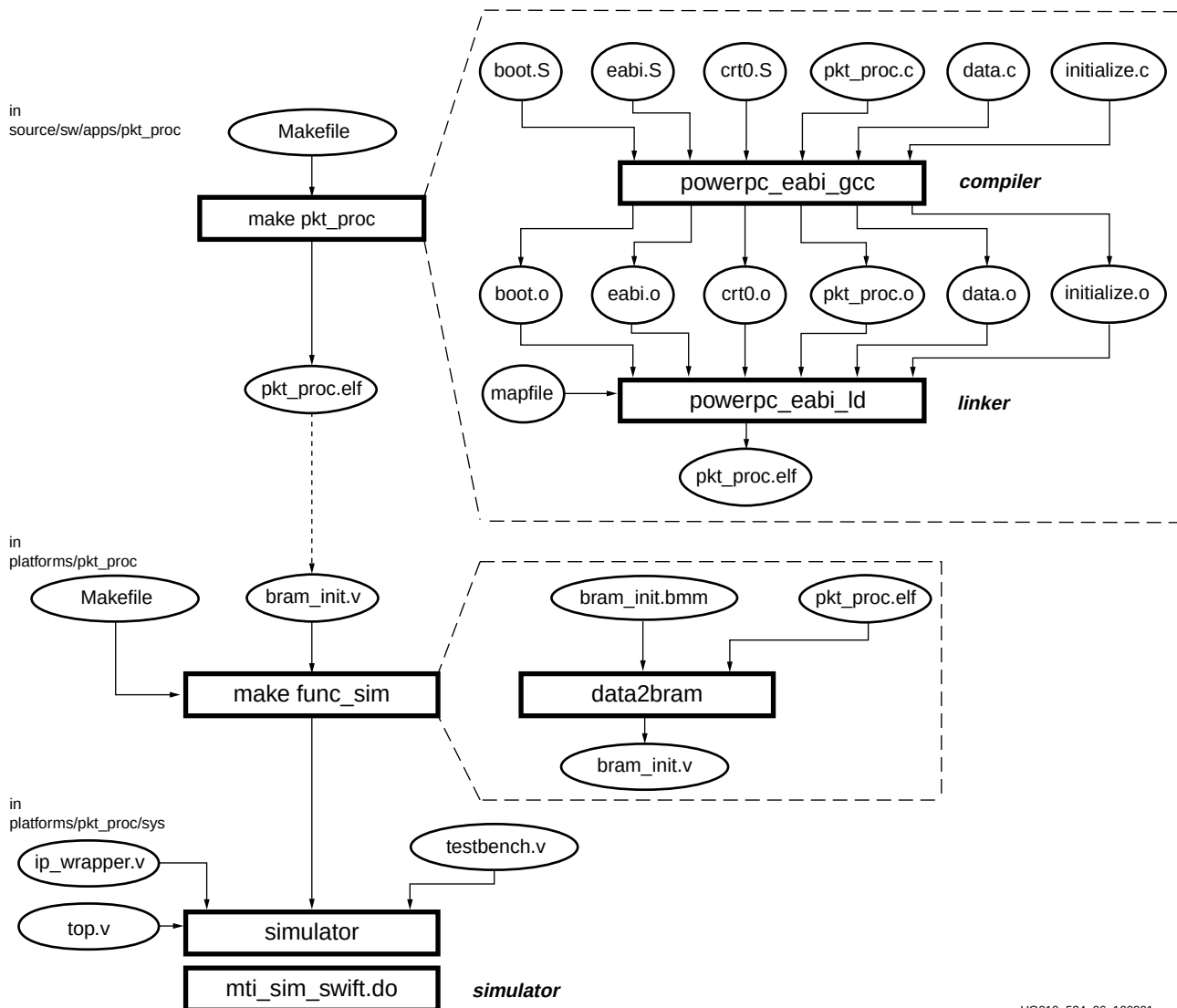
The `TransmitPacket()` function in `data.c` and the `InitializeAll()` function in `initialize.c` contain some `mtdcr` instructions that go to DCR register 300. This register does not actually exist, and the instruction is used as a `printf`-type debugging statement. This works well in the ModelSim environment when running out of cache, since cached instructions do not result in a PLB transaction.

## Execution of Software

The application code is ready to simulate, but if you want to recompile it, go to the `platforms/pkt_proc` directory and run the Makefile.

**NOTE:** When opening a new Xterm window, make sure to source the `v2pro_setup` file.

```
$ make sw
```



UG010\_524\_06\_100901

Figure 3-5: Design Flow

## HDL Organization

Figure 3-6, page 1280 diagrams the organization of the higher-level HDL files that comprise the system and testbench environment. The **ip\_wrapper.v** and **top.v** are files that assemble the pieces that make up the system in the FPGA. Some of these pieces include the processor block (as a simulation model), clock/reset logic, packet processing user block, and Rocket I/O transceiver.

For simulation, the **testbench.v** module instantiates three top levels of the design as the device under test. It provides the necessary functionality to drive the simulation and capture information. This module generates the Rocket I/O transceiver clocks and DCM reference clock and reset signals for the FPGA. The **sim\_params.v** file is designed to be user modified to customize various simulation options. These options include message display options, maximum simulation time, and clock frequency. Users should edit this file to reflect their simulation preferences.

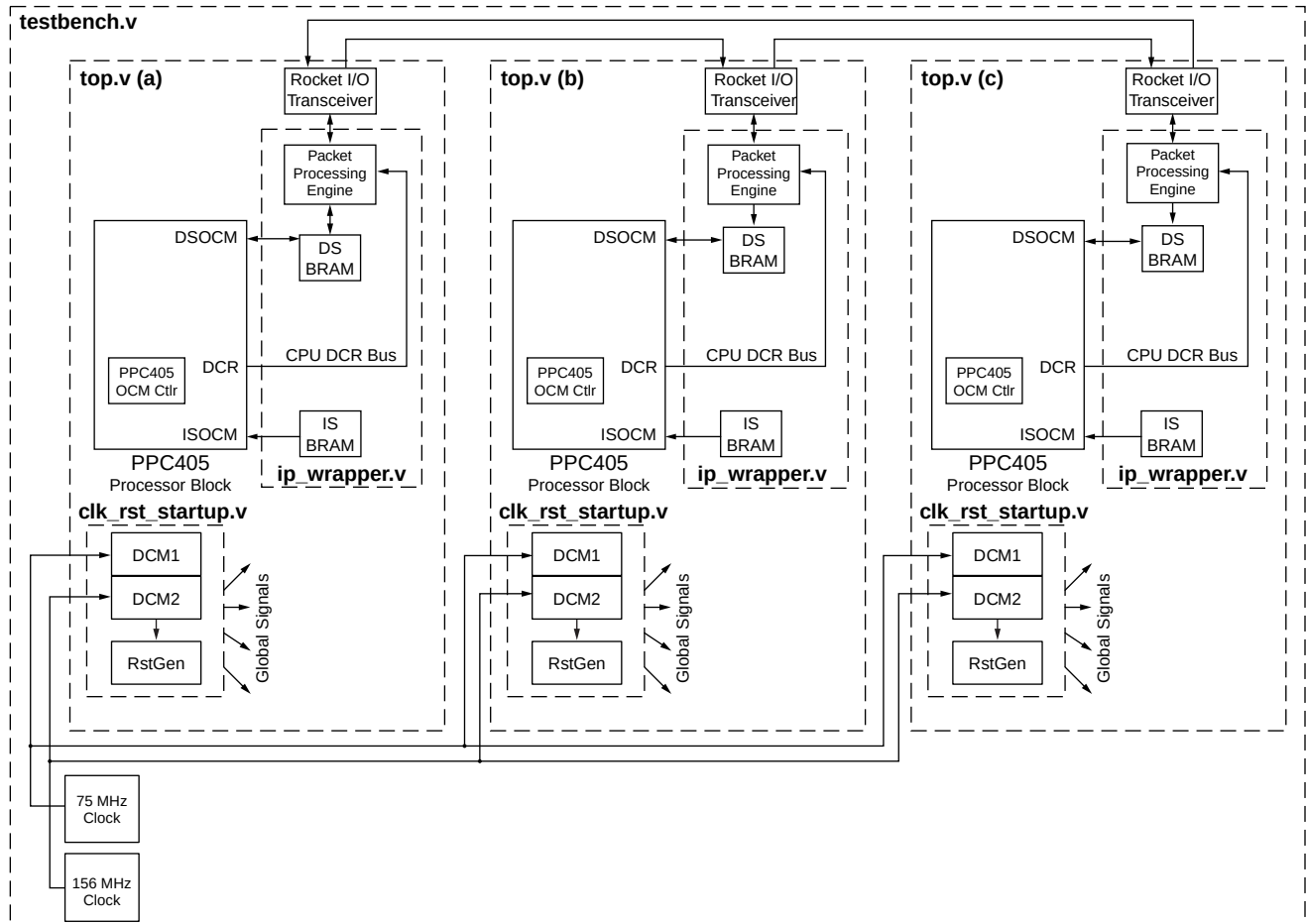


The packet processing engine (**pkt\_proc.v**) module contains the packet processing user block, which reads the PPE control registers and writes the PPE status registers. The control logic creates and retrieves packet data from BRAM and sends it to the Rocket I/O transceiver. After the user block receives packet data, the IDLE characters are removed, and the data is placed in a FIFO. The receive portion waits for the transmit portion to allow access to the BRAM so receive data can be placed in memory.

The **top.v** file instantiates the clock/reset logic, processor block and **ip\_wrapper.v** file. The module ports of **top.v** represent the external I/O signals of the FPGA. Therefore, all files in the hierarchy from the **top.v** level and below make up the Virtex-II Pro FPGA design, and would be synthesized/routed into an FPGA.

An additional HDL file called **global\_params.v** defines the global constants and parameters used throughout the reference design. This information may include memory map definitions, device configuration values, or any other information that is better organized in a global context.

Some of the testbench code accesses signals internal to the design, using hierarchy path names to reach into the design without changing any of the port interfaces. It is important that the design source files used for simulation match the source files for synthesis. Therefore, port interfaces should not be different, or else inconsistencies can result.



UG010\_523\_07\_021902

Figure 3-6: Organization of Higher-Level HDL Files

## Simulation and Verification

### SWIFT and BFM CPU Models

The PPC405 Packet Processing Engine Reference System demonstrates two different simulation methods to help verify designs using the PPC405 CPU. One method uses a full simulation model of the CPU based on the actual silicon. The second method employs bus functional models (BFMs) to generate processor bus cycles from a command scripting language. These two methods offer different trade-offs between behavior in real hardware, ease of generating bus cycles, and the amount of real time to simulate a given clock cycle.

A SWIFT model can be used to simulate the CPU executing software instructions. In this scenario, the executable binary images of the software are preloaded into memory from which the CPU can boot up and run the code. Though this is a relatively slow way to exercise the design, it more accurately reflects the actual behavior of the system. The SWIFT model is most useful for helping to bring up software and for correlating behavior in real hardware with simulation results. The PPC405 Packet Processing Engine Reference System demonstrates the SWIFT model simulation flow, by allowing the user to write a C program that is compiled into an executable binary file. This executable (in ELF format) is then converted into BRAM initialization commands using a tool called Data2BRAM. When a simulation begins and reset is released, the CPU SWIFT model fetches the instructions from BRAM (which is mapped to the boot vector) and begins running the program. The user can then observe the bus cycles generated by the CPU or any other signal in the design. For debugging purposes, the values of the CPU's internal program counter, general-purpose registers, and special-purpose registers are available for display during simulation.

Generating a desired sequence of bus operations from the CPU may require a lot of software setup or simulation time. For early hardware bring-up or IP development, a bus functional model can be used to speed up simulation cycles and avoid having to write software. A model of the CPU is available in which two PLB master BFMs and one DCR BFM are instantiated to drive the CPU's PLB/DCR ports. These BFMs are provided in the CoreConnect toolkits and allow the user to generate bus operations by writing a script written in the Bus Functional Language (BFL). The PPC405 Packet Processing Engine Reference System provides a sample BFL script that exercises the DCR bus peripherals in the system. Refer to the *CoreConnect Toolkit* documentation for more information.

Since the CPU SWIFT model and BFM model both have the same set of port interfaces, users can switch between the two simulation methods by compiling the appropriate set of files without having to modify the system's design source files. Users may, however, need to modify their testbenches to take into account which model is being used.

### Behavioral Models

The PPC405 Packet Processing Engine Reference System does not include any behavioral models.

## Synthesis and Implementation

The PPC405 Packet Processing Engine Reference System can be synthesized and placed/routed into a Virtex-II Pro FPGA. A basic set of timing constraints for the design is provided to allow the design to go through place and route. Note that some peripherals in the design are presynthesized and provided only in netlist format (.edf). For these devices, a corresponding black box has been instantiated in the source code to force the synthesis tool to leave a placeholder for the netlist. After a successful place and route, it is possible to run a simulation of the design using a back-annotated timing model of the FPGA with a SWIFT or BFM version of the processor block.

## Design Flow Environment

A flow engine provides an environment to help manage the design flow for the PPC405 Packet Processing Engine Reference System. This engine uses the utility program called **make** and a set of PERL scripts to allow the user to perform tasks such as running simulations, synthesizing a design, or implementing it on an FPGA with a simple set of commands. The design flow tool is implemented with a generic architecture to allow it to be adapted for use on a variety of different designs. For more information, refer to *Flow Documentation*.

## File Listings

The files and directories specific to the PPC405 Packet Processing Engine Reference System are located below the **\$V2PRO/platforms/pkt\_proc** directory, shown in **Figure 3-7**, and listed in the tables that follow. Source files for the IP modules and software applications are kept in the **\$V2PRO/source** directory. The **\$V2PRO/source** directory generally contains files that can be shared by different designs. The **File Structure for PPE Reference System** section diagrams the full file directory structure for this design. Refer to the appropriate instructions for installation and setup information. Note that the following tables only list files in the **\$V2PRO/platforms/pkt\_proc** directory that are present after installation. After running simulation, synthesis, or place and route, additional files may be created. Directory path names are shown separated by the "/" character as is the UNIX convention. For Windows, the "\" should be used to separate directory paths.

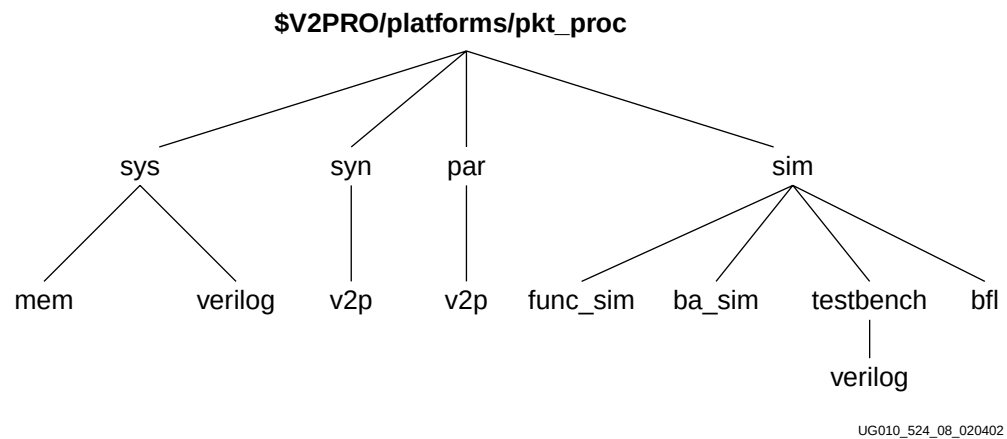


Figure 3-7: Map of Directories Below **\$V2PRO/platforms/pkt\_proc**

Table 3-8: Files in the **\$V2PRO/platforms/pkt\_proc** Directory

| File     | Description                                                                                          |
|----------|------------------------------------------------------------------------------------------------------|
| Makefile | Redirection file to allow the centralized design flow scripts to be invoked from different locations |
| flow.cfg | File used to configure the design flow scripts                                                       |

*Table 3-9: Files in the \$V2PRO/platforms/pkt\_proc/sys Directory*

| File          | Description                                                                                               |
|---------------|-----------------------------------------------------------------------------------------------------------|
| bram_init.bmm | Memory configuration file that describes the organization and address map for all the BRAMs in the system |

*Table 3-10: Files in the \$V2PRO/platforms/pkt\_proc/sys/verilog Directory*

| File              | Description                                                                                                   |
|-------------------|---------------------------------------------------------------------------------------------------------------|
| src.lst           | List of source files for the design                                                                           |
| clk_rst_startup.v | Clock, reset, startup logic for the design                                                                    |
| ip_wrapper.v      | Contains all user IP including <b>pkt_proc_dcr_module</b> , Packet Processing Engine logic, and BRAM wrappers |
| top.v             | Top-level file for Virtex-II Pro FPGA. Instantiates PPC405 and Rocket I/O transceiver                         |

*Table 3-11: Files in the \$V2PRO/platforms/pkt\_proc/sim/func\_sim Directory*

| File    | Description                    |
|---------|--------------------------------|
| wave.do | Waveform file for SWIFT models |

*Table 3-12: Files in the \$V2PRO/platforms/pkt\_proc/sim/bfl Directory*

| File              | Description                        |
|-------------------|------------------------------------|
| test_pkt_proc.bfl | Bus Functional Language (BFL) file |

*Table 3-13: Files in the \$V2PRO/platforms/pkt\_proc/sim/ba\_sim Directory*

| File    | Description                                                                                     |
|---------|-------------------------------------------------------------------------------------------------|
| wave.do | MTI command to display a sample set of interesting signals for back-annotated timing simulation |

Table 3-14: Files in the \$V2PRO/platforms/pkt\_proc/sim/testbench/verilog Directory

| File                | Description                                                                         |
|---------------------|-------------------------------------------------------------------------------------|
| sim_params.v        | Centralized location for parameters and settings that affect simulation             |
| testbench.v         | Top-level testbench wrapper around Virtex-II Pro FPGA                               |
| view_mult_cpu_reg.v | Behavioral code to display internal CPU register contents whenever they are updated |

Table 3-15: Files in the \$V2PRO/platforms/pkt\_proc/par Directory

| File    | Description                                             |
|---------|---------------------------------------------------------|
| top.ucf | User constraints file used by FPGA implementation tools |

Table 3-16: Files in the \$V2PRO/platforms/pkt\_proc/syn Directory

| File         | Description                                 |
|--------------|---------------------------------------------|
| synplify.opt | Optional synthesis project file information |

## Files Outside the platforms Directory

Below are the files that are common modules or IP blocks that can be shared among different designs. These are files that are called outside the **platforms** directory.

Table 3-17: Files in the \$V2PRO/source/hw/verilog/pkt\_proc Directory

| File                  | Description                                                                    |
|-----------------------|--------------------------------------------------------------------------------|
| pkt_proc.v            | Contains functions that create and break down packets                          |
| pkt_proc_dcr_module.v | Contains DCR registers used for data interface of <b>pkt_proc.v</b> and PPC405 |
| pkt_proc_dsocm_bram.v | Wrapper for all DS BRAM                                                        |
| pkt_proc_isocm_bram.v | Wrapper for all IS BRAM                                                        |

## Instructions for Running Functional Simulations

The PPC405 Packet Processing Engine Reference System comes with SWIFT and BFM based simulation examples. This section describes the necessary steps for running these simulations using functional timing with the original source code. It assumes the Virtex-II Pro Design Kit is properly installed and that the **\$V2PRO/platforms/pkt\_proc/flow.cfg** file is properly edited to correctly call the user's simulation tools. The user may also modify the **sim\_params.v** file to customize various simulation parameters.

The application initially places the Rocket I/O transceiver in serial loopback mode and performs a port diagnostic on the packet processing hardware. A test packet is transmitted and serially looped back through the transceiver, through the Packet Processing Engine logic and back to the dual-port data memory. The PPC405 compares the transmitted packet and receive packets. If the packets are identical, the loopback mode is disabled and

the Rocket I/O port is ready to transmit application data. The MTI window displays when the port diagnostic begins and ends. When the different ports start and end a transmission, the MTI window displays the time when these events occur, and you can see the waveforms on the wave window.

The design example shows three top blocks that consist of a PPC405 SWIFT model, a Rocket I/O transceiver SWIFT model, and some interface logic. These modules are connected together in a loop. Port A sends out a 512-byte packet that is received by Port B, which recognizes that it is not the serial destination. Port B sends the frame on to Port C, which receives the frame and then transmits an acknowledge frame back to Port A. Entire simulation time is 70  $\mu$ s.

See **Confirming the Transmit and Acknowledge Packets**, [page 1277](#), for additional information concerning the simulation waveforms.

Refer to the *Release Notes: Virtex-II Pro™ Development Kit* in conjunction with the following procedures.

## SWIFT Simulations

The steps to run a SWIFT simulation are described below:

1. Go to the main directory of the design and edit the **flow.cfg** to select the software application you wish to run:

```
$ cd $V2PRO/platforms/pkt_proc
$ <text editor> flow.cfg
```

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

The *<text editor>* can be a program like Emacs, vi, or WordPad. In the **flow.cfg** file, uncomment the SW\_MAKE and SW\_ELF lines for the program you wish to run and leave the other programs commented out. After installation, the **flow.cfg** file is set to the **pkt\_proc** program, which is precompiled and ready to run. Also, make sure that the PPC\_MODEL entry is set to "swift". After editing the **flow.cfg** file, save it.

2. Invoke the design flow engine to start the simulation:

```
$ make func_sim
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **pkt\_proc** → **Make Functional Simulation**)

The flow engine begins by compiling and linking the application software (as necessary) to generate a binary executable program file (in **.elf** format). The **.elf** file is copied locally, then **data2bram** is invoked to generate the necessary BRAM initialization command and **.mem** files. The simulator tool is then launched.

**Note:** For MTI, the scripts will copy a corresponding NT or Solaris version of the **modelsim.ini** configuration file into the local directory. This file can be found in the **\$V2PRO/tools/\$V2PRO\_HOST\_OSTYPE/xilinx/data/modelsim.ini** directory.

3. Run the simulation:

In the MTI window, type **run -all**.

For Verilog XL, the simulation begins immediately. If the " 'define VIRSIM " command in **sim\_params.v** is set, the waveforms for all the signals in the design are stored for the **virsim** program to display. You may need to edit the testbench to invoke your own Verilog XL waveform capture and display program.

It is normal to see some warnings from the PLB monitor or behavioral memory models during reset, but the DCR monitors should not report any protocol errors during simulation (warnings and notes may occur depending on the circumstances).

## BFM Simulations

The steps to run a BFM simulation are described below:

1. To run a BFM simulation, edit the **flow.cfg** file to change the processor block model to BFM:  

```
$ cd $V2PRO/platforms/pkt_proc
$ <text editor> flow.cfg
```

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

Change the line "PPC\_MODEL = swift" to "PPC\_MODEL=bfm". After editing the **flow.cfg** file, save it.

2. Invoke the design flow engine to start the simulation:  

```
$ make func_sim
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **pkt\_proc** → **Make Functional Simulation**)

This script invokes the BFC from the IBM CoreConnect toolkit on the **.bfl** script in **\$V2PRO/platforms/pkt\_proc/sim/bfl**. This generates a Verilog command file that is "compiled in" by the Verilog simulator to control the processor block PLB and DCR BFM masters. The simulator tool is then launched.

3. Run the simulation:

In the MTI window, type **run -all**. For Verilog XL, the simulation begins immediately.

The simulation stops when the BFL script is finished executing or when an error occurs. If the simulation completes successfully, the simulator displays the following message after about 4  $\mu$ s:

**Synch 31 received... Simulation Completed**

If an error is detected, either in the form of a protocol violation reported by a bus monitor or a read comparison error, the simulation stops and an error message is displayed.

## Instructions for Synthesizing the Design

The PPC405 Packet Processing Engine Reference System can be synthesized into FPGA primitive components. This section describes the necessary steps for synthesizing the design. It assumes the Virtex-II Pro Design Kit is properly installed and that the **\$V2PRO/platforms/pkt\_proc/flow.cfg** file has been properly edited to correctly call the user's synthesis tool.

1. Invoke the design flow engine to start the synthesis:  

```
$ cd $V2PRO/platforms/pkt_proc
$ make synth
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **pkt\_proc** → **Make Synthesis**.)

This script creates a project file for the synthesis tool and invokes it. For Synplify, the GUI is launched.

2. If using Synplify, click the Compile button to start the synthesis process.

**Note:** Any changes saved to the Synplify project file (**top.prj**) will be overwritten the next time "make synth" is called. You should transfer any changes to the project file to the **synplify.opt** file.

3. After completion, close the Synplify GUI.

You can edit the **\$V2PRO/platforms/pkt\_proc/syn/synplify.opt** file to add or modify any of the synthesis options. You can also edit the **\$V2PRO/platforms/pkt\_proc/flow.cfg** file to change the target part. (The default part is xc2vp4ff672-7).



## Instructions for FPGA Implementation

After synthesis, the PPC405 Packet Processing Engine Reference System can be targeted into an FPGA using the Xilinx implementation tools. This section describes the necessary steps for implementing the design. It assumes the Xilinx ISE Software is properly installed.

1. Invoke the design flow engine to start the implementation process:

```
$ cd $V2PRO/platforms/pkt_proc
$ make fpga
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **pkt\_proc** → **Make FPGA**)

The script then copies the synthesis output files to the **\$V2PRO/platforms/pkt\_proc/pkt\_proc/par/v2p** directory and runs through the implementation tools (**ngdbuild**, **map**, **par**, **trce**). Any necessary **.edf** netlists for devices in the system are also copied in for **ngdbuild** to process. The timing constraints used by the implementation tools are copied from **\$V2PRO/platforms/pkt\_proc/par/top.ucf** along with any BRAM initialization commands passed in from **data2bram**.

2. With a successfully routed design, you can generate a bitstream:

```
$ cd $V2PRO/platforms/pkt_proc
$ make bit
```

This generates a bitstream for the targeted Virtex-II Pro device.

## Instructions for Back-Annotated Timing Simulation

A placed and routed FPGA design can be simulated with full timing back-annotation. This section describes the necessary steps for simulating the back-annotated design using BFM or SWIFT models of the processor block. In general, the steps are identical to those of a functional simulation except that the "make ba\_sim" command is used.

### SWIFT Simulations

The steps to run a SWIFT simulation are described below:

1. Go to the main directory of the design and edit the **flow.cfg** to select the software application you wish to run:

```
$ cd $V2PRO/platforms/pkt_proc
$ <text editor> flow.cfg
```

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

Edit the **SW\_MAKE** and **SW\_ELF** lines for the program you wish to run and make sure that **PPC\_MODEL = swift**. After editing the **flow.cfg** file, save it.

2. Invoke the design flow engine to start the simulation:

```
$ make ba_sim
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **pkt\_proc** → **Make Timing Simulation**)

The flow engine compiles/links the software, runs **data2bram**, and invokes the Xilinx tools for back-annotation (**ngdanno**, **ngd2ver**). It then launches the simulator tool.

3. Run the simulation:

In the MTI window, type **run -all**. For VerilogXL, the simulation begins immediately.

**Troubleshooting Hint:** Edit the back annotated simulation file (**\$V2PRO/platforms/pkt\_proc/sim/ba\_sim/top.v**) and comment out the **\$sdf\_annotate** command. This causes the simulation to ignore timing, but allows a functional test on the routed FPGA design.



## BFM Simulations

The steps to run a BFM simulation are described below:

1. To run a BFM simulation, edit the **flow.cfg** file to change the processor block model to BFM:  

```
$ cd $V2PRO/platforms/pkt_proc
$ <text editor> flow.cfg
```

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

Change the line "PPC\_MODEL = swift" to "PPC\_MODEL=bfm". After editing the **flow.cfg** file, save it.

2. Invoke the design flow engine to start the simulation:  

```
$ make ba_sim
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **pkt\_proc** → **Make Timing Simulation**)

This script runs the Xilinx tools for back-annotation (**ngdanno**, **ngd2ver**) and then invokes the BFC on the **.bfl** test script. It then launches the simulator tool.

3. Run the simulation:

In the MTI window, type **run -all**. For Verilog XL, the simulation begins immediately. The test ends successfully when the following is returned:

```
Synch 31 received... Simulation Completed
```

## Memory Map

The processor memory map consists of instruction-side OCM and data-side OCM attached memory. Each OCM memory controller supports a maximum of 16 MB of memory. The design contains 4 KB of instruction memory and 8 KB of data memory. **Table 3-18** illustrates the memory map assignments and boundaries.

**NOTE:** The instruction/data OCM must be in a non-cacheable region. In real mode, cacheability controls are on 128 MB regions. Any memory-mapped I/O added to the design that needs cacheable memory would have to be in another region of memory.

**Table 3-18: Processor Memory Map**

| Processor Bus Interface | Device   | Size               | Lower Boundary (hex) | Upper Boundary (hex) |
|-------------------------|----------|--------------------|----------------------|----------------------|
| ISOCM                   | ISBRAM   | 4 KB               | FFFFFF000            | FFFFFFFF             |
| ISOCM                   | Reserved | 16 MB <sup>1</sup> | FF000000             | FFFEFFFF             |
| VOID                    | Unused   |                    | F5000000             | FEFFFFFF             |
| DSOCM                   | Reserved | 16 MB <sup>2</sup> | F4002000             | F4FFFFFF             |
| DSOCM                   | DSBRAM   | 8 KB               | F4000000             | F4001FFF             |
| VOID                    | Unused   |                    | 00000000             | F3FFFFFF             |

1. The reserved size for ISOCM is 16 MB minus the 4 KB used for this application.

2. The reserved size for DSOCM is 16 MB minus the 8 KB used for this application.

## Extending or Modifying the Design

See the *Embedded PPC405 Reference System* document for information about extending and/or modifying the design and modifying the software.

## File Structure for PPE Reference System

```

\-----platforms
|
| \-----pkt_proc
| |
| | \-----par
| | |
| | | \-----v2p
| | |
| | | \-----sim
| | | |
| | | | \-----ba_sim
| | | | \-----bfl
| | | | \-----func_sim
| | | | \-----testbench
| | | | |
| | | | | \-----verilog
| | | | | \-----vhdl
| | |
| | | \-----syn
| | | |
| | | | \-----v2p
| | |
| | | \-----sys
| | | |
| | | | \-----mem
| | | | \-----verilog
| | | | \-----vhdl
|
| \-----source
| |
| | \-----hw
| | |
| | | \-----verilog
| | | |
| | | | \-----pkt_proc
| | |
| | | \-----sim_models
| | | |
| | | | \-----Xilinx
| | | | |
| | | | | \-----proc_block_BFM
| | | | | \-----verilog
| | | | | \-----smartmodel
| | |
| | | \-----sw
| | | |
| | | | \-----apps
| | | | |
| | | | | \-----pkt_proc
| | | | |
| | | | | \-----libs
| | | | | |
| | | | | | \-----bsp
| | | | | |
| | | | | | \-----mapfiles
|
| \-----tools
| |
| | \-----cygwin
| | |
| | | \-----xilinx
| | | |
| | | | \-----bin
| | | | \-----data
| | |
| | | \-----solaris
| | | |
| | | | \-----xilinx
| | | | |
| | | | | \-----bin
| | | | | \-----data

```



# *Embedded PPC405 Reference System*

---

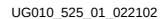
## Introduction

The Embedded PPC405 Reference System is an example of a large Virtex-II Pro™ based system. An IBM Core Connect™ infrastructure connects the CPU to numerous peripherals using Processor Local Bus (PLB), On-Chip Peripheral Bus (OPB), and Device Control Register (DCR) buses to build a complete system. This document describes the contents of the reference system and provides information about how the system is organized and implemented. It also discusses verification methodologies including software-driven and bus-model-driven simulations. A complete design cycle incorporating simulation, synthesis, and FPGA implementation is described. The information presented introduces many aspects of the Embedded PPC405 Reference System, but the user should refer to additional specific documentation for more detailed information about the software, tools, peripherals, interface protocols, and capabilities of the FPGA.

## Hardware

### Overview

**Figure 4-1** provides a high-level view of the hardware contents of the Embedded PPC405 System. This design demonstrates a system that uses PLB, OPB, and DCR devices. The PLB protocol generally supports higher bandwidths, so the memory devices are placed there. The OPB connects the peripheral devices to the CPU by way of a PLB-to-OPB Bridge and primarily is intended for lower performance devices. The OPB offers a less complex protocol relative to the PLB, making it easier to design peripherals that do not require high performance. The OPB also has the advantage that it can support a greater number of devices, and it acts to decouple the peripherals from the higher-speed memory devices. DCR is used with control and status registers for simplicity when performance is not important. Refer to the PLB, OPB, and DCR CoreConnect Architecture Specifications for more information.



## Processor Local Bus (PLB)

The PLB devices in the reference system include:

- PLB Masters
  - CPU Instruction-Side PLB Interface (Master ID = 0)
  - CPU Data-Side PLB Interface (Master ID = 1)
  - OPB-to-PLB Bridge (Master ID = 2)
  - 640x480 VGA TFT LCD Controller (Master ID = 3)
- PLB Slaves
  - BRAM Controller
  - Double Data Rate (DDR) SDRAM Controller

- SRAM/Flash Controller
- PLB-to-OPB Bridge-Out
- Zero Bus Turnaround (ZBT) SRAM Controller
- PLB Arbiter
  - 8 Master, 64-bit Xilinx PLB Arbiter
- PLB Bus Logic
  - The specification for the PLB protocol requires that some additional logic be included to OR together the outputs of the slaves to create the PLB slave bus. This sample system supports up to nine PLB slave devices, but can easily be expanded to support additional slaves.

In general, all PLB devices are optimized around the Virtex-II Pro architecture and make use of pipelining to improve maximum clock frequencies and reduce logic utilization. Refer to the accompanying documentation for each device for more information about its design.

## On-Chip Peripheral Bus (OPB)

The OPB connects lower-performance peripheral devices to the system. The OPB has a less complex architecture, which simplifies peripheral development. A PLB-to-OPB Bridge translates PLB transactions into OPB transactions, allowing the CPU to access the OPB devices. OPB devices can also access PLB devices by way of an OPB-to-PLB Bridge.

The OPB devices in the reference system include:

- OPB Masters
  - PLB-to-OPB Bridge-Out
  - Ethernet Controller (DMA Engine)
- OPB Slaves
  - OPB Arbiter (Configuration Registers)
  - IIC Controller
  - General-Purpose Input/Output (GPIO)
  - 16450 UART
  - 16550 UART
  - Configuration ROM
  - PCI Bus Master
  - LCD Controller
  - OPB-to-DCR Bridge
  - OPB-to-PLB Bridge-In
  - PS/2 (two instances)
  - Touch Screen Digitizer
  - Ethernet Controller\*

\*The embedded reference system design contains a placeholder for the Xilinx Ethernet Controller. Contact your local sales office for information about obtaining this core.

- OPB Arbiter
  - 4 Master, 32-bit Xilinx OPB Arbiter
- OPB Bus Logic
  - The OPB protocol specification requires that some additional logic be included to AND/OR together the outputs of the slaves to create the OPB slave bus. This sample system supports up to 17 OPB slave devices, but can easily be expanded to support additional slaves.

In general, all OPB devices are optimized around the Virtex-II Pro architecture and make use of pipelining to improve maximum clock frequencies and reduce logic utilization. Refer to the accompanying documentation for each device for more information about its design.

The OPB devices in the reference design make use of Intellectual Property InterFace (IPIF) modules to further simplify IP development. The IPIF converts the OPB protocol into common interfaces, such as an SRAM protocol or a control register interface. IPIF modules also provide support for DMA and interrupt functionality. IPIF modules simplify software development since the IPIF framework has many common features and supports a discovery process using a Configuration ROM (CROM). The CROM contains information about the devices in the system (such as address mapping, device capabilities, and revisions) that software can read at boot-up to configure its device drivers. Refer to [Volume 7, Chapter 3, IPIF Specification](#) for more information.

Note that the IPIF is designed mainly to support a wide variety of common interfaces, but may not be the optimal solution in all cases. Where additional performance or functionality is required, the user can develop a custom OPB interface. The IPIF protocols can also be extended to support other bus standards, such as PLB. This allows the backend interface to the IP to remain the same while the bus interface logic in the IPIF is changed. This provides an efficient means for supporting different bus standards with the same IP device.

The OPB specification supports masters and slaves of up to 64 bits with a *dynamic bus sizing* capability that allows OPB masters and slaves of different sizes to communicate with each other. The Embedded PPC405 Reference System uses a subset of the OPB specification which only supports 32-bit byte enable masters and slaves. Legacy devices utilizing 8- or 16-bit interfaces or those that require dynamic bus sizing functionality are not directly supported. For systems requiring legacy support, an OPB interface module is available to convert OPB byte enable cycles into dynamic bus sizing transactions. It is recommended that all new OPB peripherals support byte enable operations for better performance and reduced logic utilization.

## Device Control Registers (DCR)

The DCR offers a very simple interface protocol and is used for accessing control and status registers in various devices. It allows for register access to various devices without loading down the OPB and PLB interfaces. Since DCR devices are generally accessed infrequently and do not have high-performance requirements, they are used throughout the reference design for functions, such as error status registers, interrupt controllers, and device initialization logic.

The CPU contains a DCR master interface that is accessed through special *Move To DCR* and *Move From DCR* instructions. Some users may prefer to see DCR registers memory mapped so an alternative DCR Master is provided. An OPB-to-DCR Bridge can be instantiated to locate the 4-KB DCR space within the general system memory space. The reference design demonstrates both methods for accessing DCR.

The DCR slave devices connected to the CPU's DCR port include:

- Packet Processing Engine (Instance connected to OCM)
- Data-Side OCM (8 KB)
- Instruction-Side OCM (4 KB)

The DCR slave devices connected to the OPB-to-DCR Bridge include:

- Critical Interrupt Controller
- Non-Critical Interrupt Controller
- PLB Arbiter
- PLB-to-OPB Bridge
- OPB-to-PLB Bridge

- Packet Processing Engine (Instance connected to PLB BRAM Controller)

The DCR specification requires that the DCR master and slave clocks be synchronous to each other and related in frequency by an integer multiple. It is important to be aware of the clock domains of each of the DCR devices to ensure proper functionality.

## Other Devices

In addition to the PLB, OPB, and DCR devices, the Embedded PPC405 Reference System contains Instruction-Side and Data-Side OCM modules. The OCM consists of BRAMs directly connected to the CPU. They allow the CPU fast access to memory and are useful for providing instructions or data directly to the CPU, bypassing the cache. This can prevent thrashing of caches to better process packet data or execute interrupt service routines. Refer to the OCM documentation for information about applications and design information.

Virtex-II Pro FPGAs contain Rocket I/O interfaces capable of gigabit serial data transfer rates. The Embedded PPC405 Reference System instantiates two packet processing engines to demonstrate the use of these I/O capabilities. Each packet processor uses a dual-port BRAM to pass packet data to and from the CPU. It also has a DCR port to control it from. To illustrate the flexibility that is possible, one packet processor is directly connected to the CPU's DCR interface and OCM. The other packet processor is tied to a PLB BRAM Controller and the OPB-to-DCR Bridge.

## Interrupts

The CPU also contains two interrupt pins, one for critical interrupt requests and the other for non-critical interrupts. An interrupt controller for critical and non-critical interrupts is controlled through the DCR. It allows multiple edge or level sensitive interrupts from peripherals to be OR'ed together back to the CPU. The ability for bitwise masking of individual interrupts is also provided. [Table 4-1](#) summarizes the connections from the IP to the interrupt controllers.

**Table 4-1: Connections from the IP to the Interrupt Controllers**

| Interrupt Controller | Interrupt Input Line | Description            |
|----------------------|----------------------|------------------------|
| Non-Critical INTC    | 0 (MSB)              | Uart 16450             |
|                      | 1                    | Uart 16550             |
|                      | 2                    | IIC                    |
|                      | 3                    | Ethernet               |
|                      | 4                    | PS/2 Port #1           |
|                      | 5                    | PS/2 Port #2           |
|                      | 6                    | Touch Screen Digitizer |
|                      | 7-31 (LSB)           | Unassigned             |
| Critical INTC        | 0 (MSB)              | PLB-to-OPB Bridge      |
|                      | 1                    | OPB-to-PLB Bridge      |
|                      | 2-31 (LSB)           | Unassigned             |

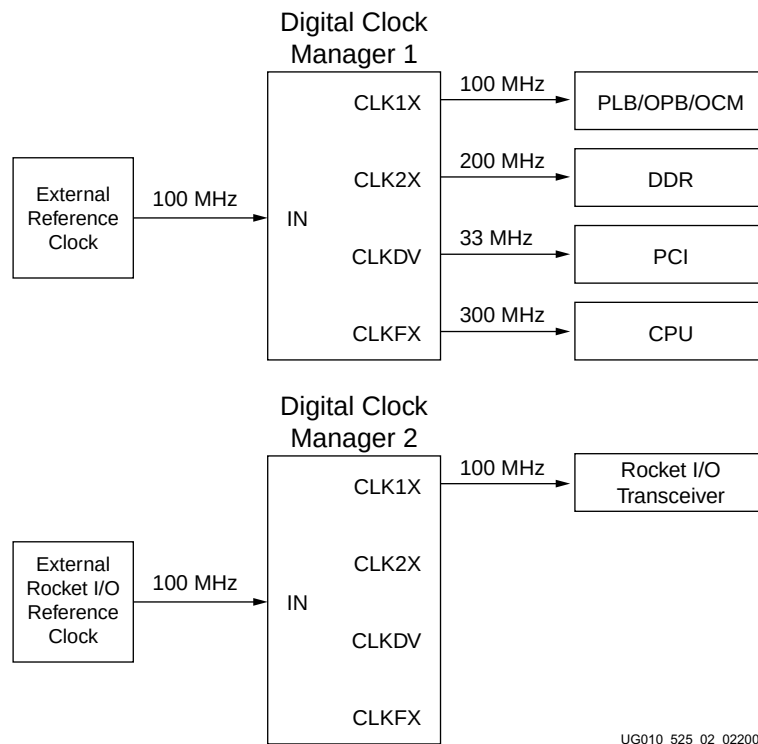
## Clock/Reset Distribution

Virtex-II Pro FPGAs have abundant clock management and global clock buffer resources. To demonstrate some of these capabilities, the Embedded PPC405 Reference System uses a variety of different clocks. [Figure 4-2](#) illustrates use of the Digital Clock Managers (DCMs)



for generating clocks in the design. A 100 MHz input reference clock is used to generate the main 100 MHz PLB, OPB, and OCM clocks. A 200 MHz clock is also generated to support the DDR SDRAM module. The main 100 MHz clock is divided down by three to create a 33 MHz PCI clock. The CPU clock is multiplied up from the PLB clock to 300 MHz. Since each clock is generated from the same 100 MHz input, they are all phase aligned to each other. This synchronous phase alignment is required by the CPU and many other devices so they can pass signals from one clock domain to another.

The CPU clock can run at any integer multiple of the PLB clock up to the maximum CPU clock frequency. During reset, internal clock synchronizers in the CPU detect the phase alignment of the PLB and CPU clocks and adjust for it automatically. The OCM clock must be divided down from the CPU clock by an integer multiple (up to eight), and the two clocks must be synchronous to each other. A second DCM acts independently to drive the clocks in the Rocket I/O Transceiver and Packet Processing Engine modules. The Rocket I/O transceiver clock is set to 100 MHz to yield a 2-GHz serial output rate.



UG010\_525\_02\_022002

Figure 4-2: Clock Generation

After a system reset or at FPGA startup, a debounce circuit holds the FPGA in reset until the DCM has locked onto its reference clock. Once the DCM is locked and the clocks remain stable for several cycles (may take several microseconds in simulation), the reset condition is released to allow the system logic to begin operating. (For example, the CPU will begin fetching instructions a few cycles after reset is released.) Since the reset net is a high-fanout signal, it may not be able to reach all the logic in the design within one clock cycle. User IP blocks should be designed to take into account the possible skew in the global reset and still start up properly. Alternatively, the global reset can be registered locally in each IP block to generate a synchronous reset signal.

The design implements the three levels of reset supported by the PPC405:

- Core reset
- Chip reset
- System reset

The core reset only affects the processor while the chip reset clears all the logic on the FPGA. The system reset is designed to reset the entire system including the FPGA and external devices connected to the FPGA. The CPU provides an internal special-purpose register that allows software to request that one of the three resets be performed.

Figure 4-3 shows how the reset logic in the Embedded PPC405 Reference System is implemented. The chip and core reset request lines from the CPU are driven to the corresponding reset nets through shift registers to ensure that minimum pulse width requirements are met. The system reset request line from the CPU drives an external pin as an open-drain signal (with resistive pullup) so that other external devices can sense or drive the system reset. Note that the system reset has the effect of resetting the DCMs in the FPGA, while the chip reset does not affect the DCMs. The reset logic in the Embedded PPC405 Reference System is an example implementation of the PPC405 reset architecture. Designers should set the scope, boundaries, and effects of resets as appropriate to their designs.

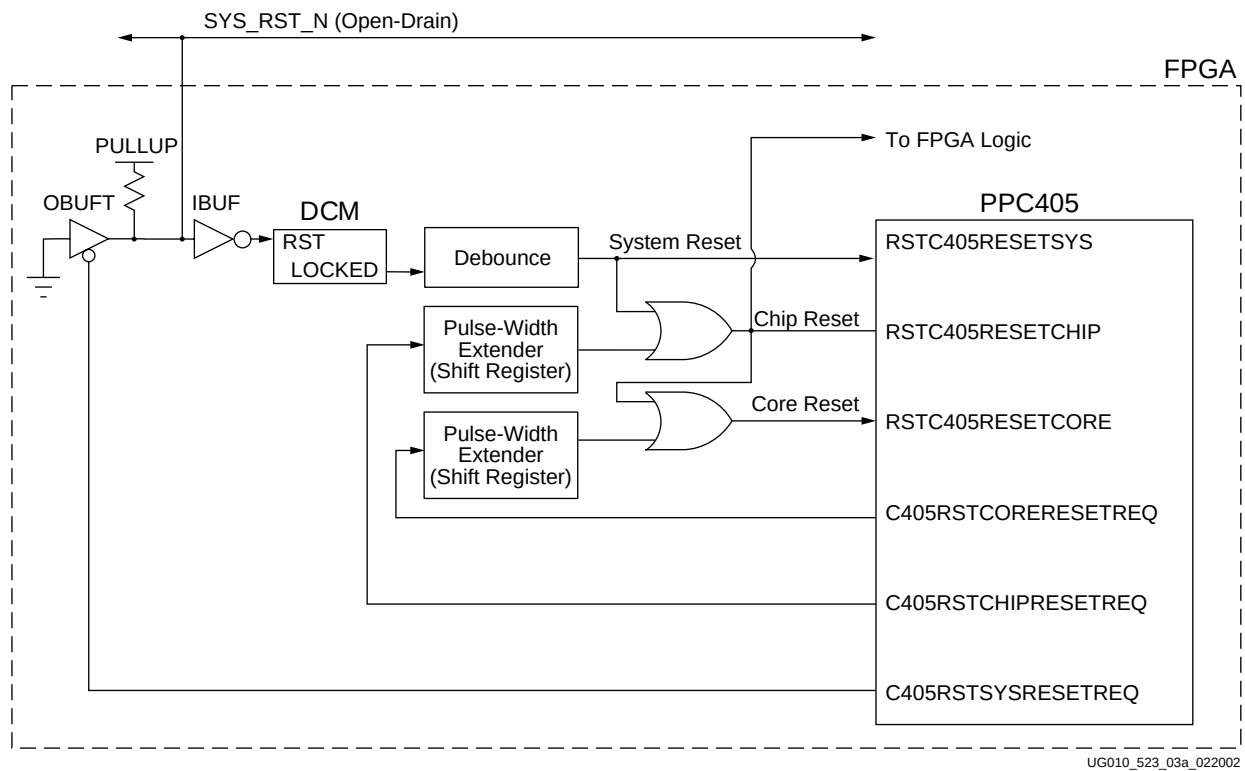


Figure 4-3: Embedded Reference System Reset Logic

## HDL Organization

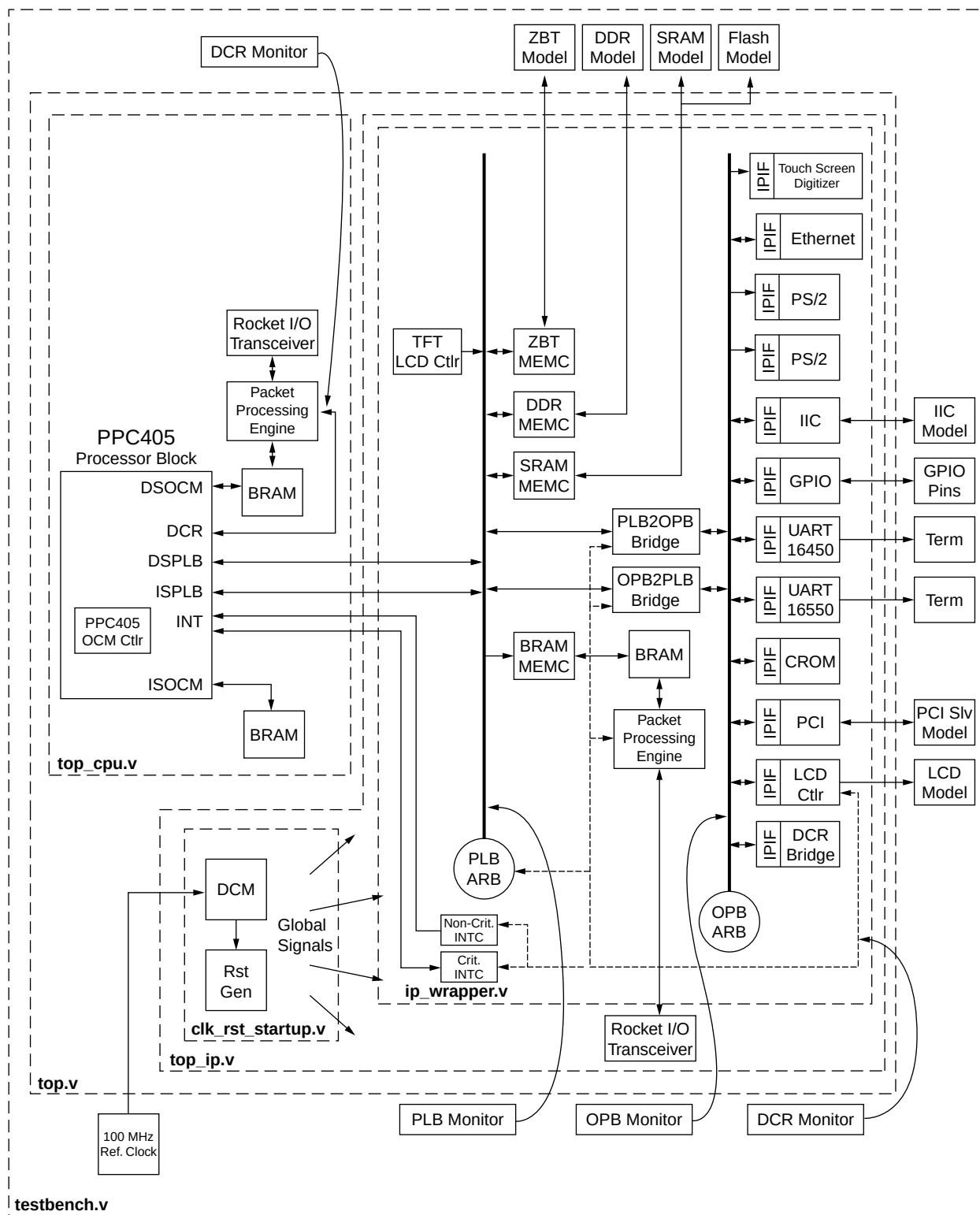
**Figure 4-4** diagrams the organization of the higher-level HDL files that comprise the system and testbench environment.

All of the peripherals, the memory controllers, and the CoreConnect infrastructure are instantiated in the **ip\_wrapper.v** file. The external I/O signals, the clock/reset connections, and the processor block connections are propagated up to the next level file, **top\_ip.v**. The **top\_ip.v** file instantiates the clock/reset logic, one of the Rocket I/O transceivers and the **ip\_wrapper.v** file. Another file called **top\_cpu.v** contains the processor block, OCM BRAM, and OCM connected packet processor logic. Finally, the top level design file **top.v** combines together **top\_ip.v** and **top\_cpu.v**. The module ports of **top.v** represent the external I/O signals of the FPGA. Therefore, all files in the hierarchy from the **top.v** level and below make up the Virtex-II Pro FPGA design that is synthesized and routed into an FPGA.

For simulation, the file **testbench.v** instantiates the FPGA as the device under test and includes behavioral models for the FPGA to interact with. In addition to behavioral models for memory devices, clock oscillators, and external peripherals, the testbench also instantiates the CoreConnect bus monitors to observe the PLB, OPB, and DCR buses for protocol violations. The testbench can also preload some of the memories in the system for purposes such as loading software for the CPU to execute. The **sim\_params.v** file is designed to be modified by the user to customize various simulation options. These options include message display options, maximum simulation time, and clock frequency. The user should edit this file to reflect personal simulation preferences.

An additional HDL file, called **global\_params.v**, defines the global constants and parameters used throughout the reference design. This information may include, memory map definitions, device configuration values, or any other information that is better organized in a global context.

Some of the testbench code is used to access signals internal to the design using hierarchy path names to reach into the design without changing any of the port interfaces. It is important that the design source files used for simulation match the source files for synthesis. Therefore, port interfaces should not be different or else inconsistencies can result.



UG010\_525\_03\_022102

Figure 4-4: Organization of Higher-Level HDL Files

## Simulation and Verification

### SWIFT and BFM CPU Models

The Embedded PPC405 Reference System demonstrates two different simulation methods to help verify designs using the PPC405 CPU. One method uses a full simulation model of the CPU based on the actual silicon. The second method employs bus functional models (BFMs) to generate processor bus cycles from a command scripting language. These two methods offer different trade-offs between behavior in real hardware, ease of generating bus cycles, and the amount of real time to simulate a given clock cycle.

A SWIFT model can be used to simulate the CPU executing software instructions. In this scenario, the executable binary images of the software are preloaded into memory from which the CPU can boot up and run the code. Though this is a relatively slow way to exercise the design, it more accurately reflects the actual behavior of the system.

The SWIFT model is most useful for helping to bring up software and for correlating behavior in real hardware with simulation results. The Embedded PPC405 Reference System demonstrates the SWIFT model simulation flow, by allowing the user to write a C program that is compiled into an executable binary file. This executable (in ELF format) is then converted into BRAM initialization commands using a tool called Data2BRAM. (Note that Data2BRAM can also generate memory files for the Verilog command **readmemh** to initialize other memories such as SRAM or DDR memory.)

When a simulation begins and reset is released, the CPU SWIFT model fetches the instructions from BRAM (which is mapped to the boot vector) and begins running the program. The user can then observe the bus cycles generated by the CPU or any other signal in the design. For debugging purposes, the values of the CPU's internal program counter, general-purpose registers, and special-purpose registers are available for display during simulation.

Generating a desired sequence of bus operations from the CPU may require a lot of software setup or simulation time. For early hardware bring-up or IP development, a bus functional model can be used to speed up simulation cycles and avoid having to write software. A model of the CPU is available in which two PLB master BFMs and one DCR BFM are instantiated to drive the CPU's PLB/DCR ports. These BFMs are provided in the CoreConnect toolkits and allow the user to generate bus operations by writing a script written in the Bus Functional Language (BFL). The Embedded PPC405 Reference System provides a sample BFL script that exercises many of the peripherals in the system. Refer to the *CoreConnect Toolkit* documentation for more information.

Since the CPU SWIFT model and BFM model both have the same set of port interfaces, users can switch between the two simulation methods by compiling the appropriate set of files without having to modify the system's design source files. Users may, however, need to modify their testbenches to take into account which model is being used.

### Behavioral Models

The Embedded PPC405 Reference System includes some behavioral models to help exercise the devices and peripherals in the FPGA. Many of these models are freely available from various manufacturers and include interface protocol-checking features. The behavioral models and features included in the reference design are:

- ZBT, DDR SDRAM, Flash, and SRAM memory models for testing the memory controllers:
  - These models can also be preloaded with data for simulations
- EEPROM model with IIC interface
- Pull-ups connected to the GPIO for reading and driving outputs without getting unknown values
- Terminal interface connected to the UARTs for sending and receiving serial data

- The terminal allows a user to interact with the simulation in real time
- Characters sent out by the UARTs are displayed on a terminal while characters typed into the terminal program are serialized and sent to the UARTs
- A simple file I/O mechanism passes data between the hardware simulator and the terminal program
- Simple PCI Slave that responds to commands from the PCI Master in the reference design
  - The PCI Slave acts as a memory device that the PCI Master can write to and read back from
  - The PCI Slave responds to configuration, memory, and I/O PCI cycles
- LCD model that implements some registers that the LCD controller can write to and read back from
  - It does not decode the control register commands to display the characters sent to the LCD
  - A terminal program similar to the UART terminal is provided to display characters written to the LCD
- Two Rocket I/O packet processor modules that are connected to each other for passing packets back and forth between them

## Synthesis and Implementation

The Embedded PPC405 Reference System can be synthesized and placed/routed into a Virtex-II Pro FPGA. A basic set of timing constraints for the design is provided to allow the design to go through place and route. Note that some peripherals in the design are pre-synthesized and provided only in netlist format (.edf, .edn, .ngc, .ngo). For these devices, a corresponding black box has been instantiated in the source code to force the synthesis tool to leave a placeholder for the netlist. After a successful place and route, it is possible to run a simulation of the design using a back-annotated timing model of the FPGA with a SWIFT or BFM version of the processor block.

## Design Flow Environment

A flow engine provides an environment to help manage the design flow for the Embedded PPC405 Reference System. This engine uses the utility program called **make** and a set of PERL scripts to allow the user to perform tasks, such as running simulations, synthesizing a design, or implementing it on an FPGA with a simple set of commands. The design flow tool is implemented with a generic architecture to allow it to be adapted for use on a variety of different designs. For more information, refer to *Flow Documentation*.

## File Listings

The files and directories specific to the Embedded PPC405 Reference System are located below the **\$V2PRO/platforms/embedded** directory, shown in [Figure 4-5](#), and listed in the tables that follow. Source files for the IP modules and software applications are kept in the **\$V2PRO/source** directory. The **\$V2PRO/source** directory generally contains files that can be shared by different designs. The **File Structure for Embedded PPC405 Reference System** section diagrams the full file directory structure for this design. Refer to the appropriate instructions for installation and setup information. Note that the following tables only list files in the **\$V2PRO/platforms/embedded** directory that are present after installation. After running simulation, synthesis, or place and route, additional files may be created. Directory path names are shown separated by the "/" character as is the UNIX convention. For Windows, the "\" should be used to separate directory paths.

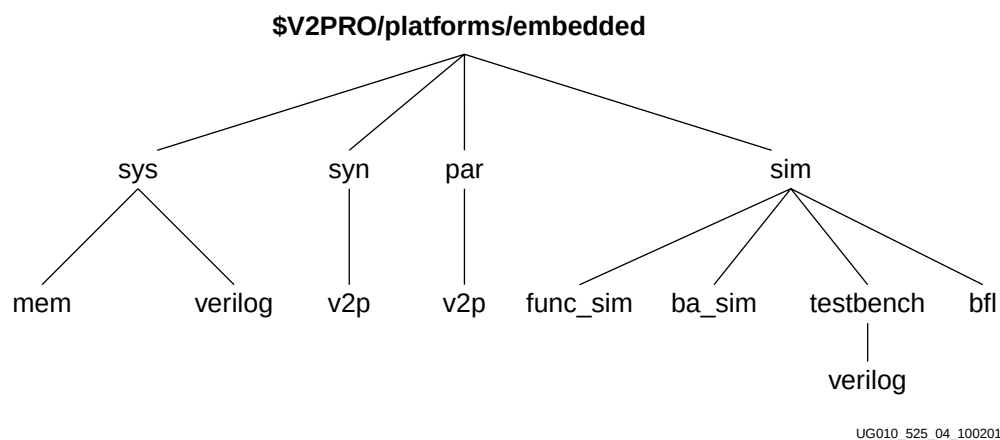


Figure 4-5: Map of Directories Below \$V2PRO/platforms/embedded

Table 4-2: Files in the \$V2PRO/platforms/embedded Directory

| File     | Description                                                                                          |
|----------|------------------------------------------------------------------------------------------------------|
| Makefile | Redirection file to allow the centralized design flow scripts to be invoked from different locations |
| flow.cfg | File used to configure the design flow scripts                                                       |

Table 4-3: Files in the \$V2PRO/platforms/embedded/sys Directory

| File            | Description                                                       |
|-----------------|-------------------------------------------------------------------|
| memory_init.bmm | Data2BRAM configuration file describing the system memory devices |

Table 4-4: Files in the \$V2PRO/platforms/embedded/sys/verilog Directory

| File              | Description                                                                                                                                        |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| src.lst           | List of source files for the design                                                                                                                |
| bram_block.v      | Wrapper holding PLB connected block RAMs                                                                                                           |
| clk_rst_startup.v | Clock, reset, startup logic for the design                                                                                                         |
| global_params.v   | Centralized location for the global parameters used within the design                                                                              |
| ip_wrapper.v      | Wrapper file around all the IP in the design. Contains PLB, OPB, and DCR as well the interrupt controller and PLB-connected packet processor logic |
| opb_bus_logic.v   | OR logic for OPB to connect OPB master, slave, and arbiter devices together                                                                        |
| plb_bus_logic.v   | OR logic for PLB to connect PLB slaves to the PLB arbiter                                                                                          |

Table 4-4: Files in the \$V2PRO/platforms/embedded/sys/verilog Directory

| File      | Description                                                                 |
|-----------|-----------------------------------------------------------------------------|
| top.v     | Top-level file for Virtex-II Pro FPGA                                       |
| top_cpu.v | Instantiates processor block, OCM, and OCM-connected packet processor block |
| top_ip.v  | Instantiates IP wrapper file and clock, reset, startup logic                |

Table 4-5: Files in the \$V2PRO/platforms/embedded/sys/mem Directory

| File       | Description                                |
|------------|--------------------------------------------|
| ddr*.mem   | Memory initialization data files for DDR   |
| flash*.mem | Memory initialization data files for Flash |
| sram*.mem  | Memory initialization data files for SRAM  |
| zbt*.mem   | Memory initialization data files for ZBT   |

Table 4-6: Files in the \$V2PRO/platforms/embedded/sim/func\_sim Directory

| File        | Description                                                                              |
|-------------|------------------------------------------------------------------------------------------|
| Uart1.input | Character to be sent to UART 1 from the terminal (set first line to "@0 0" for no input) |
| Uart2.input | Character to be sent to UART 2 from the terminal (set first line to "@0 0" for no input) |
| wave.do     | MTI command to display a sample set of interesting signals for functional simulation     |

Table 4-7: Files in the \$V2PRO/platforms/embedded/sim/bfl Directory

| File                        | Description                                                                                     |
|-----------------------------|-------------------------------------------------------------------------------------------------|
| test_emb_ref_des.bfl        | Sample BFL script to exercise the devices in the system during functional simulation            |
| test_emb_ref_des_ba_sim.bfl | Sample BFL script to exercise the devices in the system during back-annotated timing simulation |



Table 4-8: Files in the \$V2PRO/platforms/embedded/sim/ba\_sim Directory

| File        | Description                                                                                     |
|-------------|-------------------------------------------------------------------------------------------------|
| Uart1.input | Character to be sent to UART 1 from the terminal (set first line to "@0 0" for no input)        |
| Uart2.input | Character to be sent to UART 2 from the terminal (set first line to "@0 0" for no input)        |
| wave.do     | MTI command to display a sample set of interesting signals for back-annotated timing simulation |

Table 4-9: Files in the \$V2PRO/platforms/embedded/sim/testbench/verilog Directory

| File                 | Description                                                                                            |
|----------------------|--------------------------------------------------------------------------------------------------------|
| empty.dat            | Dummy (empty) data file used by Flash model. Note that Flash data is explicitly preloaded by testbench |
| flash_BlockLock.dat  | Flash model data file. Contains block lock information                                                 |
| flash_model_cfg.v    | Configuration file to setup Flash model for proper operation                                           |
| flash_querytable.dat | Flash model data file. Contains status and ID register information                                     |
| opb_dcl.inc          | OPB bus monitor configuration file set for 17 OPB devices                                              |
| sim_params.v         | Centralized location for parameters and settings that affect simulation                                |
| testbench.v          | Testbench source file                                                                                  |
| view_reg.v           | Behavioral code to display internal CPU register contents whenever they are updated                    |

Table 4-10: Files in the \$V2PRO/platforms/embedded/par Directory

| File    | Description                                             |
|---------|---------------------------------------------------------|
| top.ucf | User constraints file used by FPGA implementation tools |

Table 4-11: Files in the \$V2PRO/platforms/embedded/syn Directory

| File         | Description                                 |
|--------------|---------------------------------------------|
| synplify.opt | Optional synthesis project file information |

## Instructions for Running Functional Simulations

The Embedded PPC405 Reference System comes with SWIFT and BFM based simulation examples. This section describes the necessary steps for running these simulations using functional timing with the original source code. It assumes the Virtex-II Pro Design Kit is properly installed and that the `$V2PRO/platforms/embedded/flow.cfg` file is properly edited to correctly call the user's simulation tools. The user may also modify the `sim_params.v` file to customize various simulation parameters.

### SWIFT Simulations

The steps to run a SWIFT simulation are described below:

1. Go to the main directory of the design and edit the `flow.cfg` to select the software application you wish to run:

```
$ cd $V2PRO/platforms/embedded
$ <text editor> flow.cfg
```

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

The `<text editor>` can be a program like Emacs, vi, or WordPad. In the `flow.cfg` file, uncomment the `SW_MAKE` and `SW_ELF` lines for the program you wish to run and leave the other programs commented out. Table 4-12 lists the programs available. After installation, the `flow.cfg` file is set to the `hello_uart` program, which is precompiled and ready to run. Also, make sure that the `PPC_MODEL` entry is set to "swift". After editing the `flow.cfg` file, save it.

2. Invoke the design flow engine to start the simulation:

```
$ make func_sim
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Embedded** → **Make Functional Simulation**)

The flow engine begins by compiling and linking the application software (as necessary) to generate a binary executable program file (in `.elf` format). The `.elf` file is copied locally, then `data2bram` is invoked to generate the necessary BRAM initialization command and `.mem` files that, in turn, initialize the DDR, SRAM, FLASH, and ZBT memories. The simulator tool is then launched along with terminals for the UARTs and LCD.

**Note:** For MTI, the scripts will copy a corresponding NT or Solaris version of the `modelsim.ini` configuration file into the local directory. This file can be found in the `$V2PRO/tools/$V2PRO_HOST_OSTYPE/xilinx/data/modelsim.ini` directory.

3. Run the simulation:

In the MTI window, type `run -all`. For the `hello_uart` and `hello` programs, you can interact with the simulation using the terminal program. On the terminal, the words "Hello world" followed by a carriage return will appear as the program executes in the simulation. You can also type into the terminal (after **Hello world** is displayed) and have the characters echoed back after some delay. Keep in mind that the program is running in simulation, so it may take many seconds of real time for the characters to get transmitted and received. Pressing the **Esc** (escape) key will exit the terminal and stop the simulation after a few seconds.

For Verilog XL, the simulation begins immediately. You can interact with the terminal for the `hello_uart` and `hello` programs as described above. If the "define VIRSIM" command in `sim_params.v` is set, the waveforms for all the signals in the design are stored for the `virsim` program to display. You may need to edit the testbench to invoke your own Verilog XL waveform capture and display program.

The `hello_lcd` program sends the necessary software commands to cause the LCD to display **Hello world** on the LCD terminal, while the `hello_gpio` program causes the ASCII characters for **Hello world** to be driven out on the GPIO pins. The `hello_pit`

program uses the programmable interval timer to periodically interrupt the **hello\_uart** program to display **Hello world** on the LCD. The **hello\_iic\_at24cxx** program causes the ASCII characters for **Hello world** to be stored in the EEPROM. Note that no screen output is displayed by **hello\_gpio** or **hello\_iic\_at24cxx**, but the waveforms for these devices can be observed. The **scan\_pci** program looks for PCI devices in the system and reports its findings back to the terminal. Running the **xrom** program displays a menu on the terminal that allows you to choose from a variety of system tests, including memory tests. Refer to the document describing the software applications for more information about these and other test programs.

It is normal to see some warnings from the PLB monitor or behavioral memory models during reset, but the PLB/OPB/DCR monitors should not report any protocol errors during simulation (warnings and notes may occur depending on the circumstances). For programs other than **hello\_uart**, **hello**, **hello\_pit**, or **xrom**, you should stop the program after the amount of time specified in Table 4-12. You can modify the **sim\_params.v** file to stop the simulation at the desired time or press **Ctrl c** (break).

Table 4-12: SWIFT Simulation Programs

| Program Name             | Description                                           | Simulation Duration         |
|--------------------------|-------------------------------------------------------|-----------------------------|
| <b>hello_uart</b>        | Hello world example using the UART                    | Press <Esc> in the terminal |
| <b>hello</b>             | Hello world example using the C library               | Press <Esc> in the terminal |
| <b>hello_lcd</b>         | Hello world example using the LCD                     | 51 $\mu$ s                  |
| <b>hello_gpio</b>        | Hello world example using the GPIO                    | 11 $\mu$ s                  |
| <b>hello_iic_at24cxx</b> | Hello world example using the IIC EEPROM              | 2852 $\mu$ s                |
| <b>hello_pit</b>         | Hello world example using programmable interval timer | Press <Esc> in the terminal |
| <b>scan_pci</b>          | Scans the PCI bus for devices                         | 5451 $\mu$ s                |
| <b>xrom</b>              | Menu-driven system test program                       | Press <Esc> in the terminal |

## BFM Simulations

The steps to run a BFM simulation are described below:

- To run a BFM simulation, edit the **flow.cfg** file to change the processor block model to BFM:  

```
$ cd $V2PRO/platforms/embedded
$ <text editor> flow.cfg
```

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

Change the line "PPC\_MODEL = swift" to "PPC\_MODEL=bfm". After editing the **flow.cfg** file, save it.

- Invoke the design flow engine to start the simulation:  

```
$ make func_sim
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Embedded** → **Make Functional Simulation**)

This script invokes the BFC from the IBM CoreConnect toolkit on the **.bfl** script in **\$V2PRO/platforms/embedded/sim/bfl**. This generates a Verilog command file that is

"compiled in" by the Verilog simulator to control the processor block PLB and DCR BFM masters. The simulator tool is then launched along with terminals for the UARTs and LCD.

3. Run the simulation:

In the MTI window, type **run -all**. For Verilog XL, the simulation begins immediately. The BFL script performs a set of memory write/read tests to all memory devices in the system. It then sends out "Hello" over UART1, "world" over UART 2, and "V2PRO" over the LCD after about 30  $\mu$ s. The script finishes up by accessing the other peripherals in the system, including PCI, IIC, and CROM.

The simulation stops when the BFL script is finished executing or when an error occurs. If the simulation completes successfully, the simulator displays the following message after about 315  $\mu$ s:

**Synch 31 received... Simulation Completed**

If an error is detected, either in the form of a protocol violation reported by a bus monitor or a read comparison error, the simulation stops and an error message is displayed.

It is a useful exercise to view the simulator's waveform display and correlate the commands in the **.bfl** script with the bus transaction waveforms over PLB, OPB, and DCR.

## Instructions for Synthesizing the Design

The Embedded PPC405 Reference System can be synthesized into FPGA primitive components. This section describes the necessary steps for synthesizing the design. It assumes the Virtex-II Pro Design Kit is properly installed and that the **\$V2PRO/platforms/embedded/flow.cfg** file has been properly edited to correctly call the user's synthesis tool.

1. Invoke the design flow engine to start the synthesis:

```
$ cd $V2PRO/platforms/embedded
$ make synth
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Embedded** → **Make Synthesis**.)

This script creates a project file for the synthesis tool and invokes it. For Synplify, the GUI is launched.

2. If using Synplify, click the Run button to start the synthesis process.

**Note:** Any changes saved to the Synplify project file (**top.prj**) will be overwritten the next time "make synth" is called. You should transfer any changes in the project file to the **synplify.opt** file.

3. After completion, close the Synplify GUI.

You can edit the **\$V2PRO/platforms/embedded/syn/synplify.opt** file to add or modify any of the synthesis options. You can also edit the **\$V2PRO/platforms/embedded/flow.cfg** file to change the target part. (The default part is xc2vp20ff896-7).

## Instructions for FPGA Implementation

After synthesis, The Embedded PPC405 Reference System can be targeted into an FPGA using the Xilinx implementation tools. This section describes the necessary steps for implementing the design. It assumes the Xilinx ISE software is properly installed.

1. Invoke the design flow engine to start the implementation process:

```
$ cd $V2PRO/platforms/embedded
```

```
$ make fpga
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Embedded** → **Make FPGA**)

The script then copies the synthesis output files to the

**\$V2PRO/platforms/embedded/par/v2p** directory and runs through the implementation tools (**ngdbuild**, **map**, **par**, **trce**). Any necessary netlists (**.edf**, **.edn**, **.ngc**, **.ngo**) for devices in the system are also copied in for **ngdbuild** to process. The timing constraints used by the implementation tools are copied from **\$V2PRO/platforms/embedded/par/top.ucf** along with any BRAM initialization commands passed in from **data2bram**.

2. With a successfully routed design, you can generate a bitstream:

```
$ cd $V2PRO/platforms/embedded
$ make bit
```

This generates a bitstream for the targeted Virtex-II Pro device. Note that the PCI, UART, and IIC Cores are only evaluation versions with a fixed functionality. They will timeout and become non-functional 2<sup>40</sup> clock cycles after the FPGA is configured. Contact your local Xilinx sales office for information about obtaining complete, parameterizeable versions of these and other cores.

## Instructions for Back-Annotated Timing Simulation

A placed and routed FPGA design can be simulated with full timing back-annotation. This section describes the necessary steps for simulating the back-annotated design using BFM or SWIFT models of the processor block. In general, the steps are identical to those of a functional simulation except that the "make ba\_sim" command is used.

### SWIFT Simulations

The steps to run a SWIFT simulation are described below:

1. Go to the main directory of the design and edit the **flow.cfg** to set the processor block model:

```
$ cd $V2PRO/platforms/embedded
$ <text editor> flow.cfg
```

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

Make sure that **PPC\_MODEL = swift**. After editing the **flow.cfg** file, save it.

2. Invoke the design flow engine to start the simulation:

```
$ make ba_sim
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Embedded** → **Make Timing Simulation**)

The flow engine compiles/links the software, runs **data2bram**, and invokes the Xilinx tools for back-annotation (**ngdanno**, **ngd2ver**). It then launches the simulator tool along with terminals for the UARTs and LCD.

3. Run the simulation:

In the MTI window, type **run -all**. For VerilogXL, the simulation begins immediately.

**Troubleshooting Hint:** Try to slow down the reference clock frequency in the testbench. Another option is to edit the back annotated simulation file:

```
$V2PRO/platforms/embedded/sim/ba_sim/top.v
```

Comment out the **\$sdf\_annotate** command. This causes the simulation to ignore timing, but allows a functional test on the routed FPGA design.

## BFM Simulations

The steps to run a BFM simulation are described below:

1. To run a BFM simulation, edit the **flow.cfg** file to change the processor block model to BFM:  

```
$ cd $V2PRO/platforms/embedded
$ <text editor> flow.cfg
```

(On a Windows PC, you can use the Explorer to browse for this file and edit it.)

Change the line "PPC\_MODEL = swift" to "PPC\_MODEL=bfm". After editing the **flow.cfg** file, save it.

2. Invoke the design flow engine to start the simulation:  

```
$ make ba_sim
```

(On a Windows PC, you can also select **Start** → **Programs** → **Virtex-II Pro Developer's Kit** → **Reference Platforms** → **Embedded** → **Make Timing Simulation**)

This script runs the Xilinx tools for back-annotation (**ngdanno**, **ngd2ver**) and then invokes the BFC on the **.bfl** test script. It then launches the simulator tool along with terminals for the UARTs and LCD.

3. Run the simulation:

In the MTI window, type **run -all**. For Verilog XL, the simulation begins immediately. The test ends successfully when the following is returned:

```
Synch 31 received... Simulation Completed
```

## Memory Map

This section diagrams the system memory map for the Embedded PPC405 Reference System. It also documents the location of the DCR devices as mapped by the OPB to DCR Bridge. The memory map reflects the default location of the system devices as defined in the **global\_params.v** file.

Table 4-13: Memory Maps

### PLB Device Memory Map

| Device                             | Address  |          | Size   | Comment                                                     |
|------------------------------------|----------|----------|--------|-------------------------------------------------------------|
|                                    | Max      | Min      |        |                                                             |
| PLB Slv5 - ZBT Space               | 301FFFFF | 30000000 | 2 MB   |                                                             |
| PLB Slv1 - DDR SDRAM               | 01FFFFFF | 00000000 | 32 MB  | Chip selects can be used to support two DDR banks for 64 MB |
| PLB Slv2 - SRAM Space              | 101FFFFF | 10000000 | 2 MB   |                                                             |
| PLB Slv2 - Flash Space             | 19FFFFFF | 18000000 | 32 MB  |                                                             |
| Data Side OCM Space                | 40001FFF | 40000000 | 8 KB   |                                                             |
| Instruction Side OCM Space         | 50000FFF | 50000000 | 4 KB   |                                                             |
| PLB Slv3 - PLB to OPB Bridge Space | FFFFFFF  | 60000000 | 1.8 GB |                                                             |
| PLB Slv0 - PLB BRAM Space          | FFFF8000 | FFFF0000 | 32 KB  | Contains Boot Vector                                        |

### OPB Device Memory Map

| Device                             | Address  |          | Size   | Comment                      |
|------------------------------------|----------|----------|--------|------------------------------|
|                                    | Max      | Min      |        |                              |
| OPB Slv9 - OPB to PLB Bridge       | FFFFFFF  | F0000000 | 256 MB |                              |
|                                    | 5FFFFFFF | 00000000 | 1.5 GB |                              |
| OPB Slv0 - OPB Arbiter             | 80000007 | 80000000 | 8 B    |                              |
| OPB Slv2 - GPIO                    | 90000007 | 90000000 | 8 B    |                              |
| OPB Slv3 - UART1 (16450)           | A000101F | A0001000 | 32 B   |                              |
| OPB Slv4 - UART2 (16550)           | A001101F | A0011000 | 32 B   |                              |
| OPB Slv1 - IIC Controller          | A800003F | A8000000 | 64 B   |                              |
| OPB Slv6 - PCI Bus Master          | BFFFFFFF | B0000000 | 256 MB | mem addr = PCI addr + offset |
| OPB Slv7 - LCD Controller          | C0000007 | C0000000 | 8 B    |                              |
| OPB Slv8 - OPB to DCR Bridge       | D0000FFF | D0000000 | 4 KB   | mem addr = DCR addr x 4      |
| OPB Slv5 - CROM Space              | E00007FF | E0000000 | 2 KB   |                              |
| OPB Slv10 - Ethernet               | 600003FF | 60000000 | 16 KB  |                              |
| OPB Slv11 - PS/2 #1                | A900003F | A9000000 | 64 B   |                              |
| OPB Slv12 - PS/2 #2                | A900103F | A9001000 | 64 B   |                              |
| OPB Slv13 - Touch Screen Digitizer | AA000003 | AA000000 | 4 B    |                              |

### Memory-Mapped DCR Device Map

| Device                      | Address  |          | Size | Comment<br>(DCR Addr Range)      |
|-----------------------------|----------|----------|------|----------------------------------|
|                             | Max      | Min      |      |                                  |
| DCR Slv3 - BESR / BEAR      | D000000F | D0000000 | 16 B | BESR / BEAR (0x000 - 0x003)      |
| DCR Slv0 - PLB Arbiter      | D000007F | D0000040 | 64 B | PLB Arbiter (0x010 - 0x01F)      |
| DCR Slv4 - Packet Processor | D000081F | D0000800 | 32 B | Packet Processor (0x200 - 0x207) |
| DCR Slv1 - Non-Crit. INTC   | D0000FDF | D0000FC0 | 32 B | Non-Crit. INTC (0x3F0 - 0x3F7)   |
| DCR Slv2 - Crit. INTC       | D0000FFF | D0000FE0 | 32 B | Crit. INTC (0x3F8 - 0x3FF)       |

### PCI Memory Map

| Device                          | Address  |          | Size   | Comment<br>(PCI Addr Range)            |
|---------------------------------|----------|----------|--------|----------------------------------------|
|                                 | Max      | Min      |        |                                        |
| PCI Memory Space                | B7FFFFFF | B0000000 | 128 MB | Mem (0x00000000 - 0x07FFFFFFF)         |
| PCI I/O Space                   | BBFFFFFF | B8000000 | 64 MB  | I/O (0x00000000 - 0x03FFFFFFF)         |
| PCI Card Configuration Space    | BDFFFFFF | BC000000 | 32 MB  | Config (0x00000000 - 0x01FFFFFFF)      |
| PCI Controller Master Registers | BFFFFFFF | BE000000 | 32 MB  | Self Config (0x00000000 - 0x01FFFFFFF) |

UG010\_525\_05\_022102



## Extending or Modifying the Design

The Embedded PPC405 Reference System is a good starting point from which a user can add, remove, or modify components in the system. Since most of the IP in the design is attached to the CoreConnect infrastructure, adding or removing devices is a fairly straightforward process that is explained in the CoreConnect documentation. Below is an overview for making various changes to the system.

### Adding or Removing PLB Devices

To remove a PLB slave:

- Delete the module instantiation for that piece of IP
- Delete all corresponding external I/O ports
- Tie all PLB slave output signals to zero
- Remove any global parameters used exclusively by that slave (such as the base address)

In general, a PLB master is removed in the same way as a PLB slave. The device, its external I/O, and associated global parameters are removed and the PLB signals it normally drives must be tied off to zero.

To add a PLB slave:

- Instantiate the device
- Connect its external I/O to the top level
- Set its configuration parameters (i.e., base address)
- Add the slave outputs to the terms of the OR-gate that drives the PLB slave bus

The reference design has the logic in place for up to nine PLB slaves with only five PLB slave slots used. The extra slave inputs which are tied off to zero can be connected to the new slave. If larger numbers of slaves are needed, the OR logic can be expanded.

The PLB arbiter used in the reference design supports up to eight PLB masters of which only four are instantiated. The other four PLB master ports are tied off to zero. The user can instantiate a new master and connect it to one of the unused PLB master ports in the arbiter.

Though an eight-master PLB arbiter is used in the reference design to make it easier to add PLB masters, different versions of the PLB arbiter are available which support a smaller number of PLB masters. These smaller arbiters are faster and use fewer logic resources than a large arbiter with some of its inputs tied off.

The bus logic in the PLB infrastructure supports up to nine PLB slaves. The bus logic is instantiated in the **plb\_bus\_logic.v** file and uses FPGA-optimized logic structures to build the large fan-in OR-gates more efficiently. The user can modify this file to implement logic that is more efficient for the exact number of PLB slaves in the system. Adding or removing PLB devices may also require adding or removing local signals. Any changes to the memory map must also be passed on to the PLB bus monitor.

### Adding or Removing OPB Devices

OPB devices can be added and removed in a similar manner as with PLB devices. When removing masters or slaves, tie off the OPB signals they were driving with zeros and make the necessary changes to the FPGA's port interfaces.

The reference design uses 14 of 17 available OPB slave connections in its **opb\_bus\_logic.v** module. To support more slaves, modify the OPB bus logic to add additional OR terms.

The reference design implements two OPB masters (the PLB-to-OPB Bridge and Ethernet DMA Engine), but the OPB arbiter supports up to four OPB masters. The OPB bus logic module also supports up to four masters, so adding an OPB master simply involves connecting it to the unused master ports in the bus logic. The unused bus logic master



ports are currently tied off to zero. The user can modify and optimize the OPB bus logic for the exact number of OPB devices in the system. Note that the OPB bus monitor connections may need to be changed if additional OPB devices are added. If the number of OPB devices is changed, the OPB bus monitor must be reconfigured by editing the **opb\_dcl.inc** file. The default value in the IBM CoreConnect toolkit is three, so a modified version of this file is included with the reference design to set the number of devices to 17.

The OPB bus logic assumes the masters and slaves internally gate off their outputs when not driving the bus. The OPB specification states that some signals require AND-OR bus logic structures. Since all the devices in the reference design have the gating logic built in, only OR-gates are needed to connect the OPB devices together. The user may need to add the AND-gating logic if their OPB device requires it. Another optimization in the OPB bus logic to improve performance is to separate the **OPB\_DBus** into master and slave buses. The **OPB\_DBusM** signal used by masters only needs to be driven by the **SI\_DBus** signals from the slaves. The **OPB\_DBusSI** signal used by slaves only needs to be driven by the **M\_DBus** signals from masters. If the master and slave **OPB\_DBus** signals are not combined, an extra level of logic and the added bus loading can be removed. The user can always OR together the **OPB\_DbusM** and **OPB\_DbusSI** signals to make a standard **OPB\_DBus** signal if their IP devices require it.

## Adding or Removing DCR Devices

The reference design provides two DCR buses that offer direct CPU control or control using a memory mapped interface module. The user can add or subtract from the DCR buses very easily by modifying the data bus chain and OR'ing together the corresponding acknowledge signals back to the masters (as described in the DCR specification). Keep in mind the clock domains of the DCR masters and slaves to ensure compatibility.

## Other Modifications

Both the Instruction and Data Side OCM ports are connected to BRAMs. The user can change the memory base addresses, sizes, or clock frequencies. Refer to the OCM documentation for information about connecting BRAM memory to the OCM and configuring it.

Each Interrupt Controller supports up to 32 separate interrupts, but only a few of the interrupt inputs are used. New IP capable of generating interrupts should tie in their interrupt request lines to the interrupt controller so the CPU can see them.

## Behavioral Models/Testbenches

Whenever new IP devices are added or new BFM's are added, the testbenches should be updated. This may include new device models or edits to the connections of some of the bus monitors. The sample BFL script supplied with the design can be used as a template for building custom test scripts.

## Modifying Software

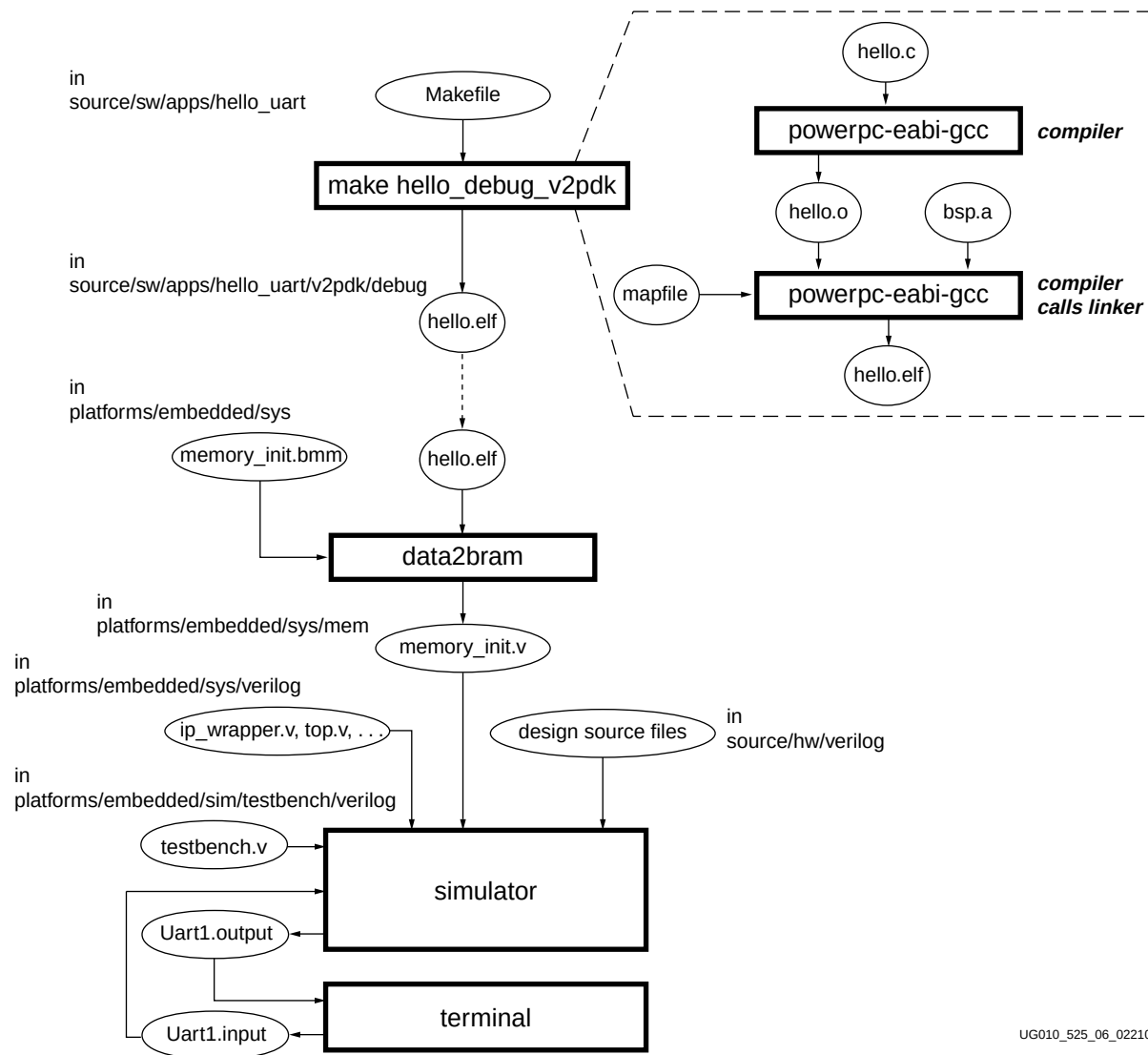
The **hello\_uart** program can be used as a template for learning about writing and compiling software to run on the PPC405. The user can make small modifications to the **hello\_uart** program (for example, changing the **Hello world** message) and take it through the process of building and simulating software. The tool flow can then be extrapolated for the design of new programs with greater complexity. A short outline of the steps for modifying and recompiling the **hello\_uart** program are provided below. (Refer to the *PPC405 Software: Demonstration and Test Applications*, which provides detailed information about designing and compiling software to run on the CPU).

1. Copy \$V2PRO/source/sw tree to your own area:  
`$ cp -Rp $V2PRO/source/sw <My_Apps_Dir>`
2. Change directory to the location of the source file in your own area:  
`$ cd <My_Apps_Dir>/apps/hello_uart`

Edit **hello.c** (change the **Hello world** message)

3. Go to the Embedded PPC405 Reference System directory  
`$ cd $V2PRO/platforms/embedded`
4. Edit the **flow.cfg** file to point to the location of the software program  
`$ <text editor> flow.cfg`  
`set SW_MAKE = make -C <My_Apps_Dir>/apps/hello_uart hello_debug_v2pdk`  
`set SW_ELF = <My_Apps_Dir>/apps/hello_uart/v2pdk/debug/hello.elf`
5. Run the simulation  
`$ make func_sim`

Your new message should be displayed on the terminal. The diagram in [Figure 4-6](#) summarizes the tool flow and source file relationships for the functional simulation of the **hello\_uart** program as described in this document and the corresponding software documentation.



UG010\_525\_06\_022102

Figure 4-6: Design Flow for Functional Simulation

## File Structure for Embedded PPC405 Reference System

```

\-----platforms
|
| \-----embedded
| |
| | \-----par
| | |
| | | \-----v2p
| | |
| | | \-----sim
| | | |
| | | | \-----ba_sim
| | | | \-----bfl
| | | | \-----func_sim
| | | | \-----testbench
| | | | |
| | | | | \-----verilog
| | | | | \-----vhdl
| | |
| | | \-----syn
| | | |
| | | | \-----v2p
| | |
| | | \-----sys
| | | |
| | | | \-----mem
| | | | \-----verilog
| | | | \-----vhdl
|
| \-----source
| |
| | \-----hw
| | |
| | | \-----verilog
| | | |
| | | | \-----crom
| | | | \-----dcr_brg
| | | | \-----ethernet
| | | | \-----gpio
| | | | \-----intc
| | | | \-----ipif_slv_sram_simple
| | | | \-----lcd_cntlr
| | | | \-----opb_arb
| | | | \-----opb_iic
| | | | \-----opb_ps2
| | | | \-----opb_tsd
| | | | \-----opb2plb_bgi
| | | | \-----pci_brg
| | | | \-----pkt_proc
| | | | \-----plb2opb_bgo
| | | | \-----plb_arbiter
| | | | \-----plb_bram_cntlr
| | | | \-----plb_ddr_cntlr
| | | | \-----plb_sram_flash_cntlr
| | | | \-----plb_tft_lcd_cntlr
| | | | \-----plb_uart
| | | | \-----plb_zbt_cntlr
| | | | \-----uart
| | |
| | | \-----sim_models
| | | |
| | | | \-----other
| | | | |
| | | | | \-----ddr_mem
| | | | | \-----flash_mem
| | | | | \-----iic
| | | | | \-----pci
| | | | | \-----sram_mem
| | | | | \-----uart_rcvr
| | | | | \-----zbt_mem
| | | |
| | | | \-----Xilinx
| | | | |
| | | | | \-----proc_block_BFM
| | | | | \-----verilog
| | | | | |
| | | | | | \-----smartmodel
| | |
| | | \-----sw
| | | |
| | | | \-----apps
| | | | |
| | | | | \-----hello
| | | | | \-----hello_gpio

```

```
`-----tools
```

# **Volume 7: Hardware IP Specifications**

## ***Virtex-II Pro™ Platform FPGA Developer's Kit***

March 2002 Release





The Xilinx logo shown above is a registered trademark of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

"Xilinx" and the Xilinx logo are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, XC5210 are registered Trademarks of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Benchner, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Benchner, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, Nano-Blaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, Rocket I/O, SelectI/O, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM      IBM Logo      PowerPC      PowerPC Logo      Blue Logic      CoreConnect      CodePack

All other trademarks are the property of their respective owners.

Xilinx does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 2002 Xilinx, Inc. All Rights Reserved.

# *Introduction to Hardware IP*

---

## Summary

This volume details the hardware IP that is shipped with the Virtex-II Pro™ Developer's Kit.

## Introduction

The Virtex-II Pro Developer's Kit currently provides a number of Processor Local Bus (PLB), On-chip Peripheral Bus (OPB), and Device Control Register (DCR) bus based hardware elements. Each of these is documented in greater detail in the chapters that follow. Generally, the interface and function of the IP is described, along with sufficient register information for customers to use the devices.

In addition to describing the individual hardware IPs, this volume also includes a specification of the IP InterFace, or IPIF modules. These modules are designed to greatly accelerate the process of hooking up pre-existent IP, or creating new IP in a system. The specification defines a CoreConnect™ compliant interface on one side, and a simple interface for hooking up existent IP on the other side. Since this is a building block piece of IP, it is available for use in any customer design.

The hardware IP uses the IBM CoreConnect bus standards as their means of communication between the PowerPC® and other devices. These standards are documented in the IBM CoreConnect release. Please see the **Further Reading** section for more information on where to find the relevant documents

Further information on each IP, and on the IPIF can be found in the following chapters of this volume. If you have suggestions for improvement, or have questions, please contact the relevant support contact listed in the letter that came with your Developer's Kit.

## Hardware IP Sizes

The following table illustrates the basic sizes of the Hardware IP available in the Virtex-II Pro Developer's Kit. This data was compiled by synthesizing the IP using Synplify, Inc. Synplify 7.0.3 tool.

**NOTE:** These hardware IPs are not yet CoreGen IP, and their size will decrease as parameters control how they are built. These area numbers represent a full implementation of each IP.



Table 1-1: Actual Logic Utilization from Embedded Reference System

| IP Module                              | DFFs        | LUTs        | Slice (Est) | BRAMs     | Notes |
|----------------------------------------|-------------|-------------|-------------|-----------|-------|
| PLB Arbiter (8M/8S)                    | 234         | 1237        | 735         | 0         | (1)   |
| PLB Bus Logic                          | 0           | 95          | 47          | 0         | (2)   |
| PLB DDR SDRAM                          | 479         | 279         | 379         | 0         |       |
| PLB ZBT SRAM                           | 401         | 97          | 249         | 0         |       |
| PLB SRAM/Flash                         | 354         | 242         | 298         | 0         |       |
| PLB BRAM                               | 183         | 149         | 266         | 16        |       |
| PLB2OPB Bridge                         | 609         | 763         | 686         | 0         | (3)   |
| OPB Arbiter                            | 33          | 120         | 76          | 0         |       |
| OPB Bus Logic                          | 0           | 108         | 54          | 0         | (4)   |
| OPB IIC                                | 182         | 279         | 230         | 0         |       |
| OPB 16450 UART                         | 322         | 425         | 373         | 0         |       |
| OPB 16550 UART                         | 418         | 678         | 548         | 0         |       |
| OPB GPIO                               | 201         | 87          | 144         | 0         |       |
| OPB CROM                               | 83          | 78          | 80          | 1         |       |
| OPB LCD Controller                     | 76          | 66          | 71          | 0         |       |
| OPB2PCI Bridge                         | 1147        | 802         | 974         | 0         | (5)   |
| OPB2DCR Bridge                         | 135         | 73          | 104         | 0         |       |
| OPB2PLB Bridge                         | 223         | 99          | 161         | 0         | (6)   |
| INTC - Crit                            | 193         | 312         | 252         | 0         | (7)   |
| INTC - Non Crit                        | 193         | 312         | 252         | 0         | (7)   |
| Packet Processor (OCM)                 | 474         | 783         | 628         | 7         | (8)   |
| Packet Processor (PLB)                 | 474         | 783         | 628         | 1         | (9)   |
| <b>Synthesis Totals</b>                | <b>6414</b> | <b>7867</b> | <b>7140</b> | <b>25</b> |       |
| <b>Foundation 4.2i post PAR totals</b> | <b>5414</b> | <b>6448</b> | <b>5075</b> | <b>25</b> |       |

**Notes:**

1. 8 master, 8 Slave. Only three master ports used; not optimized
2. Connects five PLB slaves
3. Includes logic for fully compliant 64-bit PLB interface
4. Connects one OPB master and ten OPB slaves
5. Includes PCI Core and logic to allow PCI clock to be 1/n of OPB clock. Bridge contains 415 DFFs, 78 LUTs, the remainder is PCI Core
6. Placeholder design for OPB2PLB bridge. The current bridge supports all OPB transactions, but does not attempt to optimize them for PLB. Final bridge design will add PLB burst ability.
7. The INTC allows up to 32 interrupt inputs. The area for this INTC assumes all 32 inputs are used
8. This is the Packet Processor which connects to the OCM BRAMs in the Embedded Ref System
9. This is the Packet Processor which connects to the PLB BRAM in the Embedded Ref System

## Further Reading

The Virtex-II Pro™ Developer's Kit contains a wealth of valuable information that will assist you in your design efforts. The documentation contained within the eight volumes is organized to assist you in quickly finding relevant materials. To obtain the most recent revision of this documentation, please see

[http://support.xilinx.com/xlnx/xil\\_tt\\_product.jsp?sProduct=Virtex-II+Pro](http://support.xilinx.com/xlnx/xil_tt_product.jsp?sProduct=Virtex-II+Pro).

### Documentation Provided by Xilinx

*Virtex-II Pro™ Platform FPGA Handbook*

*Virtex-II Pro™ Platform FPGA Developer's Kit Documentation:*

*Volume 1, Advance Product Specification*

*Volume 2, PowerPC® 405 Processor*

*Volume 3, Rocket I/O™ Transceiver User Guide*

*Volume 4, Design Flow*

*Volume 5, Software Development Tools*

*Volume 6, Reference Systems*

*Volume 7, Hardware IP Specifications*

*Volume 8, Software IP and Applications*

### IBM® CoreConnect™ Documentation

The Virtex-II Pro Developer's Kit integrates seamlessly with the IBM CoreConnect Toolkit. This toolkit is not included with the Developer's Kit, but is required if bus functional simulation is desired. The toolkit provides a number of features which enhance design productivity and allow you to get the most from the Developer's Kit. To obtain the toolkit, you must be a licensee of the IBM CoreConnect Bus Architecture. Licensing CoreConnect provides access to a wealth of documentation, Bus Functional Models, Hardware IP, and the toolkit.

Xilinx provides a Web-based licensing mechanism that allows you to obtain the CoreConnect toolkit from our website. To license CoreConnect, use an Internet browser to access [http://www.xilinx.com/ipcenter/processor\\_central/register\\_coreconnect.htm](http://www.xilinx.com/ipcenter/processor_central/register_coreconnect.htm). Once your request has been approved (typically within 24 hours), you will receive an e-mail granting access to a protected website. You may then download the toolkit.

If you prefer, you can also license CoreConnect directly from IBM.

If you would like further information on CoreConnect Bus Architecture, please see IBM's CoreConnect website at <http://www.ibm.com/chips/products/coreconnect>.

Once you have licensed the CoreConnect toolkit, and installed it with the Developer's Kit, the following documents will be available to you in the following locations:

#### IBM CoreConnect Bus Architecture Specifications

*IBM CoreConnect Processor Local Bus (PLB) Architecture Specification*  
see \$CORECONNECT/published/corecon/64bitPlbBus.pdf

*IBM CoreConnect On-chip Peripheral Bus (OPB) Architecture Specification*  
see \$CORECONNECT/published/corecon/OpbBus.pdf

*IBM CoreConnect Device Control Register (DCR) Bus Architecture Specification*  
see \$CORECONNECT/published/corecon/DcrBus.pdf

#### IBM CoreConnect Toolkit Documentation

*PLB Bus Functional Model Toolkit - User's Manual*  
see \$CORECONNECT/published/corecon/PlbToolkit.pdf

*OPB Bus Functional Model Toolkit - User's Manual*  
see \$CORECONNECT/published/corecon/OpbToolkit.pdf

*DCR Bus Functional Model Toolkit - User's Manual*  
see \$CORECONNECT/published/corecon/DcrToolkit.pdf

*CoreConnect Test Generator - User's Manual*  
see \$CORECONNECT/published/corecon/ctg.pdf

**Note:** \$CORECONNECT is an environment variable that is created when installing the Developer's Kit or CoreConnect Toolkit.

## Software Development Documentation

There are many sources of documentation available for those who wish to learn more about Software Development. It is recommended that a web search be conducted using a favorite search engine for keywords such as "PowerPC+Software+Development". Alternatively, a technical bookstore should be able to provide many valuable resources. The books listed below are a very small fraction of those available.

### Books About Programming in C

There are many good books about the C and C++ programming languages. A few of these are listed below:

#### Books about C

Kernigham Brian, Ritchie Dennis. 1988. *The C Programming Language*. Prentice Hall

Wang Paul. 1992. *An Introduction to ANSI C on Unix*. Wadsworth.

Kumar Ram, Agrawal Rakesh. 1992. *Programming in ANSI C*. West Publishing Company.

#### Books about C++

Stroustrup Bjarne. 1991. *The C++ Programming Language*. Addison-Wesley.

Lafore Robert. 1995. *Object-Oriented Programming in C++*. Waite Group Press.

### Online Documents About Programming in C

The Internet offers plenty of documentation about how to program in C and C++. One of the best approaches to finding documentation online is to use a search engine (such as Google, <http://www.google.com>) and search on "introduction to programming in C."

# Using IPIF to Build IP

---

## Abstract

Virtex-II Pro™ devices combine PowerPC® CPUs and FPGA fabric into one integrated circuit. In the past, system development efforts relied on engineers building each component from scratch. Today, engineers have a wide variety of microprocessor peripherals in their IP libraries. The Intellectual Property InterFace (IPIF) is designed to ease the creation of new IP, as well as the integration of existant IP, within a Virtex-II Pro device. This chapter will illustrate the utility of the IPIF to integrate IP into a system.

## Introduction

Intellectual Property InterFace (IPIF) modules simplify the development of CoreConnect™ devices. The IPIF converts complex system buses, such as the PLB or OPB, into common interfaces, such as an SRAM protocol or a control register interface. This makes IPIF modules ideal for quickly developing new bus peripherals, or converting existing IP to work in a CoreConnect bus-based system. The IPIF modules provide point-to-point interfaces using simple timing relationships and very light protocols.

The IPIF is designed to be bus-agnostic. This allows the back-end interface for the IP to remain the same while only the bus interface logic in the IPIF is changed. It therefore provides an efficient means for supporting different bus standards without change to the IP device.

IPIF modules also provide support for DMA and interrupt functionality. The IPIF is designed to support a wide variety of common interfaces (like SRAM, FIFO, and control register protocols), but may not be the optimal solution in all cases. Where additional performance or functionality is required, the user can develop a custom OPB or PLB bus interface.

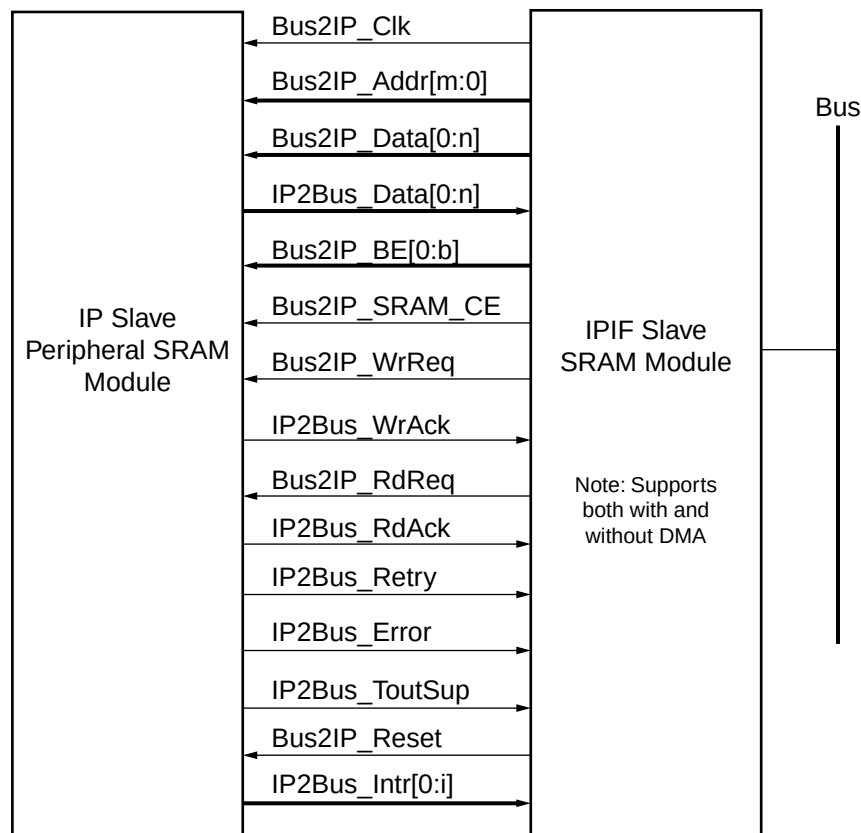
IPIF modules simplify driver software development since the IPIF framework contains many common features. These include a consistent means of interrupt handling, DMA, and optional support for a discovery process using a Configuration ROM (CROM). The CROM contains information about the devices in the system (such as address mapping, device capabilities, and revisions) that software can read at boot-up to configure its device drivers. Refer to the IPIF Specification and the CROM Specification for more detailed information.

This document demonstrates how quickly and easily a new piece of IP can be developed using the IPIF. The process and steps for building a new CoreConnect device based on the SRAM protocol IPIF is deccribed below. For this sample design, a 32-bit General Purpose I/O (GPIO) device will be created. The GPIO allows a CoreConnect master such as the CPU to be able to control a set of external pins using a simple memory-mapped interface.

## SRAM Protocol Overview of IPIF

**Figure 2-1** diagrams the connections between the IPIF and the user IP for SRAM protocol interface. The IPIF simplifies the design by providing a PLB or OPB interface and condensing it down to a small set of easily understood signals.

Figure 2-1



UG010\_506\_14\_030501

**Figure 2-1: IPIF SRAM Module Interface**

All interface signals with the IPIF are synchronous to rising clock edges. The IPIF takes the clock from the OPB or PLB bus interface and passes it to the IP, causing the IP to use the same global clock as the bus it is connected to. (Future IPIF designs may permit the IP clock and the bus clock to be independent.) The SRAM interface protocol used by the IPIF can be described by observing what a write and read transaction looks like.

## Basic Write Transactions

**Figure 2-2** shows the timing diagram for a write transaction. A write transaction begins when the IPIF drives the address (Bus2IP\_Addr), byte enables (Bus2IP\_BE), and write data (Bus2IP\_Data) to the IP. Note that the signal direction is specified in the signal name: Bus2IP vs IP2Bus. The IPIF qualifies the write by asserting a single clock cycle High pulse (Bus2IP\_WrReq) at the beginning of the transaction. It then waits for the IP device to acknowledge completion of the write by sending back a single clock cycle High pulse on IP2Bus\_WrAck. During the entire transaction from Bus2IP\_WrReq to IP2Bus\_WrAck, the signal Bus2IP\_SRAM\_CE is held high as an enveloping signal around the transaction. After a completed transaction, the IPIF may issue a new transaction. Note that burst write transactions on the bus are converted into a series of single data transfers to the IP, which all look alike.

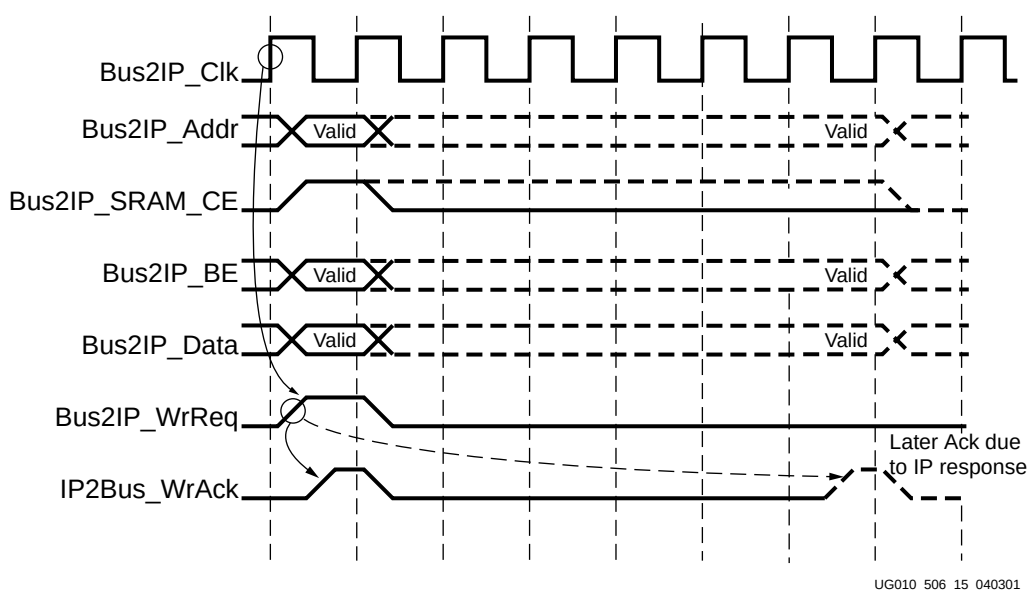


Figure 2-2: IPIF Simple SRAM Write Cycle

## Basic Read Transactions

**Figure 2-3** diagrams a read transaction, which looks very similar to a write transaction. A read transaction begins when the IPIF drives the address (Bus2IP\_Addr) and byte enables (Bus2IP\_BE) to the IP. It qualifies the read by asserting a single clock cycle high pulse (Bus2IP\_RdReq) at the beginning of the transaction. It then waits for the IP device to acknowledge completion of the read by sending back a single clock cycle High acknowledge pulse on IP2Bus\_RdAck. During the entire transaction from Bus2IP\_RdReq to IP2Bus\_RdAck, the signal Bus2IP\_SRAM\_CE is held high as an enveloping signal around the transaction. After a completed transaction, the IPIF may issue a new transaction. Note that burst read transactions on the bus are converted into a series of single data transfers to the IP, which all look alike.

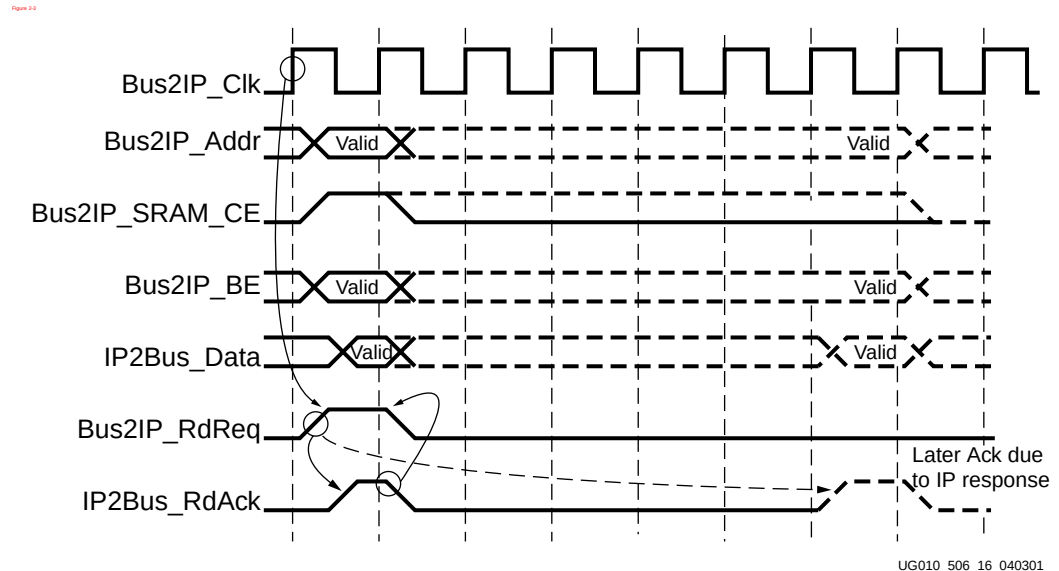


Figure 2-3: IPIF Simple SRAM Read Cycle

## IPIF Status and Control Signals

Extra status and control signals are also present in the SRAM protocol. If the IP2Bus\_Retry signal is asserted instead of IP2Bus\_RdAck/IP2Bus\_WrAck, the IPIF will assert retry on the bus side and terminate the transaction. IP2Bus\_Error asserted with IP2Bus\_RdAck/IP2Bus\_WrAck will cause the IPIF to signal an error on the bus interface. For slow IP devices, an IP2Bus\_ToutSup signal can be asserted to prevent timeouts on the bus interface. Finally the Bus2IP\_Reset passes the bus-side reset to the IP.

## Using IPIF to Create a GPIO Peripheral from Scratch

The General Purpose Input/Output (GPIO) peripheral can be used to show how the IPIF simplifies new peripheral creation. The GPIO module has three 32-bit registers: one register to control the TBUF for each I/O pin, one register to write the I/O pins, and one register to read the I/O pins. The GPIO peripheral uses a very small amount of additional "control logic" when used with a 32-Bit IPIF Slave SRAM module.

Figure 2-4 shows a conceptual view of the logic necessary to build the GPIO module using the IPIF Slave SRAM module. The IP2Bus\_RdAck / IP2Bus\_WrAck signals are directly connected to the corresponding Bus2IP\_RdReq / Bus2IP\_WrReq signals, since it only takes one clock cycle to read or write the GPIO registers. If more "access time" is required by the registers, a simple SRL16-based shift register between the Req/Ack signals could be used to set the number of cycles the register will respond in. An example use of this function is to gain timing margin by treating the register access as a multicycle path. This simple enhancement to the IPIF can have very positive effects in meeting the timing requirements typical of complex microprocessor-based systems. Note too that register response time can be tuned differently between the read and the write.

Figure 2-4

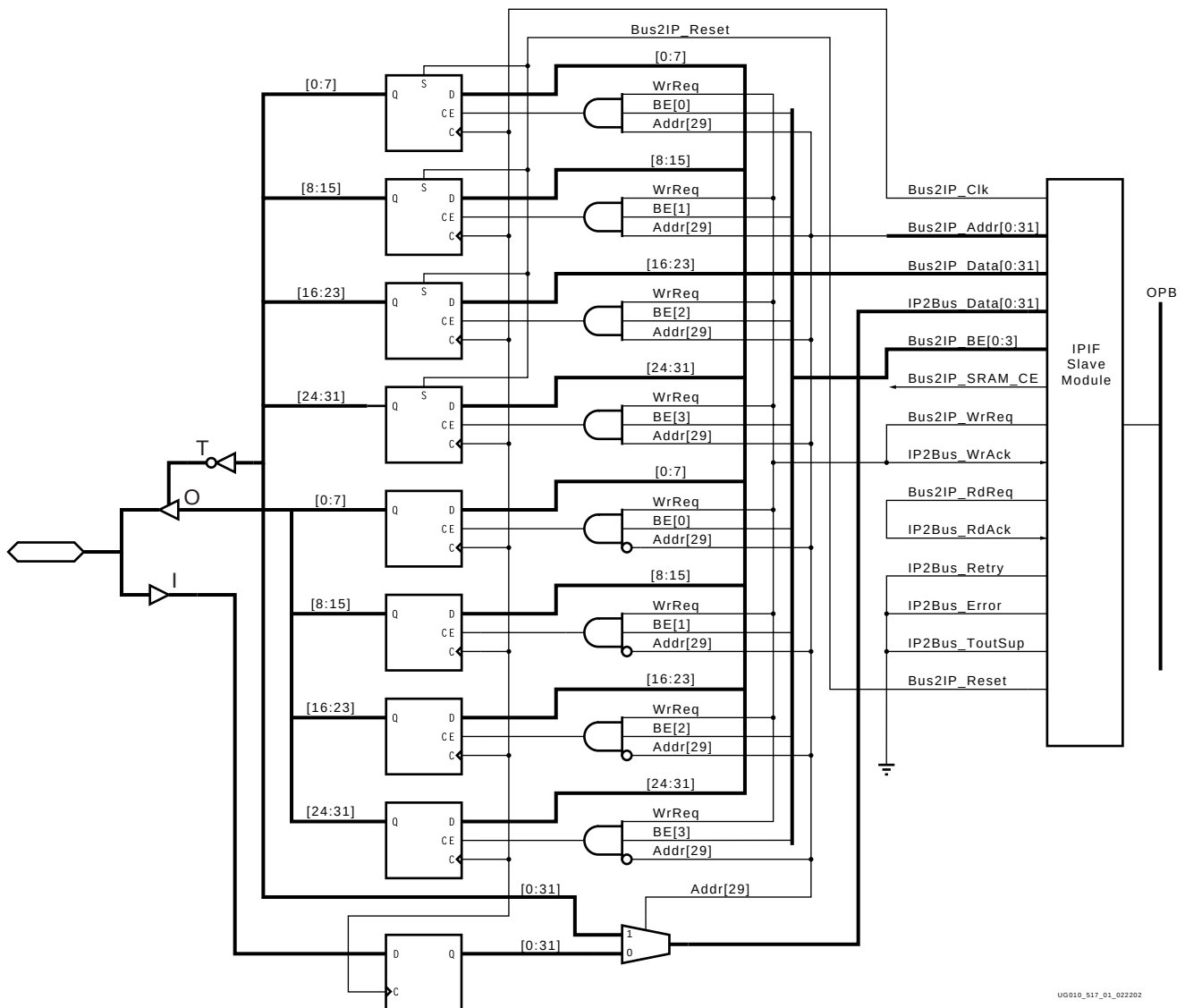


Figure 2-4: IPIF SRAM Module to GPIO logic Interface



To drive an external I/O pin, the output enable for that pin must be asserted, allowing the pin to be driven high or low based upon the contents of the write register. If the output enable for a given pin is deasserted, the pin's driver is put in a high impedance state allowing an external device to drive the pin. The CPU can sense the current value of any pin (regardless of its direction) by reading the read register. Driving the direction of the I/O pin is controlled by the contents of the three-state register.

The GPIO registers support byte enables during writes to the 32-bit registers. Note that a set of simple AND gates is all that is required to generate a Clock Enable to the registers. Four 3-input AND gates are used to drive the 4 bytes of the three-state control register, and four more 3-input AND gates are used to drive the 4 bytes of output-pin data.

The IPIF uses a set of user-specified parameters that allow common things such as the base address of the IP to be established. These parameters are specified before the system is implemented in order to minimize the logic area and maximize the performance of the system. Note that in the GPIO example, additional decoding is used externally to specify two different memory locations. One location is used for reading or writing to the I/O pins (read register, and write register share the same address), and one location is used for reading and writing to the T of the I/O pin (three-state register).

## Using IPIF to Connect a Pre-existent Peripheral to the Bus

Often, some legacy IP will need to be brought into a modern system. Many of these legacy IPs use some form of 8-bit microprocessor bus. Typically, this might consist of a few address lines, an 8 bit data bus, a read and write signal, a chip enable, a clock, a reset, and perhaps an interrupt pin. In most instances, this kind of IP can be almost directly connected to the IPIF SRAM module. This particular IPIF module was actually designed to serve this very purpose.

To connect a legacy IP, one would simply connect the address, data, chip enable, clock, reset, and interrupt pins to their corresponding versions in the IPIF SRAM module. Some small amount of logic might be needed to generate a properly timed read or write signal. The IPIF SRAM module provides separate Req/Ack pairs for read and write, because many older peripherals require different timing for reads and writes. For read or write, the logic between the IP and the IPIF must accomplish two things:

- Provide the proper response time to the IPIF so the peripheral's register can be read or written
- Provide the proper relationship of the read or write signal on the IP relative to the address and data

Consider the following example: The IP might expect its write signal to be valid one clock after address and data is valid, and be held for four clock cycles to properly write the data. Following write going invalid, the address and data must be held for one additional cycle. To accommodate this kind of pattern, a six-stage shift register (SR) can be implemented. The D input to the SR is tied to the Bus2IP\_WrReq pin, and the Q output of the SR is tied to the IP2Bus\_WrAck pin of the IPIF. This provides the proper timing for the length of time the cycle must be held on the bus. By using the first, second, third, and fourth taps of the SR, and feeding them into an OR gate, a write strobe can be generated for the IP. If this write strobe must be glitch-free, taps 0, 1, 2, and 3 could be used, OR'd and fed into a synchronizing register. **Figure 3-13 in Chapter 3, IPIF Specification**, provides a more graphic example of this type of structure. While on the surface this may appear wasteful of logic, Xilinx FPGAs are abundantly equipped with flip-flops.

## Conclusion

Using the IPIF with a small amount of logic makes it very easy to create CoreConnect devices with little knowledge of the buses used. For complex buses such as PLB, this saves the designer time and helps to ensure IP functions correctly, since the IPIF provides a pre-verified design to connect to. The GPIO design is just one example of how IPIF can be used. More examples of IPIF designs are provided within many of the other IP devices in the reference systems. Designs such as those described in **Chapter 5, OPB Configuration ROM Module**; **Chapter 7, OPB LCD Controller**; **Chapter 11, OPB to DCR Bridge**; and **Chapter 12, OPB to PCI Bridge**, demonstrate additional systems based on IPIF. It is recommended that the designer who wishes to learn about IPIF studies the sample source code for some of these IPIF-based designs in context with simulation to gain experience with IPIF.

The IPIF shipped in the Developer's Kit currently only support the SRAM module. Additional modules will be added into CoreGen for maximum parameterizability. Please check <http://www.xilinx.com/v2pdk> for the latest on IPIF, Hardware and Software IP, and updates to the Developer's Kit.



## IPIF Specification

### Overview

The Intellectual Property InterFace (IPIF) backplane is a framework that provides a common set of interfaces for connecting intellectual property to on-chip buses. These common interfaces are built in a modular fashion, allowing standardized connections between IP and the on-chip bus. Additionally, these modules provide an infrastructure to assist in developing new IP, as illustrated in the examples of on-chip bus interface techniques.

Each interface is described as a module that can be independently “plugged in” to the backplane. This modular approach allows the customer to pay only for the resources that are required to implement the desired functions.

Although this specification describes the overall use of interface modules in a system, but it does not address how the IPIF backplane integrates with the rest of the system. This information is addressed in the IPIF Architectural Specification, *not yet released*.

The following documents provide additional useful information:

- IPIF Architectural Specification
- Virtex-II Pro™ Data Sheet
- IBM CoreConnect™ PLB Architectural Specification

Xilinx IPIF modules, available to internal IP developers, third party IP developers, and customers, are designed around a common modular framework that permits mixing and matching of various modules. The interfaces provided in this framework are based on commonly used interfaces for existent IP, with a mind to future IP development as well. These interface modules represent an abstraction layer to assist in developing fully customized peripherals for an on-chip bus. **Figure 3-1** illustrates the scope of the interfaces addressed in this specification.

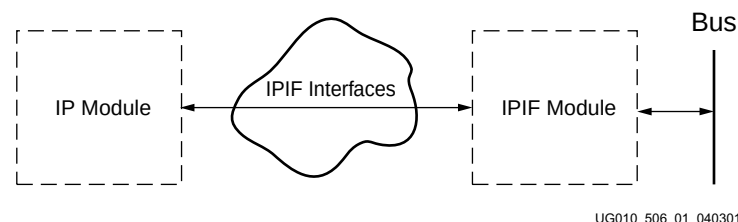


**Figure 3-1: Scope of Interfaces in This Specification**

The IPIF Backplane provides two basic interface classes: IPIF master and IPIF slave. This specification defines one IPIF master, and four IPIF slave modules. The various modules allow system integrators to minimize the FPGA resources required to implement a particular IP, easing difficulty in achieving performance targets and reducing cost.

## IPIF Master Module Overview

The IPIF master module is a single interface that looks very much like a simple SRAM interface. The master IP must provide address, data, and a read or write command in order to initiate a transaction onto the bus. The IPIF master module provides 8, 16, or 32-bit interfaces to the master IP as required by the system implemented.

## IPIF Slave Modules Overview

IPIF slave modules can be built with any combination of four different interfaces:

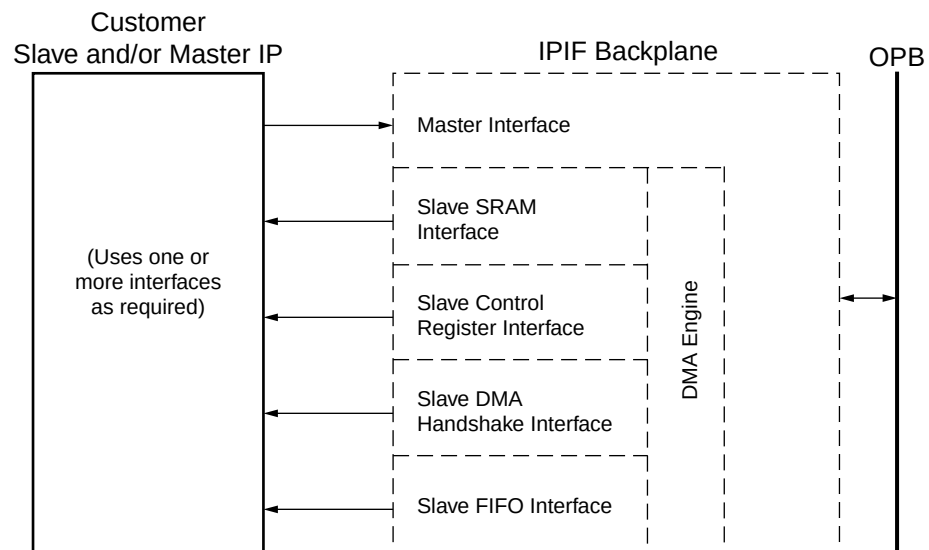
- SRAM
- Control Register
- FIFO
- DMA Handshake

Each interface is designed to address differing needs, but can be combined to meet multiple needs. For example, a customer with a typical microprocessor peripheral IP, such as a 16550 UART, can use the IPIF Slave SRAM module to easily connect a peripheral. Similarly, if a customer requires only a few control registers, the IPIF Slave Control Register module can be used to connect the control registers to the bus. The IPIF Slave FIFO module is designed to connect to communications devices with FIFO interfaces. The IPIF Slave DMA Handshake module provides the classic DMA\_Req/DMA\_Ack handshake to enable devices that require such handshakes for proper operation.

**NOTE:** The IPIF Slave DMA Handshake module differs from the optional DMA engine described in the **DMA Engine** section.

For certain existent microprocessor-style peripherals, mixing IPIF modules may be advantageous. The IPIF master module and IPIF slave modules can be instantiated as mix-and-match components. For example, a simple IP might only require an IPIF Slave SRAM module, whereas a more complicated IP might require IPIF Master, Slave SRAM, Slave Control Register, and Slave FIFO modules. All modules share logic on a common backplane, thereby minimizing FPGA resource utilization for the desired functions.

Master and slave IPIF modules have user-settable parameters that control their capabilities. Typical parameters include which IPIF modules to build, what addresses they reside at, how wide their IP data path is, etc. Additionally, the IPIF master and IPIF slave modules can be built with a Direct Memory Access (DMA) controller of varying capabilities, referred to in this specification as the DMA engine. The DMA engine can be instantiated into any of the IPIF modules to provide basic DMA service, Scatter Gather (SG) via linked list and/or linear list, as well as basic packet processing functions. The DMA engine and other capabilities of the IPIF are set at the time of instantiation, and are visible to software for device discovery and management. (For more details, see the DMA/SG section of the IPIF Architectural Specification, *not yet released*.)



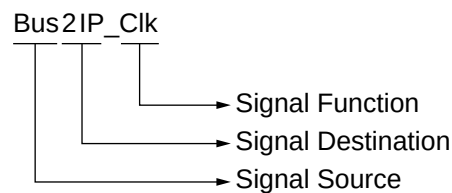
UG010\_506\_02\_030501

Figure 3-2: General Model of the IPIF Backplane

The IPIF modules are really subsets of a general model, demonstrated in Figure 3-2. The general IPIF model was developed such that each IPIF module could be instantiated as required by the IP system integrator. Note that the general model of the IPIF shows each of the module types: IPIF Slave DMA Handshake, IPIF Slave Control Register, IPIF Slave SRAM, IPIF Slave FIFO, and IPIF Master. Additionally, the general model of the IPIF shows the optional DMA engine. This general model offers the IP integrator broad design functionality.

## Signal Conventions

The signal names used throughout this specification use a specific format to identify the direction and function of each signal. This signal name format is illustrated in Figure 3-3.



UG010\_506\_03\_040301

Figure 3-3: Signal Name Format Example

In addition, three source and destination signals have specific meaning:

- Bus - Signals that connect to the IPIF
- IP - Signals that connect to the IP
- FIFO - Signals that connect to a FIFO in the IPIF, when present

## Bus Numbering and Bit Ordering

The IBM signal name convention is used for address, data, and control buses throughout this specification. IBM numbers the bits from left to right as 0 to  $n$ . For example, a 32-bit data bus is named `IP2Bus_Data[0:31]`.

The bit order for buses always has the most significant bit (MSB) on the left, and the least significant bit (LSB) on the right. Thus, `IP2Bus_Data[0:31]` is identified as a 32-bit bus, whose MSB is bit 0, and whose LSB is bit 31. This is commonly known as “big endian” format.

The IPIF specification uses big endian bit ordering for all of its examples. However, it is possible to use parameters to alter the endianness of the IPIF for proper interaction with Motorola or Intel style IP. *(These parameters will be covered in a new section, TBD.)*

## Parameter Indexing Versus Parameter Numbering

The function of the IPIF can be changed by setting various parameters. Please see **IPIF Parameterization** for more details. There are four main classifications of parameters that are used within this specification: number, index, boolean, and mask.

Consider a parameter that adjusts the size of the data bus. In this case, the parameter is best defined as a number. For example, the IPIF data bus width can be set to 8, 16, or 32, depending on how the parameter `IP_DATA_BUS_WIDTH` is set.

However, in the bits of the buses whose width is controlled by `IP_DATA_BUS_WIDTH`, the indices of the bus start at zero and end at `IP_DATA_BUS_WIDTH - 1`. For example, `IP2Bus_Data[0: $n$ ]`, where  $n$  is the index parameter based on the `IP_DATA_BUS_WIDTH` number parameter minus one.

The result of this situation is that parameters in the IPIF are defined as either number parameters or index parameters, in order to clearly identify the function of the actual parameter value.

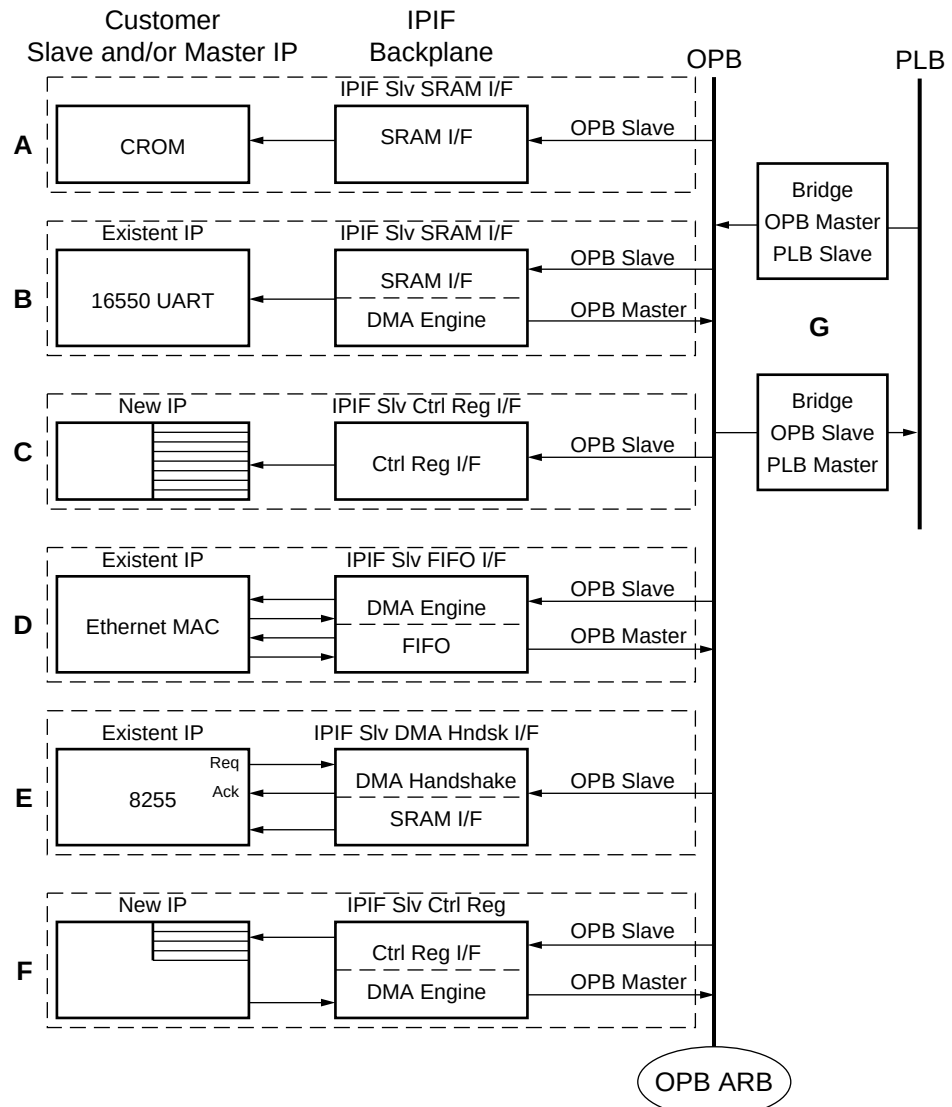
Boolean parameters are those that have binary states; 1 = on and 0 = off. For example, `IP_HAS_OWN_INTC`.

Parameters can also be mask values. A mask is a bit-wise operation that is used by the IPIF to apply a specific enable operation, typically against another parameter. For example, the IPIF Slave SRAM module defines an array of parameters called `IP_SRAMb_BASE_ADDR`. These parameters specify the start address of a particular region of memory which the IPIF will decode. In order to allow for a set of addresses to reside within the space began at `IP_SRAMb_BASE_ADDR`, a mask value parameter is used to set which specific bits of the address are decoded and which are ignored. This is set by the `IP_SRAMb_BASE_ADDR_BIT_ENBL` mask value.

## IPIF Modules in an Example OPB System

**Figure 3-4** is a diagram of an example system utilizing an IBM CoreConnect™ On-Chip Peripheral Bus (OPB). It illustrates a variety of IPIF modules connected to a variety of IP elements. This example system is intended only to illustrate general abilities, and should not be construed as limitations upon the IPIF.

The IPIF easily connects to new and existing IP. The example in [Figure 3-4](#) illustrates possible scenarios.



UG010\_506\_04\_040301

Figure 3-4: Example System Using IPIF to Connect to OPB

#### A: Slave SRAM to CROM

Section A in [Figure 3-4](#) illustrates an IPIF Slave SRAM module used to connect to a configuration ROM (CROM) made from a Xilinx block RAM. The CROM can be used to store information about the entire system, such as the system version number, IP capabilities, and IP version numbers. (More information on CROM structures appears in the IPIF Architectural Specification, *not yet released*.)

#### B: Slave SRAM to UART

Section B in [Figure 3-4](#) shows another IPIF Slave SRAM module that interconnects with an existent IP, such as the 16550 UART shown here. This example uses the optional DMA engine that is instantiated to relieve the CPU burden in moving data. (More information on DMA engine capabilities appears in the IPIF Architectural Specification, *not yet released*.)



### C: Slave Control Register to New IP

Section C in [Figure 3-4](#) shows a customized piece of new IP that utilizes the IPIF Slave Control Register module in order to reduce complexity in the IP. By using this module, the new IP can directly instantiate its registers that are connected to the IPIF Slave Control Register module. The IPIF backplane operates independently of the registers in the new IP, and does not rely on whether the registers are read/write, read only, or write only. The IPIF Slave Control Register module allows other OPB masters to have memory mapped access to the registers in the IP.

### D: Slave FIFO and DMA Engine to Ethernet MAC

Section D in [Figure 3-4](#) illustrates the IPIF Slave FIFO module and optional DMA engine interfaced to an Ethernet MAC, directly providing the bulk of logic to allow quick design of an IP system. Some communications IP might also contain registers that are memory mapped, which may require additional IPIF Slave Control Register or IPIF Slave SRAM models, depending upon the type of interface provided by the IP.

### E: Slave DMA Handshake to 8255

Section E in [Figure 3-4](#) shows another example of an old legacy CPU peripheral, in this case an 8255 Parallel I/O IP. Legacy devices often contain fairly simple CPU interfaces and DMA handshaking provisions. These kinds of IP can easily connect to the IPIF backplane. The IPIF Slave SRAM module is used for the CPU interface to the IP and the IPIF Slave DMA Handshake module is used for the DMA handshake of the IP. The IPIF DMA Handshake module allows the IPIF backplane to store up a series of accesses before interrupting the processor that is requesting service.

While it is possible to implement the optional DMA engine in this example, it might be FPGA-area inefficient to use it for a slow device that does not have high bus utilization (e.g. small data rates). The DMA Handshake module allows support of the DMA function of the IP, but does not inherently require the use of the optional DMA Engine to effect the proper behavior. The point of this IPIF example is to illustrate the power to choose various options while considering other variables in the system, such as performance, area, and speed.

### F: Master with Slave Control Register and DMA Engine to New IP

Section F in [Figure 3-4](#) shows a typical instance of a master/slave IPIF. In general, most master IP devices also require slave interfaces to control the master interface. This example assumes that the a new piece of IP has been created that requires both master and slave access. Additionally, the optional DMA engine is used in this example to offload the CPU from the data transfers. The IPIF Slave Control Register module is used to communicate to the slave side of the IPIF backplane. This offers quick and easy memory mapped access to the internal registers of the IP, including the DMA engine registers.

### G: Bus-to-Bus Bridges

Section G in [Figure 3-4](#) illustrates a pair of bus-to-bus bridges. In this case, between Processor Local Bus (PLB) and the OPB. One bridge provides master access to the OPB, while the other bridge provides slave access.

## Design Considerations

IPIF modules can be implemented with many different options. The available options and related issues are addressed in this section.

### DMA Engine

The optional capabilities of the DMA engine are illustrated throughout [Figure 3-4](#). For example, the DMA engine in the IPIF Slave SRAM module connected to the 16550 UART

only requires very basic DMA movement operations. In contrast, the Ethernet MAC requires packet-aware DMA for highest bandwidth transmission and reception. While a single DMAC could be instantiated to handle both, it can complicate how embedded software handles the device. Xilinx's recommended system uses distributed DMA engines, one per device, to ensure a clean device driver environment. Additionally, distributed DMA will enhance performance of the overall system since many independent agents can be simultaneously active. Having optional DMA engine capabilities means that only the logic required by the IP is paid for in the FPGA fabric. (More information on DMA engine capabilities will appear in the IPIF Architectural Specification, *not yet released*.)

## Interrupts

In addition to the DMA engine capabilities, the IPIF backplane can handle various aspects of interrupts for the system, as shown in **Figure 3-5**. Existent IPs typically have their own interrupt pins and sets of registers internal to the IP for status and control of interrupts. However, this infrastructure must be designed for new IPs. The IPIF model allows for all of this and more. The IPIF model generally allows from 0 to 8 independent interrupts. Two basic modes can be optionally enabled: existent IP mode, where a single INT pin is provided from the IP to the IPIF; and new IP mode where up to 8 INT pins are provided and the register infrastructure is built in the IPIF. The parameters to the IPIF modules permit the FPGA resources to be optimized to the minimum required by the IP. (More information on interrupts appears in the IPIF Architectural Specification, *not yet released*.)

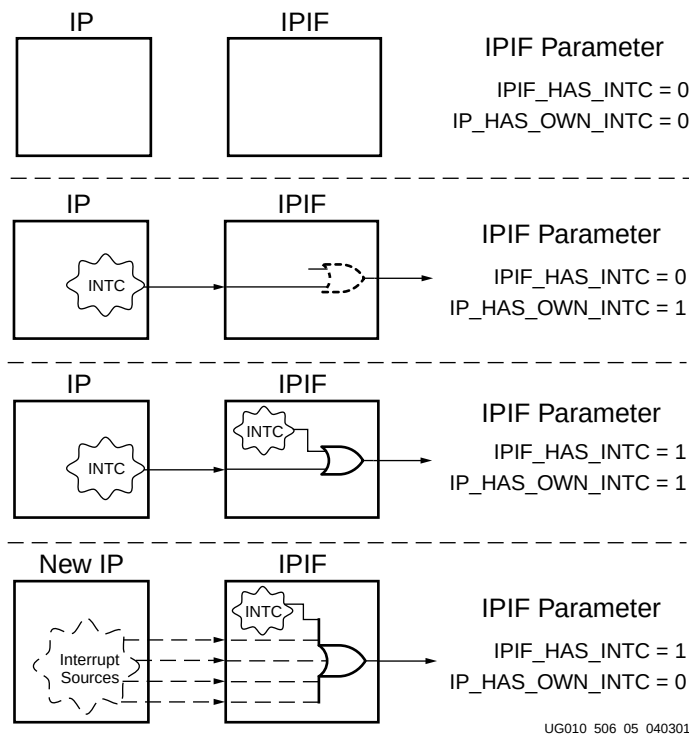


Figure 3-5: Example of Three Interrupt Scenarios

## Bus Arbiter and Bridges

Every on-chip bus that contains multiple masters requires some form of arbitration control. Section G in **Figure 3-4** illustrates an example of the OPB arbiter. Arbiters typically have a variety of parameters that can be statically or dynamically set to address system integration issues such as priority control and pipelining. (See the IBM CoreConnect™ PLB Architectural Specification for additional information on both PLB and OPB arbiters.)

Most embedded computing systems use layers of bus hierarchy to separate the high-speed devices from the low-speed devices. Typically, the CPU and memory sit on one high speed, lightly loaded bus, while the other elements reside on a heavily loaded, slower bus. To communicate between the two, a bridge-in/bridge-out function is used. The bridge out allows the CPU to act as a master to the slower bus. The bridge in function allows devices (such as DMA engines) to act as a master on the higher speed bus. Typically, to talk to system memory. The IPIF example in section G of Figure 3-4 illustrates both bridge-in and bridge-out functions of a more complex system.

## Data Bus Width

The bus side of the IPIF is always 32 bits. Since an IPIF module can be 8, 16, or 32 bits wide, some unexpected behavior may occur. If a bus master issues a request for a word transfer (32 bits) across the bus to an IPIF module that implements only a byte-wide IP data width, the IPIF will sequence through the data as four separate requests to the IP. That is, four separate request/acknowledge cycles will occur between the IP and IPIF.

## Retry, Error, and Timeout Suppress

All IPIF Slave modules permit the IP to tell the bus to retry, or that an error has occurred. The IP logic tells the IPIF this by asserting the **IP2Bus\_Retry** or **IP2Bus\_Error** signal, with or without the appropriate acknowledge signal for the intended cycle. Additionally, the IP may suppress the normal timeout (16 bus clocks) mechanism for transfers that take longer on the bus. The use of the **IP2Bus\_ToutSup** signal is not recommended unless absolutely necessary since it has a detrimental effect on bus bandwidth.

TBD

UG010\_506\_06\_040301

Figure 3-6: TBD

Figure 3-6 shows the relationship of the **IP2Bus\_ToutSup**, **IP2Bus\_Error**, and **IP2Bus\_Retry**. The **IP2Bus\_Error** signal is used to indicate to the master (which initiated the transaction) that some unspecified error has occurred.

The **IP2Bus\_Retry** is not a likely requirement in most applications. It is used to signal the bus master to retry the bus cycle, forcing the bus master off the bus. The master then rearbitrates for the bus and attempts the cycle again. In general, retry is only used for deadlock conditions, such as when a bus master attempts to access a bus master/slave while the bus master/slave is attempting a cycle of its own.

**IP2Bus\_Retry** is sometimes used as a means to hold off further bus access to the IP until the bus is no longer busy. Typically, a bus cycle initiates an operation that forces the IP to a busy condition. Further bus cycles could result in inappropriate behavior, therefore the IP issues **IP2Bus\_Retry**. This use would strain available bus bandwidth, but can be used to implement a primitive semaphoring mechanism. The **IP2Bus\_Retry** signal would be issued by the IP while the device is not ready, and the cycle would be acknowledged once the IP was ready. Again, this is very expensive in bus bandwidth, but the use is not prohibited.

## IPIF Module Specifications

### Slave DMA Handshake Module

This section covers the following topics:

- **Example Slave DMA Handshake Application**

- **Generic Slave DMA Handshake Model**
- **Slave DMA Handshake Signal Protocol**
- **Slave DMA Handshake Signal List**
- **Slave DMA Handshake Parameters**

The simplest IPIF slave module is the DMA Handshake module. This module is typically used in concert with one of the other slave modules to connect existing IP to a bus. It uses two simple handshake signals, **IP2DMA\_Req** and **DMA2IP\_Ack**, along with two unidirectional data buses. This permits the IP to request service from the optional DMA engine, which can be built into an IPIF module. The IPIF Slave DMA Handshake module is not recommended for new IP designs since its functionality can be handled directly by the optional DMA engine in other modules.

### Example Slave DMA Handshake Application

**Figure 3-7** shows an example application of the IPIF Slave DMA Handshake module. This example shows how the **DMA\_Req** and **DMA\_Ack** pins of an existent IP peripheral can connect to the IPIF module. The IP block will likely use other IPIF modules too, even though they are not shown in this example.

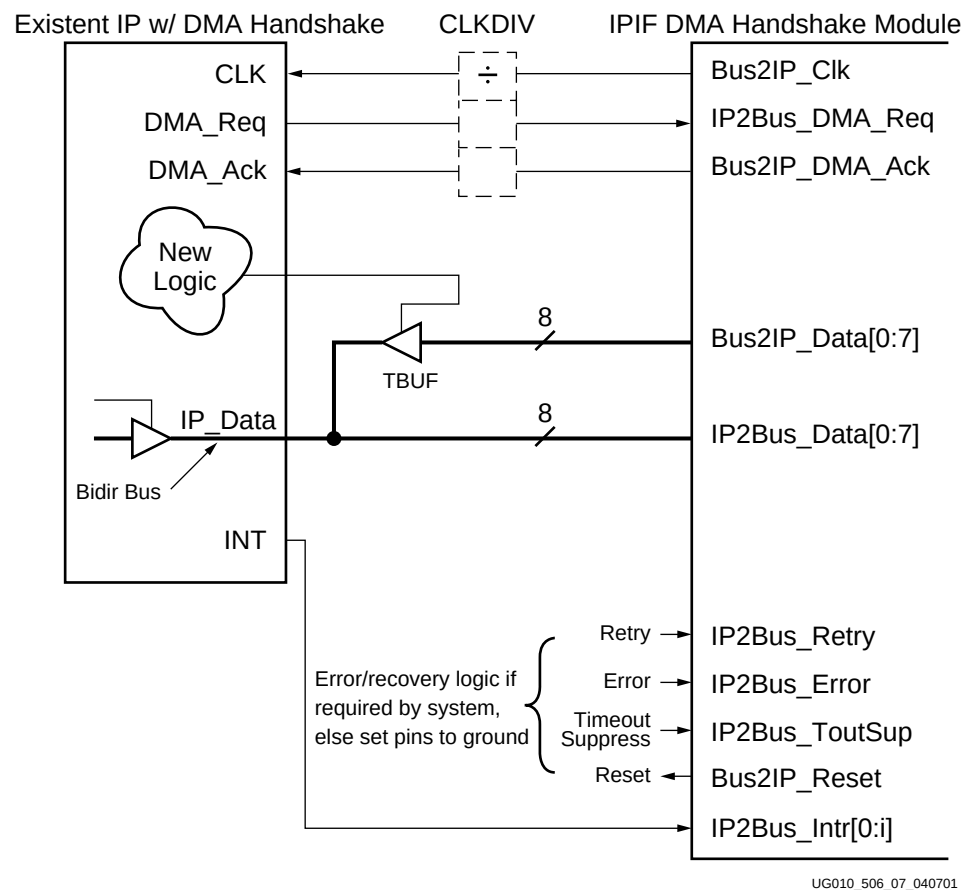


**Figure 3-7: Example Slave DMA Handshake Application**

Existent IP often utilizes a bidirectional bus instead of dual unidirectional buses. In these kinds of scenarios, the read and write sides must be separated into unidirectional buses. In some IP, the 3-state logic is implemented in the I/O where it is easy for it to cut out the I/Os. However, often the IP utilizes FPGA TBUFs to accomplish the same end. When TBUFs are used internal to the IP, removing them may be problematic.

The example application in Figure 3-7 assumes the use of TBUFs internal to the IP and, therefore, must be converted to single unidirectional data lines. The IP provides its data on the **IP\_Data** bus. **IP\_Data** is split into **Bus2IP\_Data** by using a set of TBUFs to provide isolation from the **Bus2IP\_Data** bus from the **IP\_Data** bus. The T pins on the TBUFs must be driven from within the IP using some small, new logic. The IP must tell the **Bus2IP\_Data** bus when it can write onto the **IP\_Data** bus. This arrangement permits the **Bus2IP\_Data** bus to be valid for long periods of time before a write cycle, if required. The **IP2Bus\_Data** is simply connected to the **IP\_Data** bus unless it is required to be otherwise qualified.

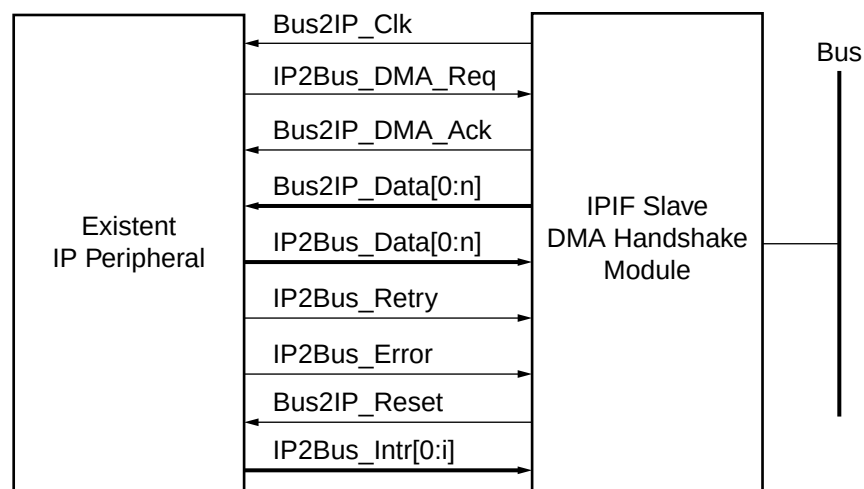
The IP's **DMA\_Req** signal connects to the **IP2Bus\_DMA\_Req** signal and the IP's **DMA\_Ack** signal connects to the **Bus2IP\_DMA\_Ack** signal on the IPIF Slave DMA Handshake module. Since this type of interface does not imply a direction of data flow, it is important that the direction is agreed to prior to any requests for service. Generally, this is done by the CPU setting specific registers inside the IP to indicate a read or write, and then doing the same in the IPIF module.

Since many existent IPs will not operate at the high clock frequency of a bus, it might be necessary to provide a clock divider and global clock buffer to provide a frequency that the IP can handle. If this occurs, the **IP2Bus\_DMA\_Req** and **Bus2IP\_DMA\_Ack** require additional circuitry to cross the clock domains. Simple synchronous set/reset flip-flops can be used to guarantee correct operation.

The 3-state control on the TBUF which drives **Bus2IP\_Data** onto the **IP\_Data** bus usually requires some small modifications to the existent IP logic. Typically, a write signal is available inside the IP, and can be used to build a read/write qualifier. Additionally, the **DMA\_Req** and **DMA\_Ack** signals can be fed into a set/reset synchronous flip-flop to build an enveloping signal. The enveloping signal, when AND'd with the write signal of the IP, can then be used to control the 3-state control of the **Bus2IP\_Data** TBUF.

## Generic Slave DMA Handshake Model

Figure 3-8 illustrates a generic instance of the IPIF Slave DMA Handshake module. The data bus widths and number of interrupts for the IP are generically specified.



UG010\_506\_08\_040301

Figure 3-8: Generic Instance of the IPIF Slave DMA Handshake Module

The IPIF Slave DMA Handshake module outputs a clock, **Bus2IP\_Clk**, that is sourced by a BUFG elsewhere in the design. The required BUFG reduces clock skew on all the synchronous elements clocked by **Bus2IP\_Clk**. This clock can be asynchronous from the

bus, but requires extra synchronization in the IPIF module. This synchronization is provided for in the IPIF module.

## Slave DMA Handshake Signal Protocol

**Figure 3-9** illustrates a simplified signal protocol diagram of the IPIF Slave DMA Handshake module. Note that the **IP2Bus\_DMA\_Req** is required to be a single clock high at the **Bus2IP\_Clk** rate. The data validity is based upon the direction agreed to between the IP and the IPIF. For IPIF read data, the data bus is **IP2Bus\_Data**. For IPIF write data, the data is **Bus2IP\_Data**.

The IPIF Slave DMA Handshake module may include internal parameters, such as response time for **Bus2IP\_DMA\_Ack**, and its width when valid (in **Bus2IP\_Clk** cycles). These parameters are used to specify a minimum time that the IPIF module can answer a request for service, and a time that the IP requires the data to be valid. Since the IP is often legacy IP, and is generally slower than the IPIF Slave DMA Handshake module, the ability to determine the response speed is critical to ensure proper operation of the IP.

The minimum time before **Bus2IP\_DMA\_Ack** will go high is controlled by the **DMA\_HNDSHK\_RESPONSE\_TIME\_MIN** parameter. This parameter guarantees that the IPIF module will not respond any sooner than the specified number of **Bus2IP\_Clk** cycles. The IPIF module may respond in more cycles, however, depending on bus activity on the bus. In the example shown in **Figure 3-9**, the **DMA\_HNDSHK\_RESPONSE\_TIME\_MIN** parameter is set to two cycles, but the IPIF does not acknowledge the cycle until the third cycle after the request.

The **DMA\_HNDSHK\_DATA\_VALID\_WIDTH** parameter sets the number of **Bus2IP\_Clk** cycles that the **Bus2IP\_DMA\_Ack** signal will be held high. During IPIF write cycles, the IPIF will drive the **Bus2IP\_Data** bits with write data for the entire duration of **Bus2IP\_DMA\_Ack**. IPIF read cycles require the IP to drive the **IP2Bus\_Data** buses with read data prior to the last positive edge of **Bus2IP\_Clk** that samples **Bus2IP\_DMA\_Ack** high. The **IP2Bus\_Data** line must continue to be held until the **Bus2IP\_DMA\_Ack** had drawn low. Read data is sampled by the IPIF on the rising edge of the **Bus2IP\_Clk** that caused **Bus2IP\_DMA\_Ack** to fall.

### Notes:

1. Data buses are shared between the IPIF Slave SRAM and IPIF Slave Control Register modules. Data qualifiers are used to indicate which module has access to the bus.
2. The specification allows driving data into the next cycle, as shown in **Figure 3-9**.

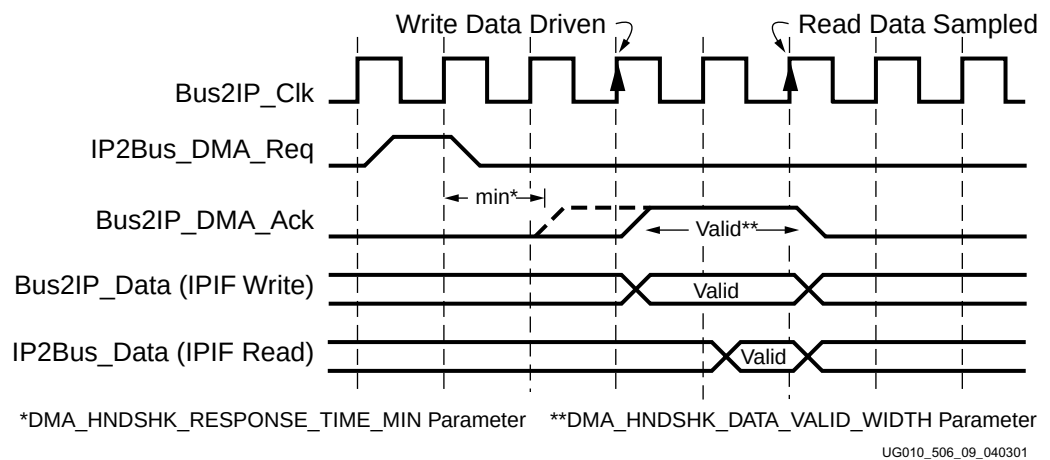


**Figure 3-9: IPIF Slave DMA Handshake Signal Protocol**

## Slave DMA Handshake Signal List

**Table 3-1** shows the name, direction, and a brief description of the signals that connect to the IP from the IPIF Slave DMA Handshake module.

**Table 3-1: IPIF Slave DMA Handshake Signals Connecting to IP**

| Name                    | Direction | Description                                                                 |
|-------------------------|-----------|-----------------------------------------------------------------------------|
| <b>Bus2IP_Clk</b>       | From IPIF | Clock source (from global buffer)                                           |
| <b>Bus2IP_Reset</b>     | From IPIF | Active-high synchronous reset source (from global buffer)                   |
| <b>IP2Bus_Intr[0:i]</b> | To IPIF   | Interrupt input from IP to IPIF                                             |
| <b>IP2Bus_Error</b>     | To IPIF   | Error signal from IP to IPIF (valid only during a data acknowledge cycle)   |
| <b>IP2Bus_Retry</b>     | To IPIF   | Indicates IP wants master to retry the cycle                                |
| <b>IP2Bus_ToutSup</b>   | To IPIF   | Forces the suppression of watch dog timeout on the bus                      |
| <b>Bus2IP_Data[0:n]</b> | From IPIF | IPIF Write data (where n = IPIF_DATA_BUS_WIDTH -1)                          |
| <b>IP2Bus_Data[0:n]</b> | To IPIF   | IPIF Read data (where n = IPIF_DATA_BUS_WIDTH -1)                           |
| <b>Bus2IP_BE[0:b]</b>   | From IPIF | Byte enable, 1 = byte lane valid (where b = IPIF_NUMBER_OF_BYTE_ENABLES -1) |
| <b>IP2Bus_DMA_Req</b>   | To IPIF   | DMA handshake transfer request from IP                                      |
| <b>Bus2IP_DMA_Ack</b>   | From IPIF | DMA handshake transfer acknowledge from IPIF                                |



## Slave DMA Handshake Parameters

**Table 3-2** shows the parameters that can be selected for the IPIF Slave DMA Handshake module..

**Table 3-2: IPIF Slave DMA Handshake Module Parameters**

| Affects                                 | Parameter                                                                                                                                                       | Value            | Type    |
|-----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|---------|
| General IPIF                            | IPIF_DATA_BUS_WIDTH<br>Sets the size of the data bus for IPIF (where “n” in <b>Bus2IP_Data[0:n]</b> or <b>IP2Bus_Data[0:n]</b> is equal to IPIF_DATA_BUS_WIDTH) | 8, 16, or 32     | number  |
|                                         | IPIF_NUMBER_OF_BYTE_ENABLES<br>Sets the number of byte enables                                                                                                  | 1, 2, or 4       | number  |
|                                         | IPIF_NUMBER_OF_INTR<br>Sets the number of interrupts the IP provides to the IPIF (where “i” in <b>IP2Bus_Intr[0:i]</b> is equal to IPIF_NUMBER_OF_INTR)         | 0 to 8           | number  |
|                                         | IPIF_INTR_ID<br>Sets the unique interrupt ID for this IPIF                                                                                                      | 16 bits          | number  |
|                                         | IPIF_HAS_INTC<br>Sets whether the IPIF has a built-in interrupt controller (See <b>Figure 3-5</b> for more information)                                         | 0 or 1, 1 = true | boolean |
|                                         | IP_HAS_OWN_INTC<br>Sets whether the IP has its own interrupt controller (See <b>Figure 3-5</b> for more information)                                            | 0 or 1, 1 = true | boolean |
| General IPIF Slave DMA Handshake Module | DMA_HNDSHK_RESPONSE_TIME_MIN<br>Sets the minimum number of <b>Bus2IP_Clk</b> cycles in which the IPIF will respond with a DMA_ACK                               | 0 to 255         | number  |
|                                         | DMA_HNDSHK_DATA_VALID_WIDTH<br>Sets the number of <b>Bus2IP_Clk</b> cycles that the data will remain valid during DMA handshaking                               | 0 to 255         | number  |

## Slave Control Register Module

This section covers the following topics:

- **Example Slave Control Register Application**
- **Generic Slave Control Register Model**
- **Slave Control Register Signal Protocol**
- **Slave Control Register Signal List**
- **Slave Control Register Parameters**

The IPIF Slave Control Register module provides a set of signals which permit basic control registers to directly attach to a bus. The overall number of registers, number of bits in each specified register, and direction for each bit of each register are parameters that may be specified when instantiating this module.

### Example Slave Control Register Application

**Figure 3-10** contains an example application of the IPIF Slave Control Register module. It illustrates how read/write, read only, and write only registers can be implemented. Typical



systems contain many kinds of registers, each of which can be easily connected to the IPIF Slave Control Register module.

The read/write (R/W) register shown in [Figure 3-10](#) is 24 bits wide, and is centered about the 32-bit data bus as D[8:31] logically. Since it is a read/write register, it requires data into the flip-flops from both user logic (if present) and the IPIF module. Accordingly, a multiplexer, controlled by the user logic, allows the write update to the register to be selected between the IP and IPIF. Also note that the CE pin of the R/W register is ORed together with the CE (**Bus2IP\_RegCE1**) provided by the IPIF and the CE from the user logic. Also note that the Q outputs of the R/W register are available to user logic as needed, and are multiplexed into the **IP2Bus\_Data** path for read access of the register.

The read-only (R/O) register is read back across the bus when the address on the bus is set for the decodes of **IP2Bus\_RegCE2**. Address decoding for all the registers is contained within the IPIF Slave Control Register module and is specified by the user during instantiation. In this example, the R/O register is 16 bits and connects to **D[16:31]** logically. This R/O register is truly read only. The user logic is responsible for updating the contents of the register. While the IP logic might utilize the Q outputs of the R/O register, the IPIF Slave Control Register module can not write to the register. The R/O register's data path is multiplexed with the R/W register back to the **Bus2IP\_Data** bus of the IPIF Slave Control Register.

The write-only (W/O) register in this example is instantiated as a 24-bit register and connects to **D[0:23]** logically. Like the R/W register, the W/O register may have user logic which can write to the it. Accordingly, a mux is needed on the D input and at least an OR gate on the CE input to the register. The Q outputs of the W/O register are not available to the IPIF Slave Control Register module, making this register truly write-only. The W/O register in this example also demonstrates the use of an additional qualifying signal to update the content during IPIF write operation. A signal called **WE** causes the write to occur at a specific time in the bus cycle. Similar logic can be used to force the write to occur at any time during the bus cycle.

**Note:** Reg data bits can be anywhere in data word, and can include holes, e.g. data bits [0:3], [5], [8:12], etc. Zeros are filled in for missing bits on reads.

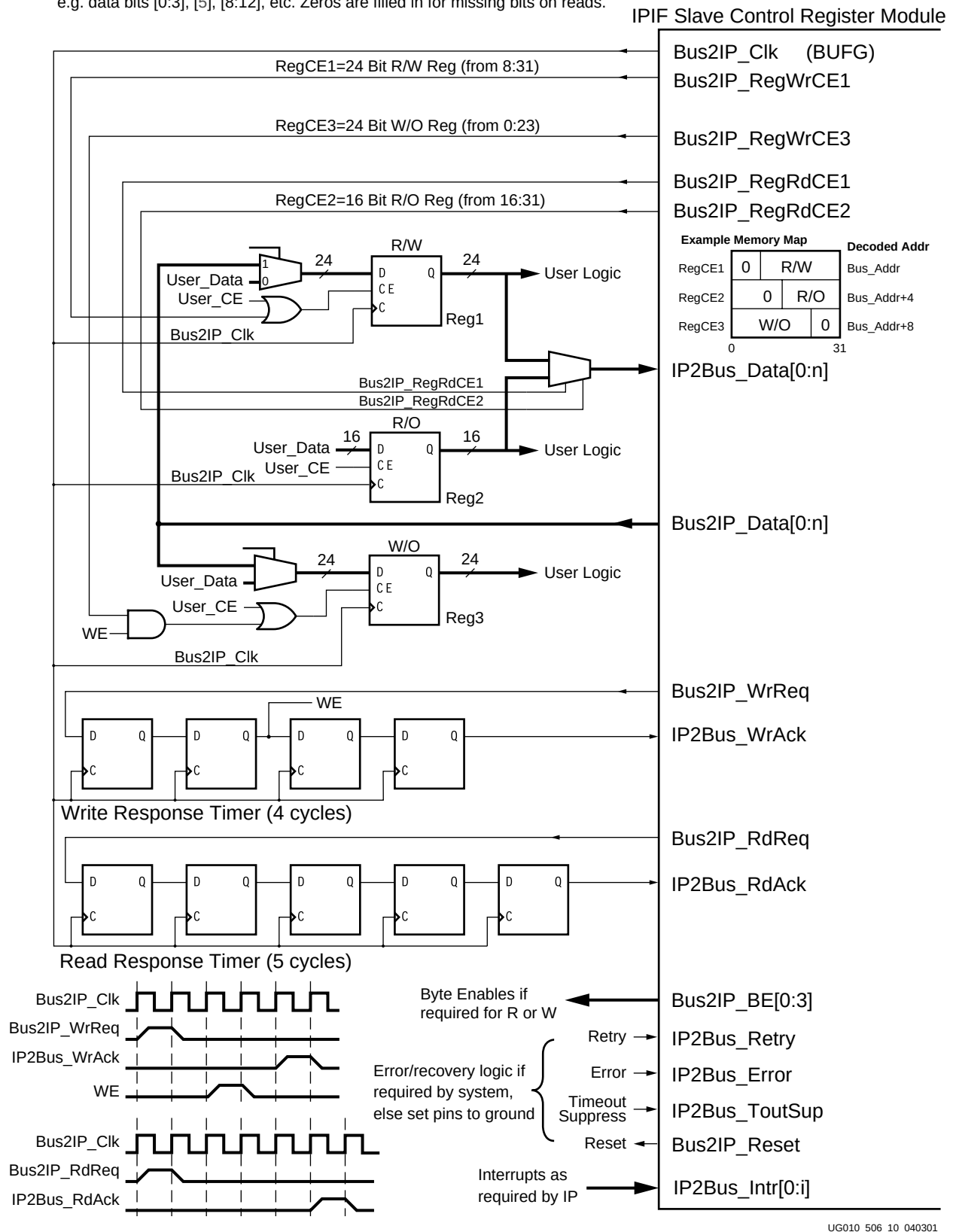


Figure 3-10: Example Application of an IPIF Slave Control Register Module

It should be noted that the IPIF Slave Control Register module does not inherently know about the read/write ability of any register that is connected to it. It is how the various registers are connected that allows for read/write, read only, or write only behavior. In addition, the width of the registers, size of the registers, etc. are defined during the instantiation of the IPIF Slave Control Register module. See [Table 3-4](#) for further information.

The IPIF Slave Control Register module can never generate both read and write requests at the same time because the bus, unlike the Processor Local Bus, does not permit simultaneous read and write transactions. Accordingly, the IP logic can be developed to look at the read and write sides independently without and fear of collision between the two sides.

[Figure 3-10](#) illustrates one simple means of answering the request for service by the IPIF Slave Control Register module. When a bus address hit (in the ranges defined by parameters) is detected in this module, the module will generate either **Bus2IP\_WrReq** or **Bus2IP\_RdReq** requesting write or read service. The connecting IP is required to issue an acknowledge when it has completed the transaction. In the case of writes, the **IP2Bus\_WrAck** signal indicates that the control register has been properly written. Reads utilize the **IP2Bus\_RdAck** signal and present data on the **IP2Bus\_Data** bus during the same cycle that the acknowledge is valid. Also shown in [Figure 3-10](#) is a simple shift register chain that allows a four-cycle write time and a five-cycle read time. For very fast systems, the requests can be tied directly to the acknowledge signals, assuming that the data can be dealt with in the given time. By using FPGA SRL16 elements, the shift registers can be built in a single look-up table (LUT) instead of using a multitude of LUT flip-flops. For more information on how to do this, see Chapter 2, page 211 of the Virtex-II Platform FPGA Handbook (UG002 version 1.0) located at:

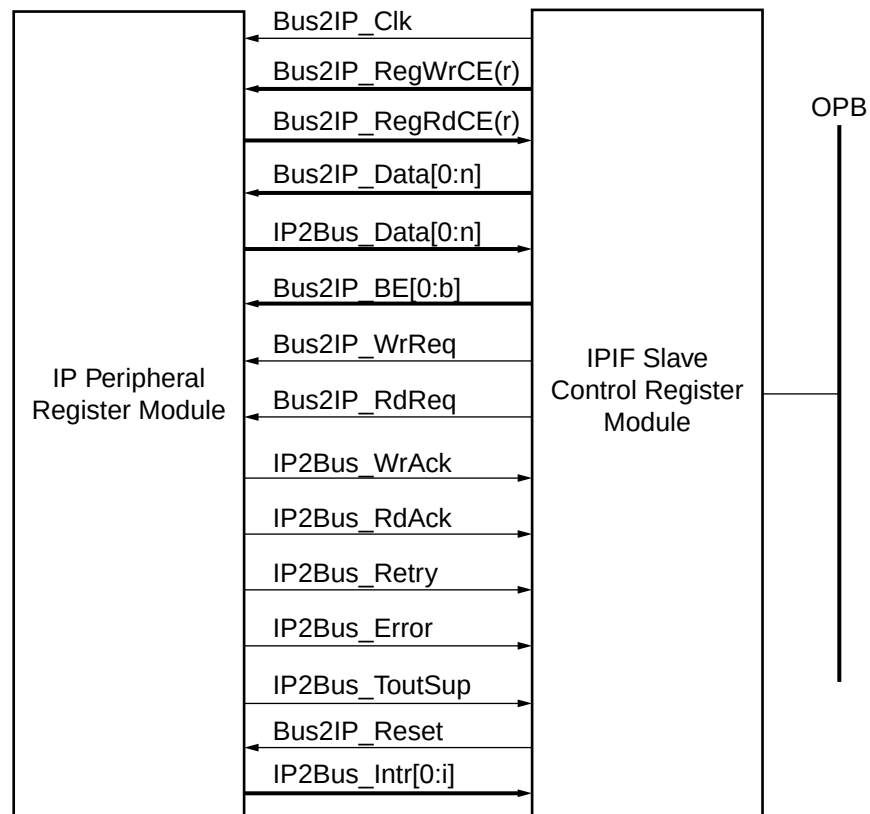
[http://www.xilinx.com/products/virtex/handbook/ug002\\_ch2\\_srlut.pdf](http://www.xilinx.com/products/virtex/handbook/ug002_ch2_srlut.pdf).

The IPIF Slave Control Register module also provides the ability to write only certain bytes of the registers. The **Bus2IP\_BE** byte enable signals can be used to qualify which bytes are valid during the transfer. [Figure 3-10](#) does not illustrate this usage, however, the byte enables can simply be used as additional qualifier terms in the clock enable term of the register.

All IPIF slave modules, including the Control Register module, permit the IP to tell the bus to retry, or that an error has occurred. The IP logic tells the IPIF this by asserting the **IP2Bus\_Retry** or **IP2Bus\_Error** signal, with or without the appropriate acknowledge signal for the intended cycle. Additionally, the IP may suppress the normal timeout (16 bus clocks) mechanism for transfers that will take longer on the bus. The use of the **IP2Bus\_ToutSup** signal is not recommended unless absolutely necessary, since it reduces bus bandwidth.

## Generic Slave Control Register Model

[Figure 3-11](#) illustrates a generic instance of the IPIF Slave Control Register module. The data bus widths, number of registers, number of byte enables, and number of interrupts for the IP are generically specified.



UG010\_506\_11\_040301

Figure 3-11: Generic Instance of the IPIF Slave Control Register

The IPIF Slave Control Register module outputs a clock, **Bus2IP\_Clk**, that is sourced by a BUFG elsewhere in the design. The required BUFG reduces clock skew on all the synchronous elements clocked by **Bus2IP\_Clk**. This clock can be asynchronous from the bus, but requires extra synchronization in the IPIF module. This synchronization is provided for in the IPIF module.

The registers generate a unique decode off the internal address decoder of the IPIF. The number of bits available for each register depends on the data width of the IPIF module. For example, if an 8-bit IPIF Slave Control Register module is instantiated, all registers are limited to a maximum of 8 bits. Each register can be from 0 to 8 bits in this case. The 0 case seems odd at first; however, it may be advantageous for software to “skip” an address location, therefore, 0-bit registers are allowed. Based upon the size of the data bus, the IPIF Slave Control Register module assigns address decodes for the **Bus2IP\_RegWrRdCE** signals. If the data bus is 8 bits, then each **Bus2IP\_RegWrRdCE** will be up to a byte wide, and will start at the base address of the IPIF Slave Control Register module, and increment by one. If the data bus is 16 bits, then each **Bus2IP\_RegWrRdCE** will be up to 16 bits wide, and will increment by two from the base address. Similarly, for 32-bit data bus, each register decode is offset by four from the base address.

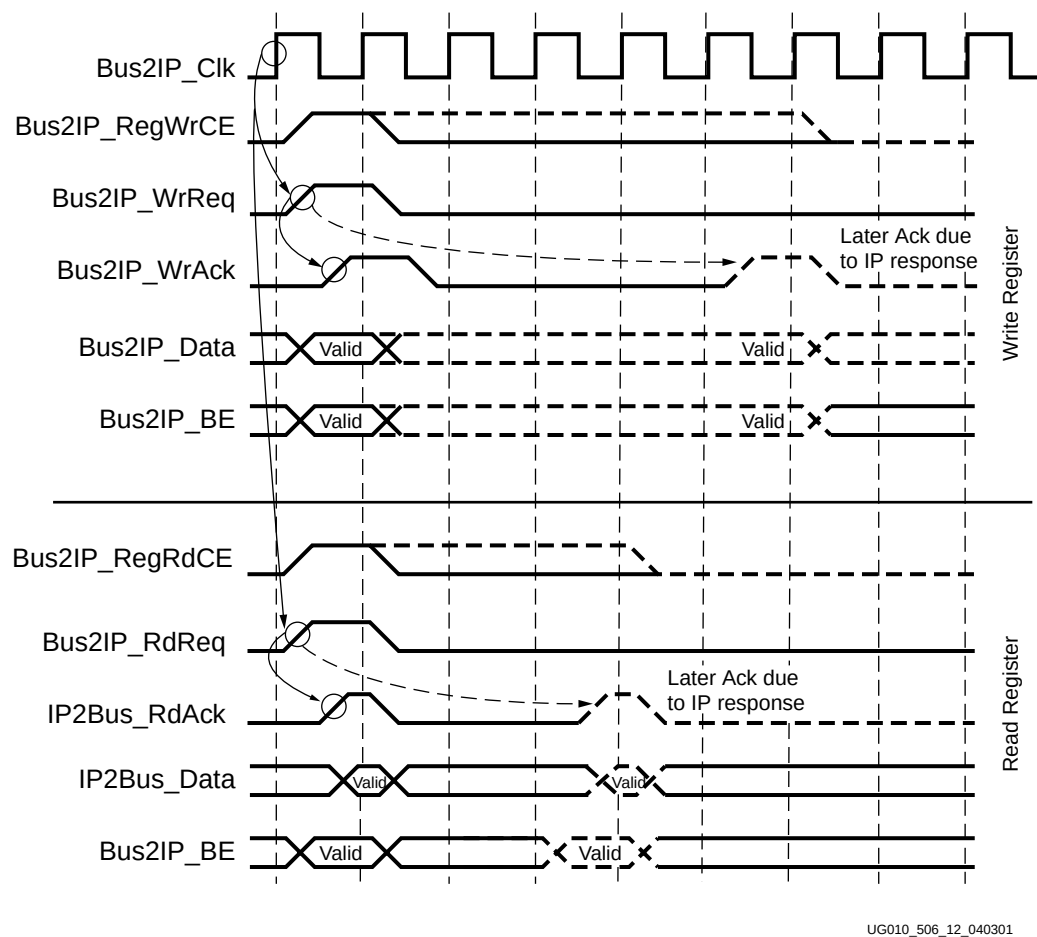
The indexing of the **Bus2IP\_RegWrCE** or **Bus2IP\_RegRdCE** is based upon the bus size. Regardless of whether the bus is 8 bits or 32 bits, **Bus2IP\_RegWrRdCE** will increment by one for each instantiated register. There will be holes for zero-bit registers, however. (For example, if register 3 is zero bits and register 4 is 12 bits, no register 3 clock enable decodes are provided.)

The IPIF Slave Control Register module allows the IP implementor to emplace only the register bits required for the function. This minimizes the logic required for the implementation of registers. Since undriven read bits left are forced to a zero inside the IPIF module, even data multiplexing requirements are eased.

Control over this flexibility is represented as an array of four basic parameters. Each register has control over the total number of bits (0 to 32 data bus width) that can be contained within the register. Additionally, each register can identify the bit-wide position of every bit in the register. Each bit position can also be marked as readable and/or writable by the IPIF.

## Slave Control Register Signal Protocol

Figure 3-12 illustrates the IPIF Slave Control Register module protocol, highlighting the simplicity of both read and write transactions. Note that all signals are active-high true, and are referenced to the positive edge of the **Bus2IP\_Clk** signal. These states are generally more favorable to FPGA logic implementation.



UG010\_506\_12\_040301

Figure 3-12: Timing Diagrams for Slave IP Module Register Protocol  
Read and Write

## Write Transactions

Write transactions begin with the IPIF Slave Control Register issuing a **IP2Bus\_WrReq** on a positive edge of **Bus2IP\_Clk**. The data to be written is presented on **Bus2IP\_Data**, qualified by **Bus2IP\_RegRdWrCE(r)** to indicate a register IPIF cycle, and is byte-qualified by the **Bus2IP\_BE** signals. (If high, the byte lane is valid; if low, ignore the data.) When the IP has correctly stored the data, the **IP2Bus\_WrAck** signal is asserted. Two cases are

illustrated in **Figure 3-12**: immediate acknowledgement, and extended **IP2Bus\_WrAck** by five clock cycles. The limit to issuing **IP2Bus\_WrAck** is approximately 10 cycles<sup>(1)</sup> after **Bus2IP\_WrReq**, unless the **IP2Bus\_ToutSup** signal is held high. If **IP2Bus\_ToutSup** is used, it must go low in the same cycle that **IP2Bus\_WrAck** transitions from high to low.

**Notes:**

1. Actual number of cycles to be determined.

### Read Transactions

Read transactions are very similar to write transactions in the IPIF Slave Control Register module. Reads utilize the **Bus2IP\_RdReq** and **IP2Bus\_RdAck** counterparts to the write signals and **Bus2IP\_RegRdCE(r)**. The data must be presented to the **IP2Bus\_Data** bus during the cycle that **Bus2IP\_RdAck** is valid. The **Bus2IP\_BE** signals can be used to mask read data values, but this is not required. The master that generated the transaction on the bus is always required to pull only the data it wants from the bus. The **Bus2IP\_BE** signals are generally only used for writes.

## Slave Control Register Signal List

**Table 3-3** shows the name, direction, and a brief description of the signals that connect to the IP from the IPIF Slave Control Register module.

**Table 3-3: Signals for the IPIF Slave Control Register Module**

| Name                    | Dir       | Description                                                                                             |
|-------------------------|-----------|---------------------------------------------------------------------------------------------------------|
| <b>Bus2IP_Clk</b>       | From IPIF | Clock source (from global buffer)                                                                       |
| <b>Bus2IP_Reset</b>     | From IPIF | Active-high synchronous reset source (from global buffer)                                               |
| <b>IP2Bus_Intr[0:i]</b> | To IPIF   | Interrupt input from IP to IPIF                                                                         |
| <b>IP2Bus_Error</b>     | To IPIF   | Error signal from IP to IPIF (valid only during a data acknowledge cycle)                               |
| <b>IP2Bus_Retry</b>     | To IPIF   | Indicates IP wants master to retry the cycle                                                            |
| <b>IP2Bus_ToutSup</b>   | To IPIF   | Forces the suppression of watch dog timeout on the bus                                                  |
| <b>Bus2IP_Data[0:n]</b> | From IPIF | IPIF Write data (where n = IPIF_DATA_BUS_WIDTH - 1)                                                     |
| <b>IP2Bus_Data[0:n]</b> | To IPIF   | IPIF Read data (where n = IPIF_DATA_BUS_WIDTH - 1)                                                      |
| <b>Bus2IP_BE[0:b]</b>   | From IPIF | Byte enable, 1 = byte lane valid (where b = IPIF_NUMBER_OF_BYTE_ENABLES - 1)                            |
| <b>Bus2IP_WrReq</b>     | From IPIF | Write request from IPIF to IP, single clock high                                                        |
| <b>IP2Bus_WrAck</b>     | To IPIF   | Acknowledge that write data has been taken from <b>Bus2IP_Data[0:n]</b> , single <b>Bus2IP_Clk</b> high |
| <b>Bus2IP_RdReq</b>     | From IPIF | Read request from IPIF to IP, single clock high                                                         |
| <b>IP2Bus_RdAck</b>     | To IPIF   | Acknowledge that read data has been placed on <b>IP2Bus_Data[0:n]</b> , single <b>Bus2IP_Clk</b> high   |
| <b>Bus2IP_RegCE(r)</b>  | From IPIF | Clock enable of decoded "r" register (where r = 0 to IPIF_NUMBER_OF_REGS - 1)                           |

## Slave Control Register Parameters

**Table 3-4** shows the parameters that can be selected for the IPIF Slave DMA Handshake module.

**Table 3-4: IPIF Slave Control Register Parameters**

| Affects                                    | Parameter                                                                                                                                                       | Value                             | Type    |
|--------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|---------|
| General IPIF                               | IPIF_DATA_BUS_WIDTH<br>Sets the size of the data bus for IPIF (where “n” in <b>Bus2IP_Data[0:n]</b> or <b>IP2Bus_Data[0:n]</b> is equal to IPIF_DATA_BUS_WIDTH) | 8, 16, or 32                      | number  |
|                                            | IPIF_NUMBER_OF_BYTE_ENABLES<br>Sets the number of byte enables                                                                                                  | 1, 2, or 4                        | number  |
|                                            | IPIF_NUMBER_OF_INTR<br>Sets the number of interrupts the IP provides to the IPIF (where “i” in <b>IP2Bus_Intr[0:i]</b> is equal to IPIF_NUMBER_OF_INTR)         | 0 to 8                            | number  |
|                                            | IPIF_INTR_ID<br>Sets the unique interrupt ID for this IPIF                                                                                                      | 16 bits                           | number  |
|                                            | IPIF_HAS_INTC<br>Sets whether the IPIF has a built-in interrupt controller (See <b>Figure 3-5</b> for more information)                                         | 0 or 1, 1 = true                  | boolean |
|                                            | IP_HAS_OWN_INTC<br>Sets whether the IP has its own interrupt controller (See <b>Figure 3-5</b> for more information)                                            | 0 or 1, 1 = true                  | boolean |
| General IPIF Slave Control Register Module | IPIF_NUMBER_OF_REGS<br>Sets the total number of registers at the bus width set by IPIF_DATA_BUS_WIDTH                                                           | 1 to 255                          | number  |
|                                            | IPIF_REG_BASE_ADDR<br>Sets the base address where IPIF registers will start in memory                                                                           | 32-bit decode                     | number  |
|                                            | IPIF_REG_BASE_ADDR_BIT_ENBL[0:31]<br>Allows specification of which address bits to decode in IPIF_REG_BASE_ADDR                                                 | 1 = decode respective address bit | mask    |

Table 3-4: IPIF Slave Control Register Parameters (Continued)

| Affects                                            | Parameter                                                                                                                                                             | Value                                     | Type   |
|----------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|--------|
| IPIF Slave Control Register (Arrays of Parameters) | IPIF_REGx_NUMBER_OF_BITS<br>Defines the number of bits for each register (where x = 0 to IP_NUMBER_OF_REGS -1)                                                        | ordinal between 0 and 32                  | number |
|                                                    | IPIF_REGx_DATA_BIT_VALID_MASK[0:n]<br>Defines which bits in each register are physically present (where x = 0 to IP_NUMBER_OF_REGS -1 and n = IPIF_DATA_BUS_WIDTH -1) | 1 = bit position is used in this register | mask   |
|                                                    | IPIF_REGx_READABLE_BITS[0:n]<br>Defines which bits in each register are readable by the IPIF (where x = 0 to IP_NUMBER_OF_REGS -1 and n = IPIF_DATA_BUS_WIDTH -1)     | 1 = bit position will be readable by IPIF | mask   |
|                                                    | IPIF_REGx_WRITEABLE_BITS[0:n]<br>Defines which bits in each register are writable by the IPIF (where x = 0 to IP_NUMBER_OF_REGS -1 and n = IPIF_DATA_BUS_WIDTH -1)    | 1 = bit position will be writable by IPIF | mask   |

## Slave SRAM Module

This section covers the following topics:

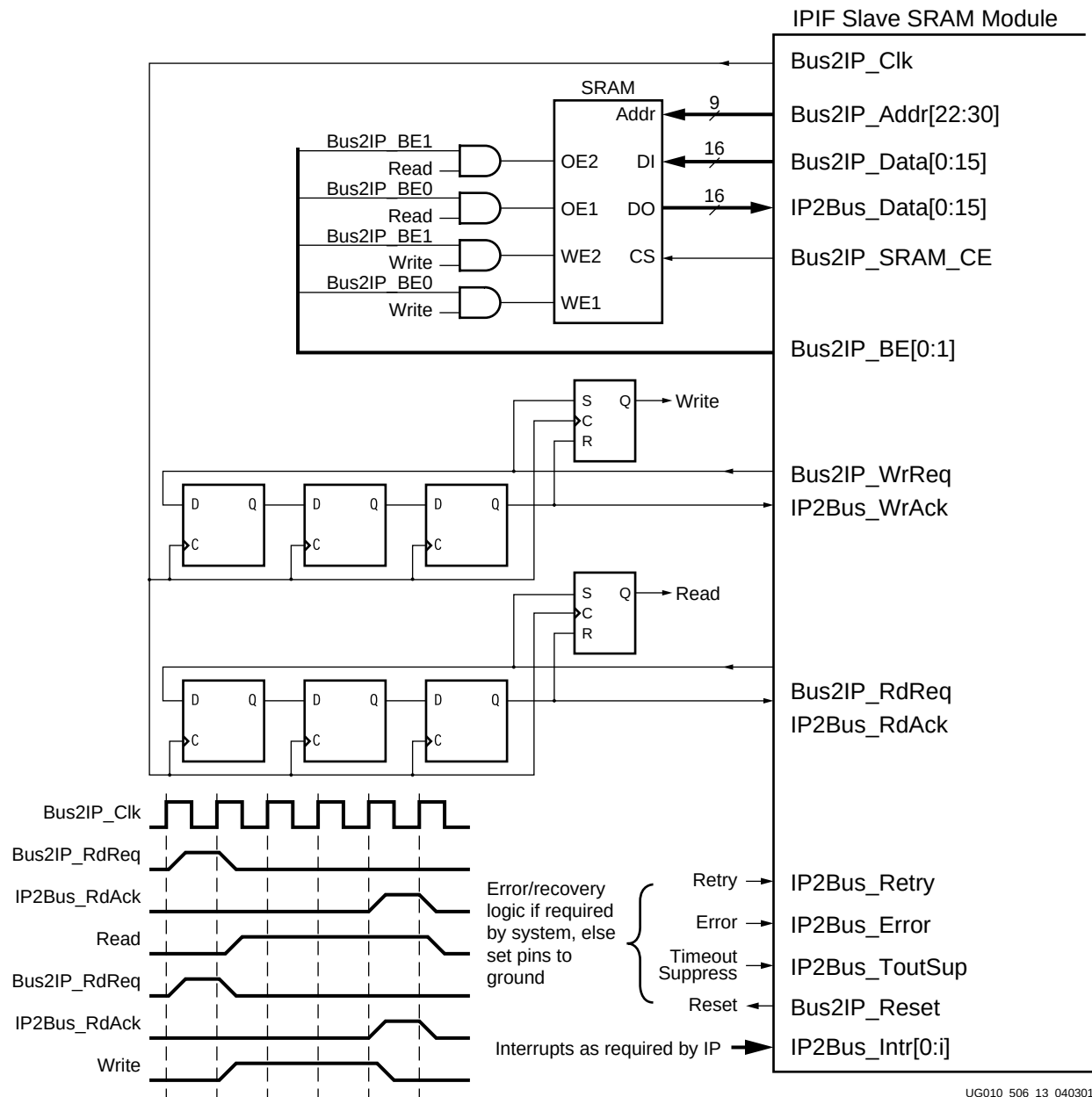
- **Example Slave SRAM Application**
- **Generic Slave SRAM Model**
- **Slave SRAM Signal Protocol**
- **Slave SRAM Signal List**
- **Slave SRAM Parameters**

The IPIF Slave SRAM module is virtually identical to the IPIF Slave Control Register module. The primary difference is that the IPIF Slave SRAM module also includes an address, and only provides a single clock enable for the range of valid addresses. The similarity between the two interfaces is intentional. It is recognized that many forms of IP require both registers and random accessible storage space. Accordingly, the IPIF Slave SRAM module and IPIF Slave Control Register module can be easily combined. In high-performance systems, a single IPIF Slave SRAM module per IP is necessary; whereas in lower performance systems, several IPs can share one IPIF Slave SRAM module.

### Example Slave SRAM Application

**Figure 3-13** illustrates an example application of the IPIF Slave SRAM module. This example assumes use of a 16-bit wide asynchronous SRAM with some additional logic surrounding it to handle byte data, and setting of the read and write cycle times. While this example shows connection to a real SRAM, the IPIF Slave SRAM module can be connected to any IP that requires a multitude of addresses for its operation. For example, an existent disk drive controller whose interface is a microprocessor memory mapped device can take advantage of this IPIF module with little modification.





UG010\_506\_13\_040301

Figure 3-13: Example Application of IPIF Slave SRAM Module

The **Bus2IP\_Addr** bus is used to broadcast the address of the bus Slave transaction to the SRAM. The SRAM in this example is organized as 512 x 16 or 1 KB total. In this case, the byte address is the lower 10 bits of the address, **Bus2IP\_Addr[22:31]**. Since the example uses half words (16 bit quantities) and the IPIF Slave SRAM module always provides byte addresses, the lower order address line is left unconnected, leaving the next nine address lines to provide the 512 half words.

Since most systems require the ability to read and write individual bytes of data from an SRAM, the IPIF Slave SRAM module provides byte enables by way of the **Bus2IP\_BE** signals. In the example in [Figure 3-13](#), these byte enables are used to qualify both writes and reads by way of the active-high WE and OE pins. This permits the SRAM to be read and written a single byte at a time, if required. Byte enable usage on read cycles is not required, since the IPIF module will automatically handle the read data alignment. (More

information on data alignment will appear in the IPIF Architectural Specification, *not yet released*.)

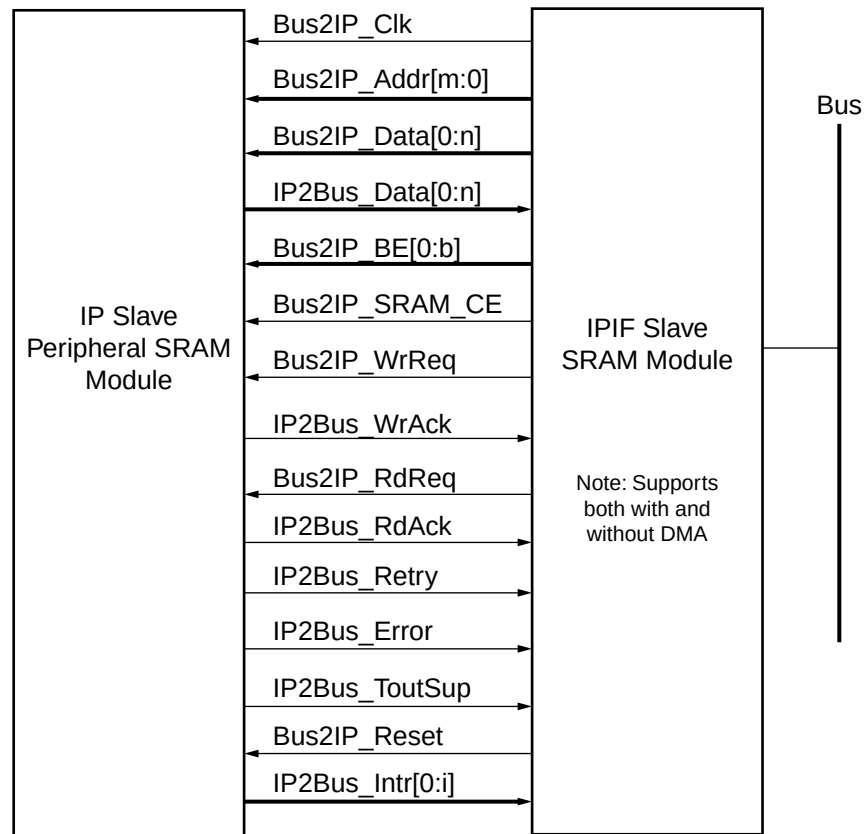
This example also shows that the **Bus2IP\_SRAM\_CE** pin is connected directly to the active-high chip select (CS) pin on the SRAM. This example assumes that the SRAM CS pin qualifies the **OE** and **WE** signals. If the SRAM does not use CS to qualify the **OE** and **WE** signals, the **Bus2IP\_SRAM\_CE** signal must qualify both reads and writes. This is a particular issue that warrants attention when the IPIF Slave SRAM module is used to talk with several ranges of memory which may be discrete SRAMs or IPs.

**Figure 3-13** also demonstrates a simple technique to generate both the acknowledgement responses and a set of enveloping signals. The acknowledgement of read and write cycles is handled by delaying the respective request signal by a known number of clocks (three in this example). This is very efficient to implement in a single FPGA look-up table (LUT) by utilizing an SRL16 model. The SRL16s allow the customer to instantiate a 1- to 16-bit shift register at the price of a single LUT.

In addition to providing the **IP2Bus\_WrAck** and **IP2Bus\_RdAck**, the outputs of the shift register can be used to build a simple enveloping signal. The write and read signals illustrated in the figure are created by flip-flops that are set by **Bus2IP\_WrReq** or **Bus2IP\_RdReq**, respectively. They are reset by the **IP2Bus\_WrReq** or **IP2Bus\_RdAck** signals, respectively. This provides a single clock delayed envelope that is valid for the transaction, and can be used to enable specific read or write operations. The read and write signals are used to qualify the **OE** and **WE** inputs to the SRAM. The write signal can also be derived as a single cycle delayed clock enable, if necessary to meet data hold time requirements (e.g., another tap can be used).

## Generic Slave SRAM Model

Figure 3-14 illustrates a generic instance of the IPIF Slave SRAM module. The data bus widths, number of address bits, number of byte enables, and number of interrupts for the IP are generically specified.



UG010\_506\_14\_030501

Figure 3-14: Slave IP Module SRAM Protocol Block Diagram

The IPIF Slave SRAM module outputs a clock, **Bus2IP\_Clk**, that is sourced by a BUFG elsewhere in the design. The required BUFG reduces clock skew on all the synchronous elements clocked by **Bus2IP\_Clk**. This clock can be asynchronous from the bus, but requires extra synchronization in the IPIF module. This synchronization is provided for in the IPIF module.

The number of address locations covered by the IPIF Slave SRAM interface, and the number of address bits required to support those address locations is specified during instantiation of the module.

The data width for the IPIF Slave SRAM interface, illustrated generically in Figure 3-14, can be configured as 8, 16 or 32 bits. The  $n$  variable in the figure is always subtracted by one from the number of bits specified by the IP\_DATA\_BUS\_WIDTH parameter.

Figure 3-14 also shows the active-high signal, **Bus2IP\_SRAM\_CE**. This signal indicates that the address presented on **Bus2IP\_Addr** is valid. Since the regions of memory decoded by the IPIF Slave SRAM module are parameterizable, it is possible that the **Bus2IP\_SRAM\_CE** may be valid in discontinuous regions. The bus addresses that will initiate the transaction on the IPIF Slave SRAM module are set during the instantiation of the module by the parameters defined in Table 3-6.

The remaining signals illustrated in Figure 3-14 are illustrated by example in Figure 3-13.

## Slave SRAM Signal Protocol

Figure 3-15 and Figure 3-16 illustrate the IPIF Slave SRAM module protocol for write and read cycles, respectively. All signals shown in both figures are active-high true, and are referenced to the positive edge of the Bus2IP\_Clk signal. These states are often most favorable to FPGA logic implementation.

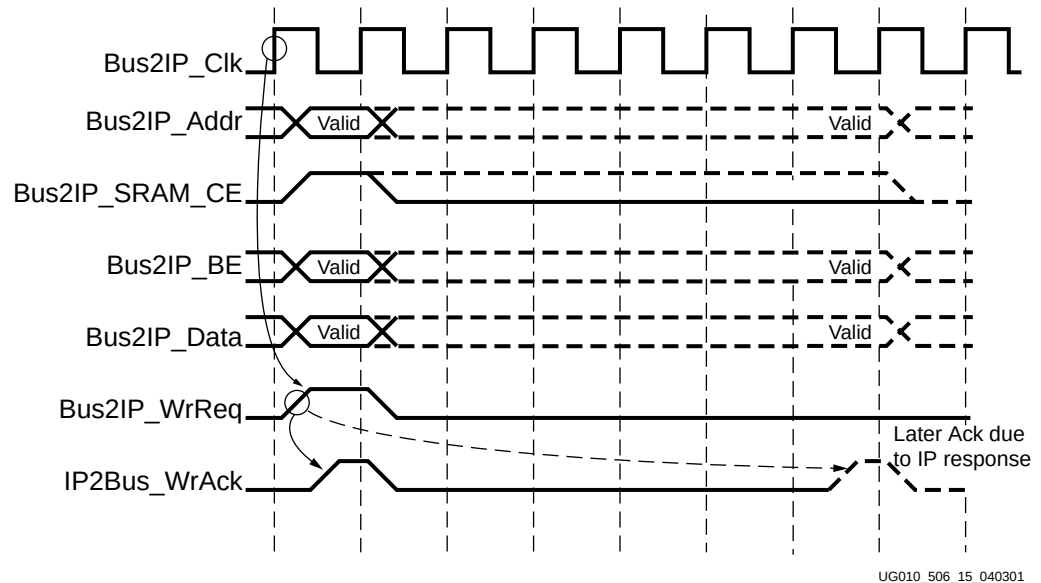


Figure 3-15: IPIF Slave SRAM Module Single Write Transaction

### Write Transactions

Write transactions begin with the IPIF Slave SRAM module issuing a **IP2Bus\_WrReq** on a positive edge of **Bus2IP\_Clk**. The data to be written is presented on **Bus2IP\_Data**, qualified by **Bus2IP\_SRAM\_CE** to indicate an SRAM cycle, and is byte qualified by the **Bus2IP\_BE** signals. (High = data in byte lane is valid; low = ignore data.) The address of the write transfer is presented on the **Bus2IP\_Addr** bus on the same rising edge of the clock that generated **Bus2IP\_WrReq**.

When the IP has correctly stored the data, it asserts the **IP2Bus\_WrAck** signal. Figure 3-15 illustrates two cases. The first case is immediate acknowledgement, and the second case extends the **IP2Bus\_WrAck** out six clock cycles. The limit to issuing **IP2Bus\_WrAck** is 10 cycles after the **Bus2IP\_WrReq**, unless the **IP2Bus\_ToutSup** signal is held high. If **IP2Bus\_ToutSup** is used, it must go low at the same time that **IP2Bus\_WrAck** transitions from high to low.

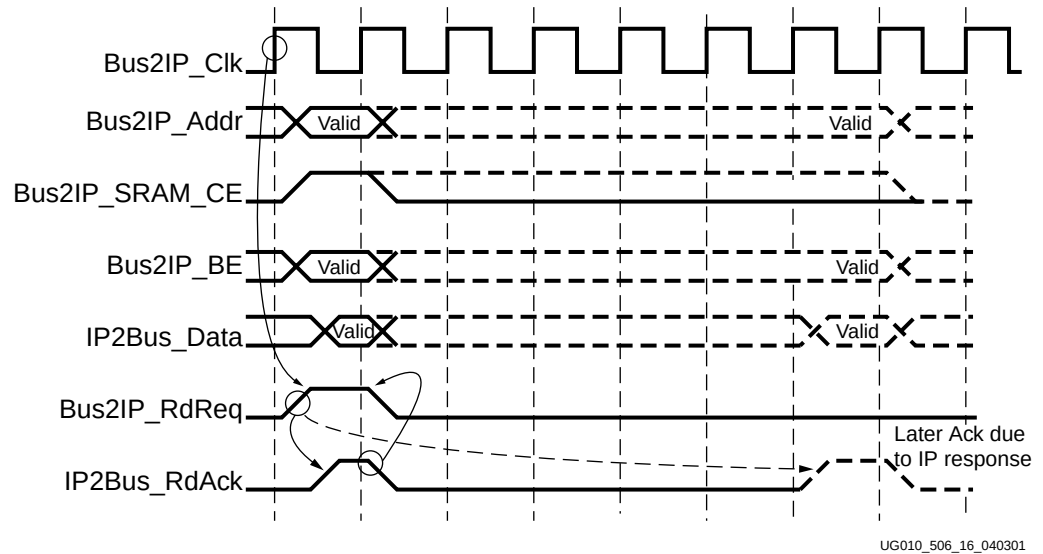


Figure 3-16: IPIF Slave SRAM Module Single Read Transaction

### Read Transactions

Read transactions are very similar to write transactions in the IPIF Slave SRAM module. Reads utilize the **Bus2IP\_RdReq** and **IP2Bus\_RdAck** counterparts to the write signals. Data must be presented to the **IP2Bus\_Data** bus during the cycle that **Bus2IP\_RdAck** is valid. The **Bus2IP\_BE** signals can be used to mask read data values, but this is not required. The master that generated the transaction on the bus is always required to pull only the data it wants from the bus. The **Bus2IP\_BE** signals are generally used only for writes.

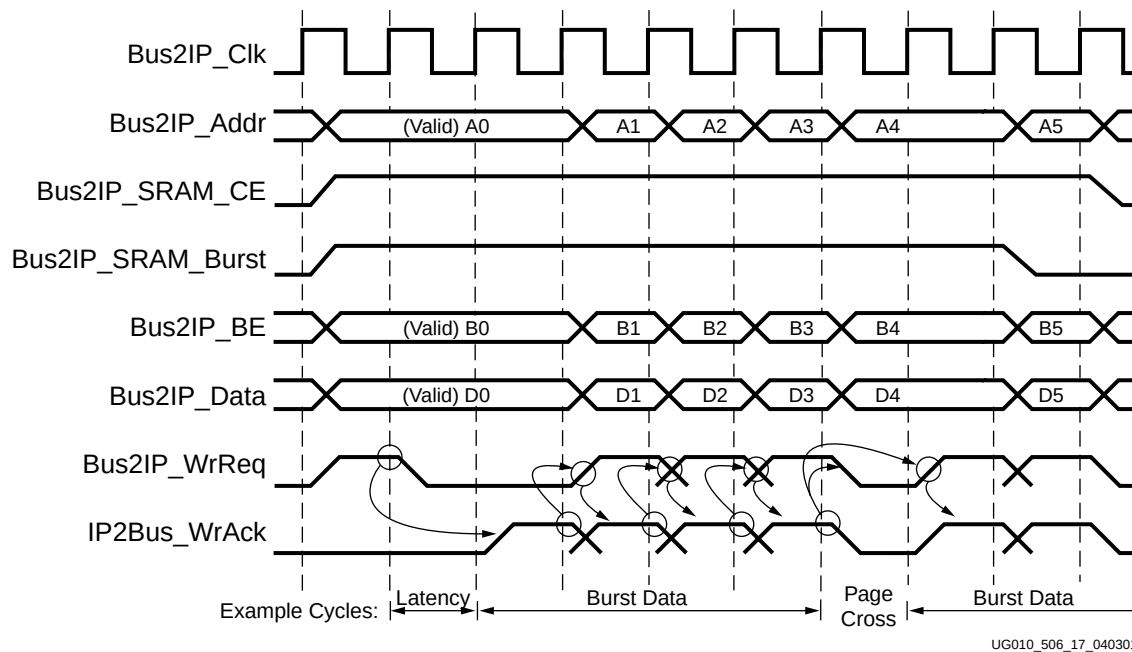


Figure 3-17: IPIF Slave SRAM Module Write Burst Transaction

## Write Burst Transactions

**Figure 3-17** illustrates the IPIF Slave SRAM module protocol for write burst cycles. Bursts from a bus master are treated as individual requests for service per datum. That is, the IPIF Slave SRAM module issues a set of requests acknowledges per datum. If the IP data bus is smaller, then the IPIF will sequence the data in the order it was received. Thus, a 32-bit bus transfer to an 8-bit IP is turned into four req/ack pairs.

Write bursts are designed to utilize maximum bandwidth across the IPIF. It is wisest to burst only to 32-bit IP devices, or else poor bus bandwidth will result. In the case of **Figure 3-17**, it is assumed that the IP is a 32-bit device. Note that the IPIF indicates an SRAM burst cycle by generating Bus2IP\_Addr, asserting Bus2IP\_SRAM\_CE, and issuing **Bus2IP\_SRAM\_Burst** and **Bus2IP\_WrReq** all in the same cycle. The IP then acknowledges the cycle by way of the **IP2Bus\_WrAck** signal after a 1-cycle latency. The IP continues to acknowledge the next three cycles. In the next cycle, the IP chooses to throttle the transaction by deasserting the **IP2Bus\_WrAck** signal. If the **IP2Bus\_WrAck** signal is deasserted for more than 16 clocks, the bus will time out, unless the IP also asserts the **IP2Bus\_ToutSup** signal, shown elsewhere. The IP eventually completes the data transfers, and the IPIF deasserts the **Bus2IP\_WrReq**. The IP will then deassert **IP2Bus\_WrAck**. Note that data is always transferred on the rising edge of the **Bus2IP\_Clock**, which sampled the **IP2Bus\_WrAck** as high.

Also note that **Bus2IP\_SRAM\_Burst** is deasserted after the second-to-last data is transferred. This allows the IP to know which datum is last in the burst. In effect, there is always only one more **IP2Bus\_WrAck** after **Bus2IP\_SRAM\_Burst** is deasserted.

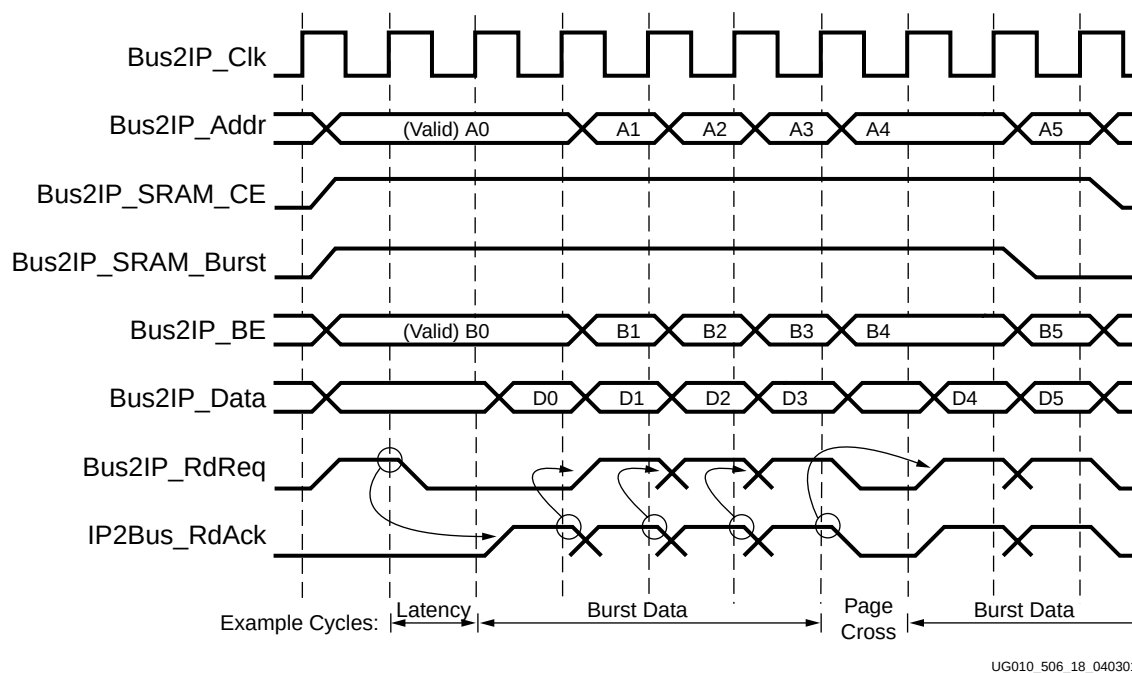


**Figure 3-18: IPIF Slave SRAM Module Read Burst Transaction**

## Read Burst Transactions

**Figure 3-18** illustrates the IPIF Slave SRAM module protocol for read burst cycles. Bursts to a bus master are treated as individual requests for service per datum. That is, the IPIF Slave SRAM module issues a set of request acknowledges per datum. If the IP data bus is smaller, then the IPIF sequences the data in the order it is received. Thus, a 32-bit bus transfer to an 8-bit IP gets turned into four req/ack pairs.

Read bursts are designed to utilize maximum bandwidth across the IPIF. It is wisest to only burst to 32-bit IP devices, or else poorer bus bandwidth will result. In the case of

**Figure 3-18**, it is assumed that the IP is a 32-bit device. Note that the IPIF indicates an SRAM burst cycle by generating **Bus2IP\_Addr**, asserting **Bus2IP\_SRAM\_CE** and issuing **Bus2IP\_SRAM\_Burst** and **Bus2IP\_RdReq** all in the same cycle. The IP then acknowledges the cycle by way of the **IP2Bus\_RdAck** signal after a 1-cycle latency. The IP continues to acknowledge the next three cycles. The IP throttles the transaction, for example, by deasserting the **IP2Bus\_RdAck** signal. If this signal is deasserted for more than 16 clocks, the bus will time out, unless the IP also asserts the **IP2Bus\_ToutSup** signal. The IP eventually completes the data transfers, the IPIF deasserts the **Bus2IP\_RdReq**, and the IP deasserts **IP2Bus\_RdAck**. Note that data is always transferred on the rising edge of the **Bus2IP\_Clk**, which sampled the **IP2Bus\_RdAck** as high.

Also note that **Bus2IP\_SRAM\_Burst** is deasserted after the second-to-last data is transferred. This allows the IP to know which datum is last in the burst. In effect, there is always only one more **IP2Bus\_WrAck** after **Bus2IP\_SRAM\_Burst** is deasserted.

### Other SRAM Module Uses

The IPIF Slave SRAM module can be used in many more forms than are illustrated in this specification. For example, it can be used as a simple external bus controller to talk to SRAM, FLASH, or external IP peripherals. Additionally, if a system is not performance intensive, it can be used to connect to all IP CPU peripherals. Since the address ranges of the IPIF module are adjustable, and multiple ranges are possible, this is easily achieved by adding an address decoder between the IPIF **Bus2IP\_Addr** lines (qualified by **Bus2IP\_SRAM\_CE**) and the IP peripherals.

## Slave SRAM Signal List

**Table 3-5** shows the name, direction, and a brief description of the signals that connect to the IP from the IPIF Slave Control Register module.

**Table 3-5: Signals for the IPIF Slave SRAM Module**

| Name                    | Direction | Description                                                                                             |
|-------------------------|-----------|---------------------------------------------------------------------------------------------------------|
| <b>Bus2IP_Clk</b>       | From IPIF | Clock source (from global buffer)                                                                       |
| <b>Bus2IP_Reset</b>     | From IPIF | Active-high synchronous reset source (from global buffer)                                               |
| <b>IP2Bus_Intr[0:i]</b> | To IPIF   | Interrupt input from IP to IPIF                                                                         |
| <b>IP2Bus_Error</b>     | To IPIF   | Error signal from IP to IPIF (Valid only during a data acknowledge cycle)                               |
| <b>IP2Bus_Retry</b>     | To IPIF   | Indicates IP wants master to retry the cycle                                                            |
| <b>IP2Bus_ToutSup</b>   | To IPIF   | Forces the suppression of watch dog timeout on the bus                                                  |
| <b>Bus2IP_Data[0:n]</b> | From IPIF | IPIF Write data (where n = IPIF_DATA_BUS_WIDTH - 1)                                                     |
| <b>IP2Bus_Data[0:n]</b> | To IPIF   | IPIF Read data (where n = IPIF_DATA_BUS_WIDTH - 1)                                                      |
| <b>Bus2IP_BE[0:b]</b>   | From IPIF | Byte enable, 1 = byte lane valid (where b = IPIF_NUMBER_OF_BYTE_ENABLES - 1)                            |
| <b>Bus2IP_WrReq</b>     | From IPIF | Write request from IPIF to IP, single clock high                                                        |
| <b>IP2Bus_WrAck</b>     | To IPIF   | Acknowledge that write data has been taken from <b>Bus2IP_Data[0:n]</b> , single <b>Bus2IP_Clk</b> high |
| <b>Bus2IP_RdReq</b>     | From IPIF | Read request from IPIF to IP, single clock high                                                         |

Table 3-5: Signals for the IPIF Slave SRAM Module (Continued)

| Name                      | Direction | Description                                                                                                                                                                        |
|---------------------------|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>IP2Bus_RdAck</b>       | To IPIF   | Acknowledge that read data has been placed on <b>IP2Bus_Data[0:n]</b> , single <b>Bus2IP_Clk</b> high                                                                              |
| <b>Bus2IP_Addr[al:ah]</b> | From IPIF | IPIF Slave SRAM address, where:<br>al = IPIF_SLV_SRAM_ADDR_BUS_MSB<br>ah = IPIF_SLV_SRAM_ADDR_BUS_LSB<br>( <i>al</i> is a lower number than <i>ah</i> due to big-endian numbering) |
| <b>Bus2IP_SRAM_CE</b>     | From IPIF | Clock enable of decoded SRAM address space, high for entire bus cycle                                                                                                              |

## Slave SRAM Parameters

Table 3-6 shows the parameters that can be selected for the IPIF Slave SRAM module..

Table 3-6: Parameters for the IPIF Slave SRAM Module

| Affects      | Parameter                                                                                                                                                                      | Value            | Type    |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|---------|
| General IPIF | <b>IPIF_DATA_BUS_WIDTH</b><br>Sets the size of the data bus for IPIF (where “n” in <b>Bus2IP_Data[0:n]</b> or <b>IP2Bus_Data[0:n]</b> is equal to <b>IPIF_DATA_BUS_WIDTH</b> ) | 8, 16, or 32     | number  |
|              | <b>IPIF_NUMBER_OF_BYTE_ENABLES</b><br>Sets the number of byte enables                                                                                                          | 1, 2, or 4       | number  |
|              | <b>IPIF_NUMBER_OF_INTR</b><br>Sets the number of interrupts the IP provides to the IPIF (where “i” in <b>IP2Bus_Intr[0:i]</b> is equal to <b>IPIF_NUMBER_OF_INTR</b> )         | 0 to 8           | number  |
|              | <b>IPIF_INTR_ID</b><br>Sets the unique interrupt ID for this IPIF                                                                                                              | 16 bits          | number  |
|              | <b>IPIF_HAS_INTC</b><br>Sets whether the IPIF has a built-in interrupt controller (See <a href="#">Figure 3-5</a> for more information)                                        | 0 or 1, 1 = true | boolean |
|              | <b>IP_HAS_OWN_INTC</b><br>Sets whether the IP has its own interrupt controller (See <a href="#">Figure 3-5</a> for more information)                                           | 0 or 1, 1 = true | boolean |



Table 3-6: Parameters for the IPIF Slave SRAM Module (Continued)

| Affects                                       | Parameter                                                                                                                                                                         | Value                             | Type   |
|-----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|--------|
| General IPIF Slave SRAM Module                | IPIF_NUMBER_OF_SRAM_DECODER_REGIONS<br>Defines the number of decoded regions of memory address space for the IPIF Slave SRAM module                                               | 1 to 4 decoded regions allowed    | number |
|                                               | IPIF_SLV_SRAM_ADDR_BUS_LSB<br>Sets the LSB index number for the address provided by the IPIF to the IP                                                                            | ordinal between 0 to 31           | number |
|                                               | IPIF_SLV_SRAM_ADDR_BUS_MSB<br>Sets the MSB index number for the address provided by the IPIF to the IP                                                                            | ordinal between 0 to 31           | number |
| IPIF Slave SRAM Module (Arrays of Parameters) | IPIF_SRAMd_BASE_ADDR<br>Sets the base address of the “d” region for the Slave SRAM module (where d = 0 to IPIF_NUMBER_SRAM_DECODER_REGIONS -1)                                    | 32-bit decode of one region       | number |
|                                               | IPIF_SRAMd_BASE_ADDR_BIT_ENBL[0:31]<br>Allows specification of which bits address bits to decode in the IPIF_SRAMd_BASE_ADDR (where d = 0 to IPIF_NUMBER_SRAM_DECODER_REGIONS -1) | 1 = decode respective address bit | mask   |

## Slave FIFO Module

This section covers the following topics:

- **Example Slave FIFO Application**
- **Generic Slave FIFO Model**
- **Slave FIFO Signal Protocol**
- **Slave FIFO Signal List**
- **Slave FIFO Parameters**

The IPIF Slave FIFO module is designed to interface between basic communications devices and a bus. The module looks substantially like a bidirectional FIFO. The IPIF Slave FIFO module provides a set of signals to the IP that permit the IP to access the module’s internal FIFOs. These FIFOs can be configured from one datum deep up to 2 KB deep.

### Example Slave FIFO Application

**Figure 3-19** illustrates a simplified example application of the IPIF Slave FIFO module, showing a vastly simplified communication IP with both transmit and receive functions. Each of the functions are connected to the IPIF by way of the IPIF Slave FIFO module.

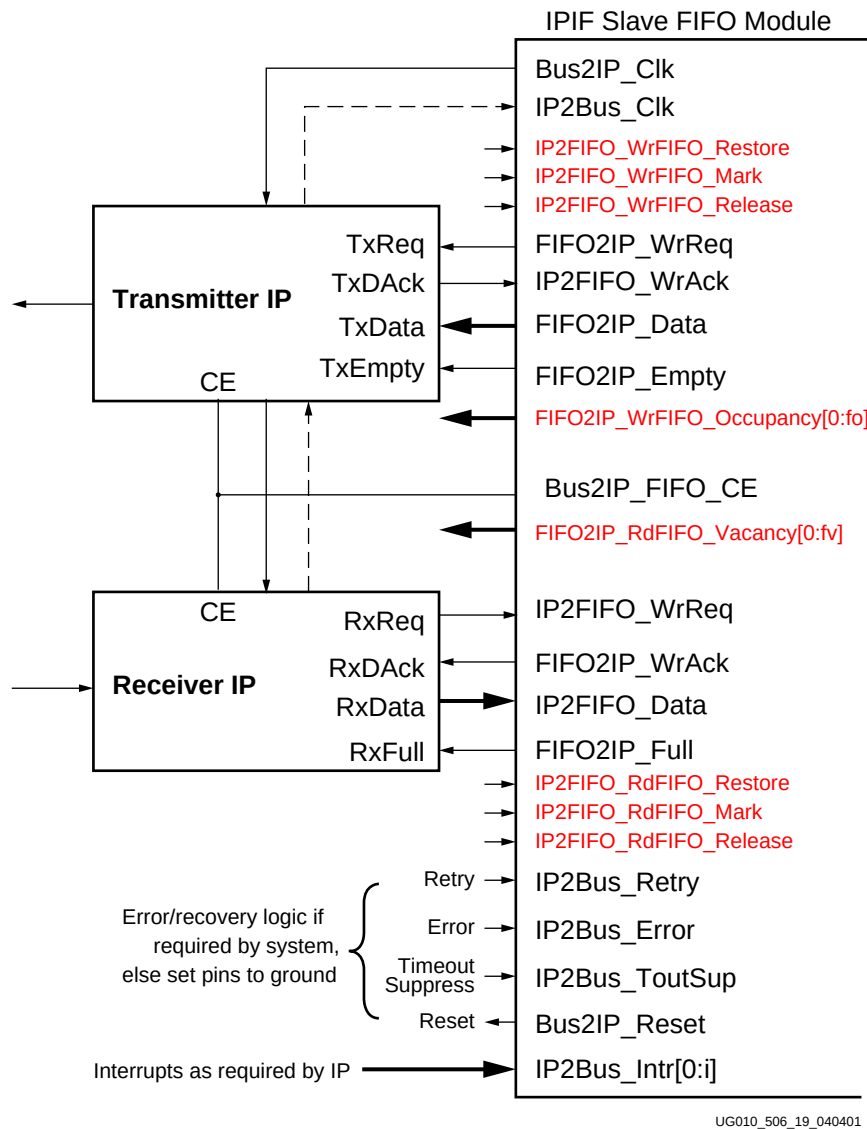


Figure 3-19: Example Application of IPIF Slave FIFO Module

The generic IP communication transmitter and receiver pair shown in Figure 3-19 are connected to the IPIF module's write FIFO and read FIFO respectively. The transmitter IP waits for data requests from the IPIF module. The receiver IP initiates requests for transfers when data is pending inside its receive buffer.

The transmitter IP in this example has signals such as:

- **TxReq**, a transmit request input
- **TxDack**, a transmit data acknowledge output
- **TxData**, a transmit data input
- **TxEmpty**, a transmit empty flag input (indicating the IPIF has no more data to transmit)

The receiver IP in this example has signals such as:

- **RxReq**, a receive request output
- **RxDack**, a receive data acknowledge input
- **RxData**, a receive data output
- **RxFull**, a receiver full flag input (indicating the IPIF can no longer accept data from

the receiver IP)

Both transmitter IP and receiver IP may also require a clock enable (CE) to indicate when cycles are for them rather than another IP block which might be connected.

The transmitter and/or receiver can be clocked from the **Bus2IP\_Clk** output of the IPIF Slave FIFO module. While this is recommended, the transmitter and/or receiver IPs can also generate a clock, **IP2FIFO\_Clk**, which is used to control the FIFO interfaces between the IPIF and the IP.

In **Figure 3-19**, when a bus master wishes to transmit data to the IP, it issues a write cycle to the IPIF Slave FIFO module. The address of this operation is settable by way of user parameters, and can be either a range of addresses spanning the depth of the FIFO, or can be a single “keyhole” address. If the IPIF Slave FIFO module detects an address hit on a write cycle, it can issue a request for service to the transmitter. The number of bus write cycles to the IPIF Slave FIFO module, before **FIFO2IP\_WrReq** goes valid, is parametrically settable at the time of instantiation of the IPIF module. (See **Table 3-8**.) Once the IPIF Slave FIFO contains a single datum, the **FIFO2IP\_Empty** flag is lowered to indicate that the IPIF has data in the write FIFO. When the IPIF Slave FIFO module has hit the write data threshold in its write FIFO, it will issue write requests until the transmitter IP causes the IPIF FIFO to go empty. Should the IPIF Slave FIFO module attempt to overflow the transmitter IP, the transmitter simply stops giving back **IP2FIFO\_WrAck** until the condition is cleared. If the transmitter is stalled so long that the internal FIFO in the IPIF fills, then bus retry is issued until the condition clears.

Additional signals are also illustrated in **Figure 3-19** which are available if the optional Packet mode of the Slave FIFO module is used. These signals are designed to add packet level awareness to the FIFO. For each FIFO, three basic signals are provided: mark, restore, and release. Vacancy or Occupancy level signals are also provided so that the Tx and Rx IP can know how much data is present in the FIFO of the IPIF.

The function of the mark, restore, and release pins is illustrated using the Write FIFO as context. The Read FIFO works identically. The purpose behind the **IP2FIFO\_WrFIFO\_Mark** signal is to mark the beginning of a packet of data. This mark causes the Slave FIFO module to memorize where in the FIFO the datum that was marked resides. This permits the IP to request a retransmit of the packet from the FIFO via the **IP2FIFO\_WrFIFO\_Restore** signal, should it be required. Once a packet has been known to be properly transmitted, or past a point where the data for the packet is committed, then the IP can issue the **IP2FIFO\_WrFIFO\_Release** signal to release the contents of the FIFO to accept new data. The overall effect of these signals is to permit smarter transmission of data between the bus and the IP. In addition to the mark, restore, and release signals, the Write FIFO also provides a set of signals which indicate how much data is present in the FIFO. This is done via the **FIFO2IP\_WrFIFO\_Occupancy[0:fo]** signals, where **fo** represents the index of bits required to contain the count of data. Similarly, the read FIFO has a set of vacancy signals, **FIFO2IP\_RdFIFO\_Vacancy[0:fv]**, where **fv** represents the index of bits required to contain the count of data.

These signals are not available at present for user designs, but may be made available in a future release of the IPIF specification. Accordingly, no timing diagrams are currently presented to illustrate this functionality.

## Generic Slave FIFO Model

Figure 3-20 illustrates the generic instance of the IPIF Slave FIFO module. The data bus widths, and number of interrupts for the IP have been generically specified.

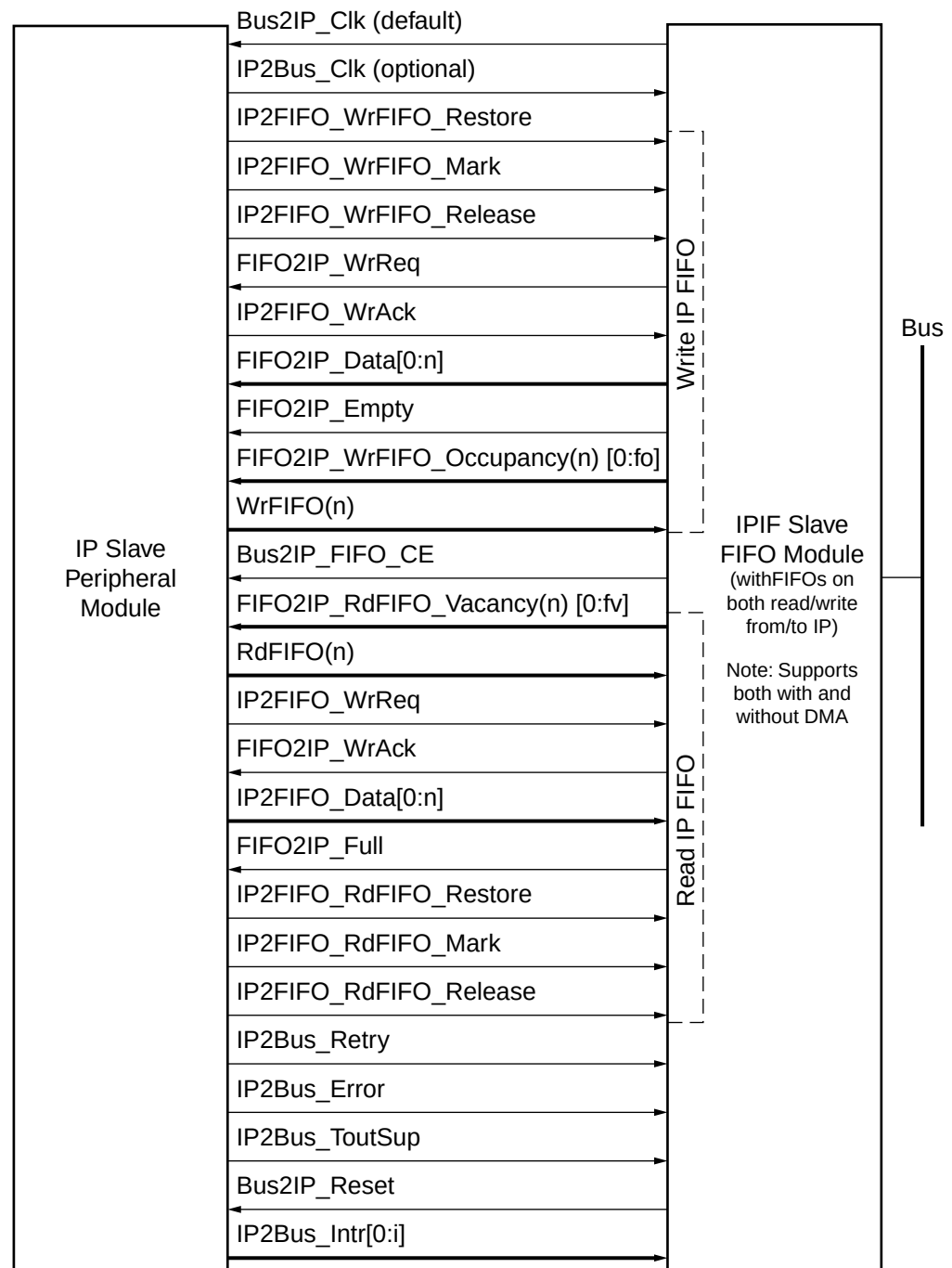


Figure 3-20: Generic Slave FIFO Module Block Diagram

The IPIF Slave FIFO module also provides a clock enable signal, **Bus2IP\_FIFO\_CE**, to indicate to other elements in the system that the FIFO is currently busy. **Bus2IP\_FIFO\_CE** is valid anytime that the IPIF module is busy transferring data on the bus, and may not reflect the current values of the write or read FIFOs. Typically, the clock enable is necessary only when using additional IPIF modules.

## Slave FIFO Signal Protocol

Figure 3-21 shows a protocol diagram of the IPIF Slave FIFO module, illustrating both reads and writes.

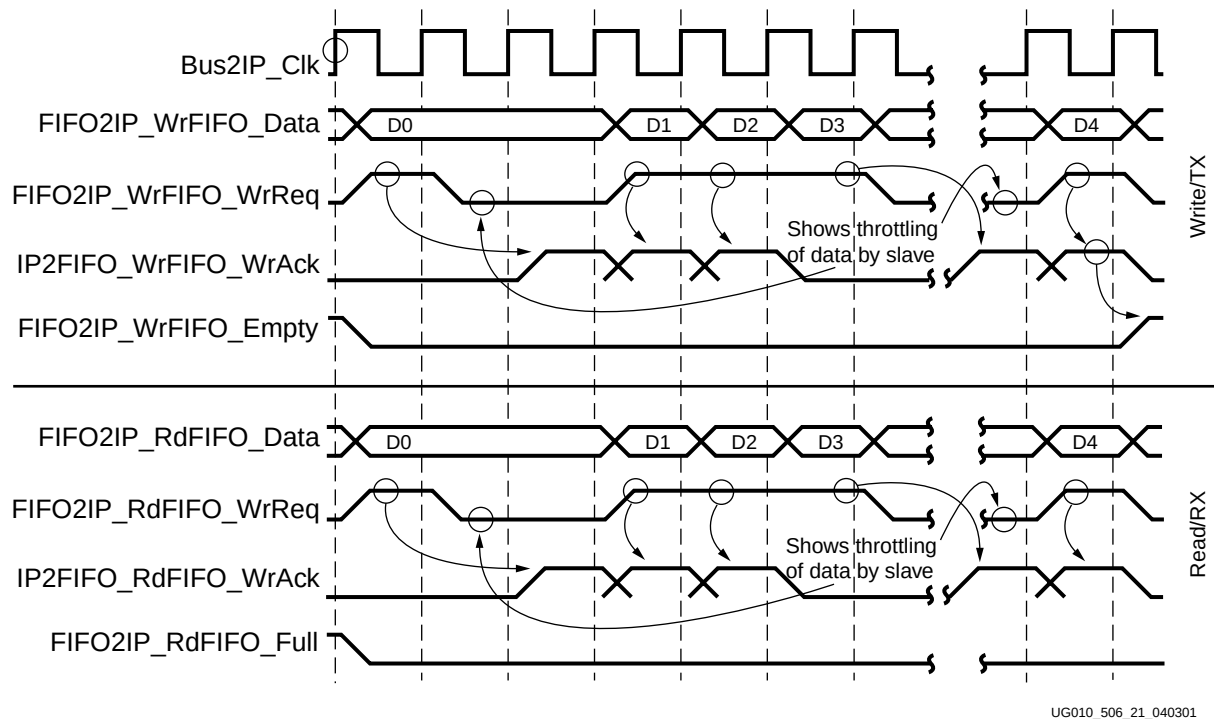


Figure 3-21: Timing Diagram for Slave IP FIFO Module Read and Write Transactions

### Write Transactions

The write FIFO in the IPIF Slave FIFO module works as the transmit FIFO.

In addition to the write FIFO data, the write FIFO example in Figure 3-21 has the following signals:

- **FIFO2IP\_WrFIFO\_WrReq**, a write request output (requesting the IPIF to write data to the IP)
- **IP2FIFO\_WrFIFO\_WrAck**, a write acknowledge input (from the IP to the IPIF, indicating it has taken the FIFO2IP\_Data)
- **FIFO2IP\_WrFIFO\_Empty**, a write FIFO empty output (indicates to the IP that the IPIF has no more data)

Figure 3-21 (top half) illustrates the IPIF Slave FIFO module write protocol. In this diagram, the IP is first notified that there is data in the IPIF's write FIFO by the deassertion of the **FIFO2IP\_WrFIFO\_Empty** signal. When it transitions from high to low, it indicates that the write FIFO has data in it. Note that the **FIFO2IP\_WrFIFO\_Data** bus is also now valid, and remains so until the cycle that acknowledges the data. The IPIF Slave FIFO module generates a request to the IP to take data from the IPIF by asserting the **FIFO2IP\_WrFIFO\_WrReq** signal. The IP can acknowledge immediately, or delay as required. The signal for acknowledging the cycle is the **IP2FIFO\_WrFIFO\_WrAck**. This permits the IP to throttle the data from the IPIF as required. The end of the diagram illustrates emptying the FIFO again.

### Read Transactions

The read FIFO of the IPIF Slave FIFO module works as the receive FIFO. In order to allow communications between the IP and the IPIF, four basic signals are provided.

In addition to the read FIFO data, the read FIFO example in [Figure 3-21](#) has the following signals:

- **IP2FIFO\_RdFIFO\_WrReq**, a write request input (indicating a request by the IP to write data to the IPIF)
- **FIFO2IP\_RdFIFO\_WrAck**, a write acknowledge output (from the IPIF to the IP, indicating it has taken the **IP2FIFO\_Data**)
- **FIFO2IP\_RdFIFO\_Full**, a read FIFO full output (indicating to the IP that the IPIF is currently full and no other requests will be acknowledged until the FIFO empties)

[Figure 3-21](#) (bottom half) illustrates the IPIF Slave FIFO module read protocol. In this diagram, the IP has issued a request to write data to the IPIF FIFO via the **IP2FIFO\_RdFIFO\_WrReq** signal. Note that in this case, the IPIF read FIFO was full, indicated by **FIFO2IP\_RdFIFO\_Full** signal. The IP must not request another data transfer until the full condition has been removed. Since the IPIF Slave FIFO module is busy transferring data on the bus, it will eventually reach a condition where it is no longer full. Once this occurs, the IPIF permits more data transfers. It holds the **FIFO2IP\_RdFIFO\_WrAck** signal low until the same cycle that the **FIFO2IP\_RdFIFO\_Full** signal will go low. Then it permits the **FIFO2IP\_RdFIFO\_Full** to transition to low, and the **FIFO2IP\_RdFIFO\_WrAck** to transition high to acknowledge the requested data transfer. The data transfers can continue for as long as the IPIF read FIFO is not full. The IP can throttle the transfers by not issuing **IP2FIFO\_RdFIFO\_WrReq**.

### Slave FIFO Signal List

[Table 3-7](#) shows the name, direction, and a brief description of the signals which connect to the IP from the IPIF Slave FIFO module.

**Table 3-7: Signals for the IPIF Slave FIFO Module**

| Name                            | Direction | Description                                                                                  |
|---------------------------------|-----------|----------------------------------------------------------------------------------------------|
| <b>Bus2IP_Clk</b>               | From IPIF | Clock source (from global buffer)                                                            |
| <b>Bus2IP_Reset</b>             | From IPIF | Active-high synchronous reset source (from global buffer)                                    |
| <b>IP2Bus_Intr[0:i]</b>         | To IPIF   | Interrupt input from IP to IPIF                                                              |
| <b>IP2Bus_Error</b>             | To IPIF   | Error signal from IP to IPIF (Valid only during a data acknowledge cycle)                    |
| <b>IP2Bus_Retry</b>             | To IPIF   | Indicates IP wants master to retry the cycle                                                 |
| <b>IP2Bus_ToutSup</b>           | To IPIF   | Forces the suppression of watch dog timeout on the bus                                       |
| <b>IP2Bus_Clk (optional)</b>    | To IPIF   | Optional clock source to control IPIF Slave FIFO module signals (not yet available)          |
| <b>Bus2IP_FIFO_CE</b>           | From IPIF | Clock enable of decoded IPIF Slave FIFO address space, high for entire bus cycle             |
| <b>FIFO2IP_WrFIFO_Data[0:n]</b> | From IPIF | IPIF write FIFO data (where n = IPIF_WR_FIFO_DATA_WIDTH - 1)                                 |
| <b>FIFO2IP_WrFIFO_Empty</b>     | From IPIF | IPIF write FIFO is empty when high                                                           |
| <b>FIFO2IP_WrFIFO_WrReq</b>     | From IPIF | IPIF write FIFO request, single <b>Bus2IP_Clk</b> (or <b>IP2Bus_Clk</b> ) high per datum     |
| <b>IP2FIFO_WrFIFO_WrAck</b>     | To IPIF   | IPIF write FIFO acknowledge, single <b>Bus2IP_Clk</b> (or <b>IP2Bus_Clk</b> ) high per datum |
| <b>IP2FIFO_RdFIFO_Data[0:n]</b> | To IPIF   | IPIF read FIFO data (where n = IPIF_RD_FIFO_DATA_WIDTH - 1)                                  |

Table 3-7: Signals for the IPIF Slave FIFO Module (Continued)

| Name                           | Direction | Description                                                                                                                                         |
|--------------------------------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| FIFO2IP_RdFIFO_Full            | From IPIF | IPIF write FIFO is empty when this signal is high                                                                                                   |
| FIFO2IP_RdFIFO_WrReq           | To IPIF   | IPIF read FIFO request, single <b>Bus2IP_Clk</b> (or <b>IP2Bus_Clk</b> ) high per datum                                                             |
| IP2FIFO_RdFIFO_WrAck           | From IPIF | IPIF read FIFO acknowledge, single <b>Bus2IP_Clk</b> (or <b>IP2Bus_Clk</b> ) high per datum                                                         |
| IP2FIFO_WrFIFO_Mark            | To IPIF   | Marks datum as first datum of packet, high during <b>IP2FIFO_WrFIFO_WrAck</b> only                                                                  |
| IP2FIFO_WrFIFO_Restore         | To IPIF   | Restores write FIFO to marked location, next request will be at marked data (this signal is valid high only during an <b>IP2FIFO_WrFIFO_WrAck</b> ) |
| IP2FIFO_WrFIFO_Release         | To IPIF   | Releases the marked position, write FIFO can now accumulate data from mark forward                                                                  |
| FIFO2IP_WrFIFO_Occupancy[0:fo] | From IPIF | Indicates how much data is in the write FIFO (where $fo = \text{IPIF\_WR\_FIFO\_NUMBER\_OF\_OCC\_BITS} - 1$ )                                       |
| IP2FIFO_RdFIFO_Mark            | To IPIF   | Marks datum as first datum of packet, high during <b>IP2FIFO_RdFIFO_WrAck</b> only                                                                  |
| IP2FIFO_RdFIFO_Restore         | To IPIF   | Restores write FIFO to marked location, next request will be at marked data (this signal is valid high only during an <b>IP2FIFO_RdFIFO_WrAck</b> ) |
| IP2FIFO_RdFIFO_Release         | To IPIF   | Releases the marked position, read FIFO can now accumulate data from mark forward                                                                   |
| FIFO2IP_WrFIFO_Vacancy[0:fv]   | From IPIF | Indicates how much data is in the read FIFO (where $fv = \text{IPIF\_WR\_FIFO\_NUMBER\_OF\_VAC\_BITS} - 1$ )                                        |

## Slave FIFO Parameters

**Table 3-8** shows the parameters that can be selected for the IPIF Slave FIFO module.

**Table 3-8: Parameters for the IPIF Slave FIFO Module**

| Affects      | Parameter                                                                                                                                                                      | Value            | Type    |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|---------|
| General IPIF | <b>IPIF_DATA_BUS_WIDTH</b><br>Sets the size of the data bus for IPIF (where “n” in <b>Bus2IP_Data[0:n]</b> or <b>IP2Bus_Data[0:n]</b> is equal to <b>IPIF_DATA_BUS_WIDTH</b> ) | 8, 16, or 32     | number  |
|              | <b>IPIF_NUMBER_OF_BYTE_ENABLES</b><br>Sets the number of byte enables                                                                                                          | 1, 2, or 4       | number  |
|              | <b>IPIF_NUMBER_OF_INTR</b><br>Sets the number of interrupts the IP provides to the IPIF (where “i” in <b>IP2Bus_Intr[0:i]</b> is equal to <b>IPIF_NUMBER_OF_INTR</b> )         | 0 to 8           | number  |
|              | <b>IPIF_INTR_ID</b><br>Sets the unique interrupt ID for this IPIF                                                                                                              | 16 bits          | number  |
|              | <b>IPIF_HAS_INTC</b><br>Sets whether the IPIF has a built-in interrupt controller (See <b>Figure 3-5</b> for more information)                                                 | 0 or 1, 1 = true | boolean |
|              | <b>IP_HAS_OWN_INTC</b><br>Sets whether the IP has its own interrupt controller (See <b>Figure 3-5</b> for more information)                                                    | 0 or 1, 1 = true | boolean |



Table 3-8: Parameters for the IPIF Slave FIFO Module (Continued)

| Affects                        | Parameter                                                                                                             | Value                             | Type    |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------|-----------------------------------|---------|
| General IPIF Slave FIFO Module | IPIF_RD_FIFO_DEPTH<br>Sets the depth of the Read FIFO in IPIF_RD_FIFO_DATA_WIDTH units                                | 1 to 2048, in units of data width | number  |
|                                | IPIF_RD_FIFO_DATA_WIDTH<br>Sets the width of the Read FIFO data bus                                                   | 1 to 32 bits                      | number  |
|                                | IPIF_RD_FIFO_TRIG_THRESHOLD<br>Sets the trigger point of the Read FIFO if packet processing                           | 1 to 2048                         | number  |
|                                | IPIF_WR_FIFO_DEPTH<br>Sets the depth of the Write FIFO in IPIF_WR_FIFO_DATA_WIDTH units                               | 1 to 2048, in units of data width | number  |
|                                | IPIF_WR_FIFO_DATA_WIDTH<br>Sets the width of the Write FIFO data bus                                                  | 1 to 32 bits                      | number  |
|                                | IPIF_WR_FIFO_TRIG_THRESHOLD<br>Sets the trigger point of the Write FIFO if packet processing                          | 1 to 2048                         | number  |
|                                | IPIF_PACKET_FIFO_MODE_ENBL<br>Enables the IPIF Slave FIFO module with packet FIFO functionality                       | 0 or 1, 1 = true                  | boolean |
|                                | IPIF_WR_FIFO_NUMBER_OF_OCC_BITS<br>Sets the total number of bits in the <b>FIFO2IP_WrFIFO_Occupancy[0:fo]</b> signals | 0 TO 12                           | number  |
|                                | IPIF_WR_FIFO_NUMBER_OF_VAC_BITS<br>Sets the total number of bits in the <b>FIFO2IP_WrFIFO_Vacancy[0:fv]</b> signals   | 0 TO 12                           | number  |

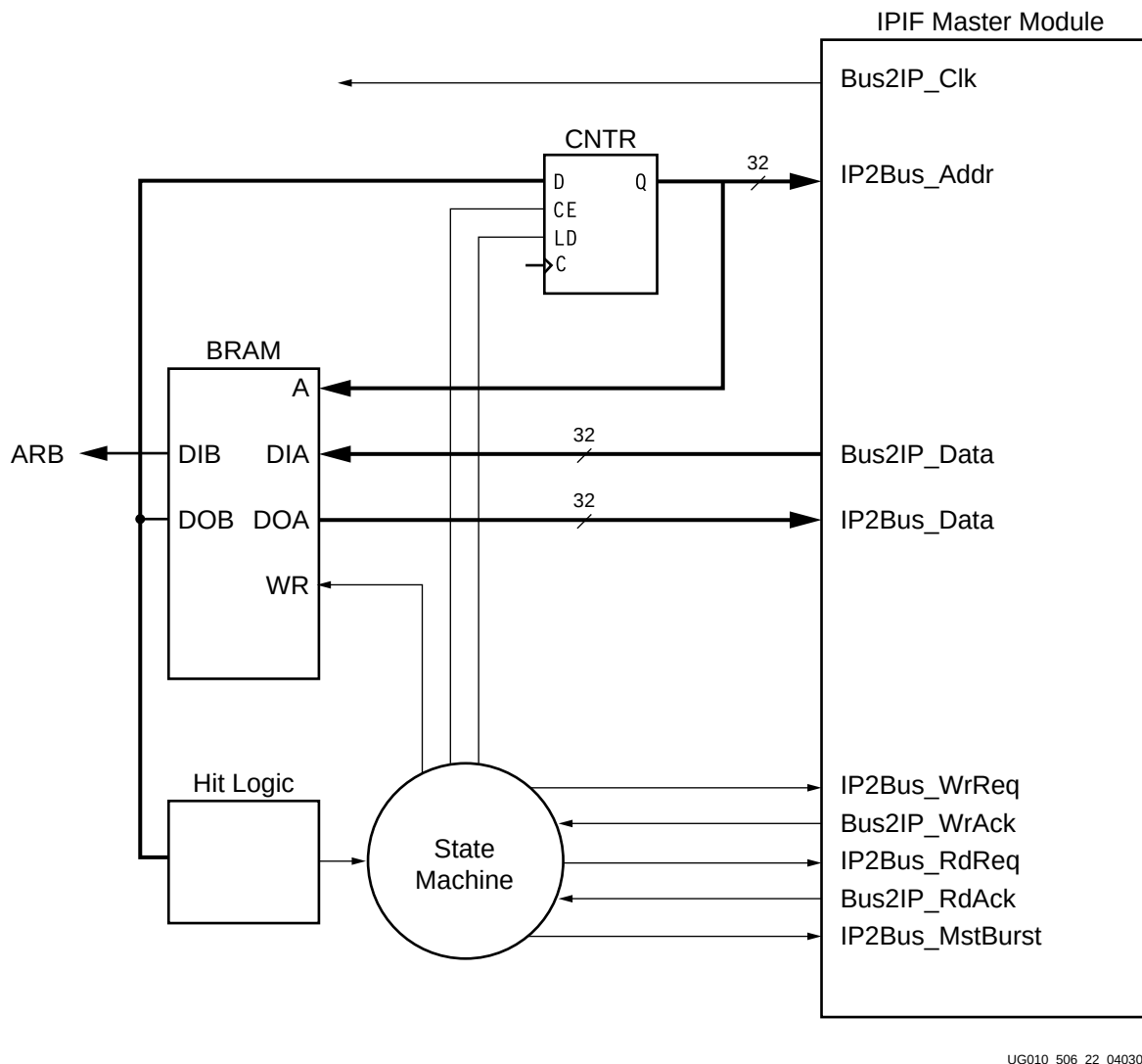
## Master Module

- **Example Master Application**
- **Generic Master Model**
- **Master Signal Protocol**
- **Master Signal List**
- **Master Parameters**

### Example Master Application

Figure 3-22 is a simplified example of an IPIF Master module application and is intended for illustrative and informative purposes only. This example illustrates a basic state

machine that monitors a block RAM (BRAM) for a semaphore, and based upon receiving the semaphore initiates a dump from the BRAM across the bus to a slave device.



UG010\_506\_22\_040301

Figure 3-22: Example Application of IPIF Master Module

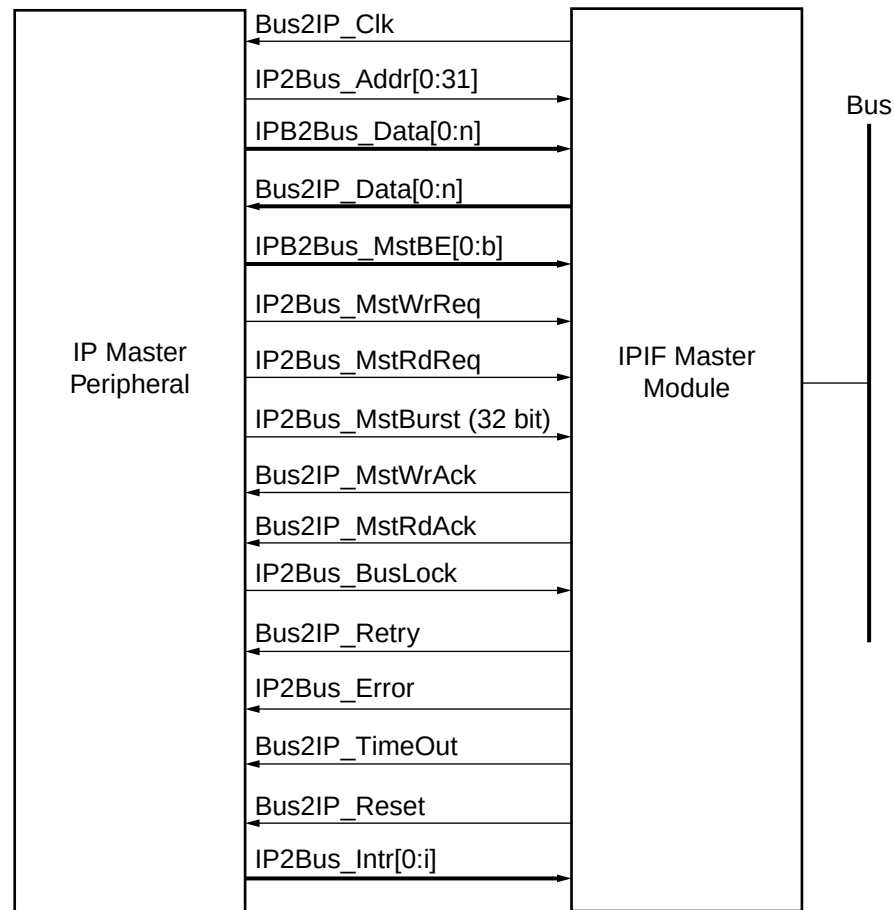
In **Figure 3-22**, a state machine watches for a semaphore, and then initiates a set of burst cycles across the bus as required to dump the data frame. The **IP2Bus\_Addr** signal is generated by a simple 32-bit counter controlled by the state machine. The **IP2Bus\_MstWrReq** and **IP2Bus\_MstRdReq** signals are also controlled by the state machine. When the state machine is told that the appropriate semaphore is found, it initiates a read or write cycle. The **Bus2IP\_MstWrAck** and **Bus2IP\_MstRdAck** signals are also brought into the state machine in order to properly sequence the address from the counter and the data to and from the BRAM. The dual port feature of the BRAM allows independence between the IPIF and the rest of the IP.

The address counter generates addresses for the BRAM using the lower-ordered bits. This allows the IPIF to have a range of memory on the bus that coincides with the BRAM. The address counter also provides the address for the IPIF Master module.

The hit logic is connected to the B port of the BRAM and the IPIF Master module is connected to the A port. It is possible to use the BRAM to cross clock boundaries with other elements of the IP. This is a useful technique for dealing with the often asynchronous nature of communications systems.

## Generic Master Model

Figure 3-23 illustrates a general model of the IPIF Master module. This module is used to initiate transactions across the IPIF and onto the bus and is only required for those IPs that need to initiate bus cycles.



UG010\_506\_23\_040301

Figure 3-23: Master IP Protocol Block Diagram

The IPIF Master module provides a clock output, **Bus2IP\_Clk**, which allows the IP to be clocked synchronously by the IPIF. This clock is provided via a BUFG in the FPGA, and thus no extra clock buffering is required in the IP.

### Initiating a Cycle

In order to initiate a cycle on the bus, the IP must provide a full 32-bit address to the IPIF module by way of **IP2Bus\_Addr[0:31]**. The addresses that are valid during IP to IPIF cycles are placed on the bus at the appropriate time by the IPIF. The lower two address bits, **IP2Bus\_Addr[30:31]**, can be tied to logic 0 if the IP always performs 32-bit transfers across the IPIF.

Figure 3-23 shows the data buses for the IP. The **IP2Bus\_Data** bus is the write data bus from the IP; that is, when the IP wishes to write data to the IPIF Master module, it asserts its write data on the **IP2Bus\_Data** lines. Similarly, when the IP wishes to read data from the IPIF Master module, it looks for read data on the **Bus2IP\_Data** lines.

The IPIF Master module uses byte enables to select the size of the data transferred during the initiated bus cycle. For 32-bit IP masters, the IPIF requires four byte enables, **IP2Bus\_MstBE[0:3]**. Each byte enable corresponds to the byte lane in which to enable

transfers. Accordingly, byte, half word, and word transfers can be accommodated by asserting the correct **IP2Bus\_MstBE[0:3]**. The **IP2Bus\_MstBE[0]** corresponds to the byte lane contained in **IP2Bus\_Data[0:7]** or **Bus2IP\_Data[0:7]** when in IBM-endianess mode.

### Requesting Service

In order to initiate the request for service, the IP must assert the **IP2Bus\_Addr**, the **IP2Bus\_MstBE**, **IP2Bus\_Data** (if required), and issue an **IP2Bus\_MstWrReq** or **IP2Bus\_MstRdReq**. Asserting the **IP2Bus\_MstWrReq**, along with the proper qualifiers, initiates a master write transaction across the bus to the address pointed at by the **IP2Bus\_Addr**. Asserting the **IP2Bus\_MstRdReq**, along with the proper qualifiers, initiates a master read transaction across the bus to the address pointed at by the **IP2Bus\_Addr**.

IP masters that are 32 bits can also issue an **IP2Bus\_MstBurst** signal along with the **IP2Bus\_MstWrReq** or **IP2Bus\_MstRdReq** signals, indicating that the IP wishes to place a set of contiguous 32-bit transfers on the bus. The address of the transfer must always be word aligned (e.g. address bits 30, 31 are both low). For best bus performance across the IPIF, the master must only perform cacheline-aligned transfers using the **IP2Bus\_MstBurst** signal.

When the IPIF receives an acknowledge from the bus slave it is talking to, it issues either the **Bus2IP\_MstWrAck** or **Bus2IP\_MstRdAck** according to the cycle type initiated by the IP. In the case of read cycles, the assertion of **Bus2IP\_MstRdAck** indicates that valid data will be removed from the **Bus2IP\_Data** bus.

### Asserting Bus Locking

The IP may also choose to lock multiple transactions together via the **IP2Bus\_MstBusLock** signal. This asserts the Bus Locking mechanism on the bus and prevents any other Masters from gaining access to the bus until the **IP2Bus\_MstBusLock** signal is deasserted. A substantial loss in performance may result if the bus locking mechanism is used too frequently. In general, only use bus lock if a set of transactions must remain atomic across the bus.

### Completing a Cycle

The IPIF provides error feedback to the IP master about how the IP-initiated cycle completed. If the cycle is completed normally, then none of the error feedback signals will be true. However, the IPIF provides the **Bus2IP\_MstRetry** signal to tell the IPIF Master module it must back off the bus and restart the transaction. Additionally, the IPIF also provides **Bus2IP\_MstError** to indicate that the bus cycle(s) ended in an error condition. Finally, the IPIF provides the **Bus2IP\_MstTimeout** signal to indicate that the bus cycle timed out. Timeouts are usually caused by an invalid address.

## Master Signal Protocol

Figure 3-24 and Figure 3-25 illustrate single write and read cycles of an IP bus cycle initiation. Each figure shows two possible access times in response to the IP, short and long.

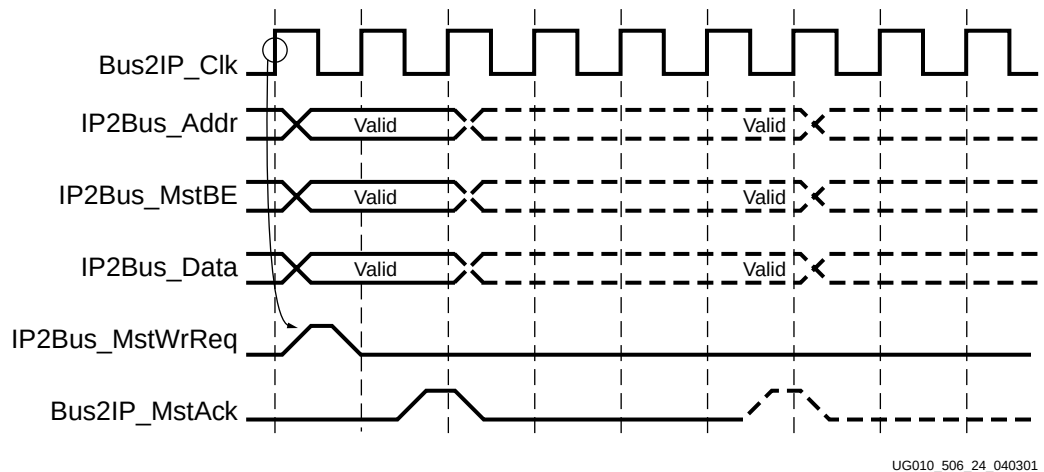


Figure 3-24: Timing Diagram for the Master IP Module Protocol Single Write Transaction

### Write Transactions

The IP initiates a single write cycle by presenting the **IP2Bus\_MstWrReq**, along with all the transaction qualifiers (including **IP2Bus\_MstAddr**, **IP2Bus\_MstBE**, and **IP2Bus\_Data**) to the IPIF master. When the IPIF master receives a write request, it initiates a write transaction. The transaction qualifiers are also used to correctly construct the master bus cycle. In general, if the bus is not busy, then the IPIF will quickly acknowledge the IP's initiated bus cycle. If the bus is busy, and the IPIF Master module has to wait for another master before it gets the bus, then the IPIF may take longer to acknowledge the cycle.

Figure 3-24 illustrates both *not busy* and *busy* cases. In the first case (left of diagram), the IP has requested the bus write data to the bus slave, specified by the address provided by the IP. In this case, the bus answers quickly, allowing the IPIF to quickly acknowledge the cycle. In contrast, the right side of the diagram indicates a case where the IPIF Master module has to wait for another master to get off the bus, and then arbitrate for the bus. In this case, the bus takes much longer to acknowledge the cycle back to the IPIF, therefore delaying the IPIF's acknowledge to the IP.

**NOTE:** In this example, the bus transactions all completed normally. In the case of an error, retry, or timeout, the IPIF Master module generates the **Bus2IP\_Error**, **Bus2IP\_Retry**, and/or **Bus2IP\_Timeout** during the **Bus2IP\_MstWrAck** signal. It is the IP's responsibility to correctly address the problem.

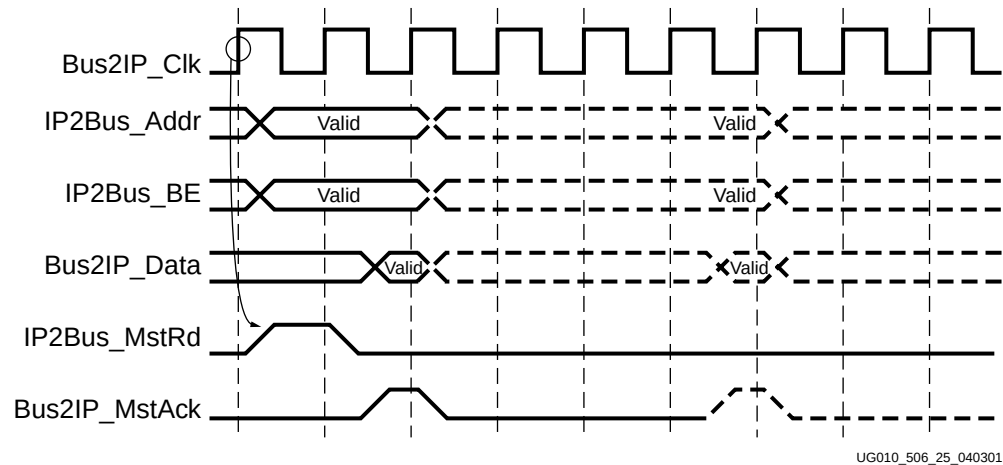


Figure 3-25: Timing Diagram for the Master IP Module Protocol Single Read Transaction

### Read Transactions

The IP initiates a single read cycle by presenting the **IP2Bus\_MstRdReq**, along with all the transaction qualifiers including **IP2Bus\_MstAddr** and **IP2Bus\_MstBE**. When the IPIF master receives a read request, it initiates a bus read transaction. The transaction qualifiers are also used to correctly construct the master bus cycle. In general, if the bus is not busy, the IPIF will quickly acknowledge the IP's initiated bus cycle. If the bus is busy, and the IPIF master module has to wait for another master before it gets the bus, then the IPIF may take longer to acknowledge the cycle.

**Figure 3-25** illustrates both *not busy* and *busy* cases. In the first case (left of diagram), the IP has requested the bus read data to the bus slave specified by the address provided by the IP. In this case, the bus answers quickly, allowing the IPIF to quickly acknowledge the cycle. In contrast, the right side of the diagram indicates a case where the IPIF Master module has to wait for another master to get off the bus, and then arbitrate for the bus. In this case, the bus takes much longer to acknowledge the cycle back to the IPIF, therefore delaying the IPIF's acknowledge to the IP.

**NOTE:** In this example, the bus transactions all completed normally. In the case of an error, retry, or timeout, the IPIF Master module generates the **Bus2IP\_Error**, **Bus2IP\_Retry**, and/or **Bus2IP\_Timeout** during the **Bus2IP\_MstWrAck** signal. It is the IP's responsibility to correctly address the problem.

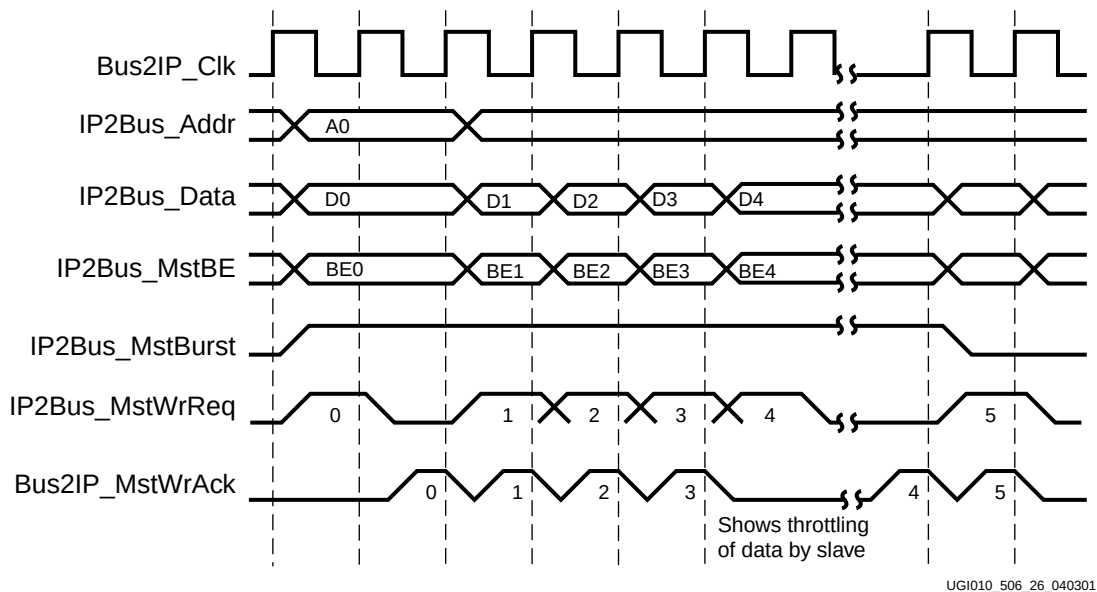


Figure 3-26: Timing Diagram for the Master IP Module Protocol Burst Write Transaction

### Burst Write Transactions

Figure 3-26 illustrates the burst write cycle case of an IP bus cycle initiation. This diagram indicates a burst write transfer. There is no real limit on the number of datums that can be transferred, other than the attendant needs of other masters on the bus or the requirements of the memory subsystem.

The IP initiates a burst write cycle by presenting the **IP2Bus\_MstWrReq**, along with all the transaction qualifiers including **IP2Bus\_MstAddr**, **IP2Bus\_MstBE**, **IP2Bus\_MstBurst**, and **IP2Bus\_Data**. When the IPIF master receives a burst write request, it initiates a bus burst write transaction. The transaction qualifiers are also used to correctly construct the Master bus cycle. In general, if the bus is not busy, the IPIF will quickly acknowledge the IP's initiated bus cycle. If the bus is busy, and the IPIF Master module has to wait for another master before it gets the bus, then the IPIF may take longer to acknowledge the cycle.

**NOTE:** The first write datum is held until acknowledged by **Bus2IP\_MstWrAck**. Datums are then sent per every active **Bus2IP\_MstWrAck**. Since the bus slave on the other end of the transaction can throttle the data rate, it is possible that gaps may exist such as that shown between cycle 3 and cycle K-1.

There is no limit to the number of burst cycles that can be accomplished other than the practical utilization of the bus. It is recommended that only cache-aligned bursts be done in order to maximize system performance.

The IP deasserts the **IP2Bus\_MstBurst** signal when the second to last datum is transferred. This is required to inform the IPIF logic that the next datum is the last datum of the transfer, and to get off the bus efficiently. Figure 3-26 indicates no gap in the **Bus2IP\_MstWrAcks** for the last two cycles. However, in some cases the datums may be separated by several clocks. In this case, the **IP2Bus\_MstBurst** signal must be terminated in the same cycle that a the second to last datum is acknowledged.

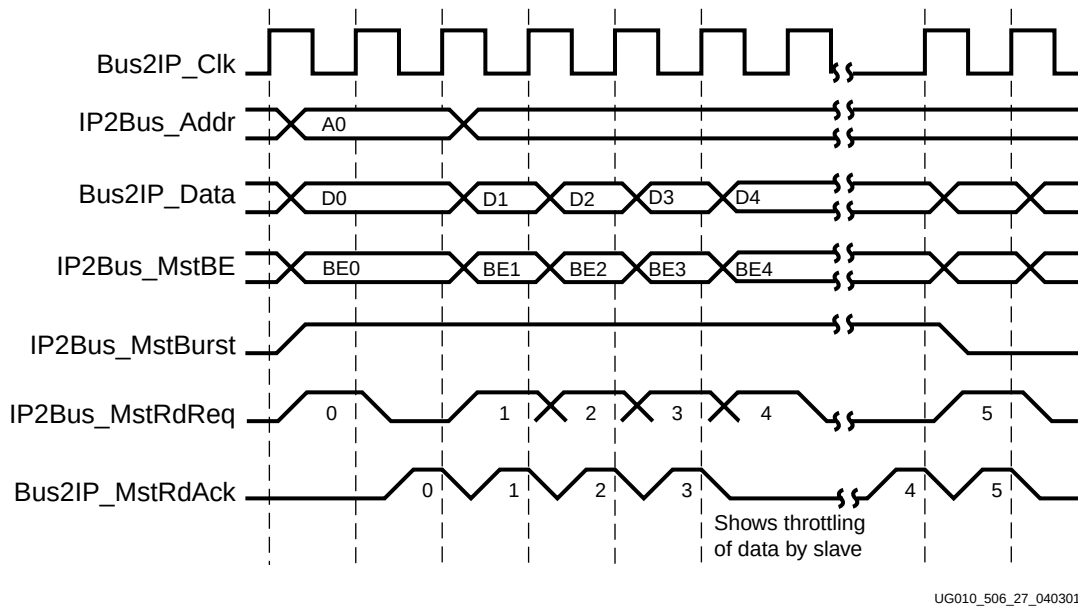


**Figure 3-27: Timing Diagram for the Master IP Module Protocol Burst Read Transaction**

### Burst Read Transactions

**Figure 3-27** illustrates the burst read cycle case of an IP bus cycle initiation. This diagram indicates a burst read transfer. There is no real limit on the number of datums that can be transferred, other than the attendant needs of other masters on the bus or the requirements of the memory subsystem.

The IP initiates a burst read cycle by presenting the **IP2Bus\_MstRdReq**, along with all the transaction qualifiers including **IP2Bus\_MstAddr**, **IP2Bus\_MstBE**, and **IP2Bus\_MstBurst**. When the IPIF master receives a burst read request, it initiates a bus burst read transaction. The transaction qualifiers are also used to correctly construct the bus Master bus cycle. In general, if the bus is not busy, the IPIF will quickly acknowledge the IP's initiated bus cycle. If the bus is busy, and the IPIF Master module has to wait for another master before it gets the bus, then the IPIF may take longer to acknowledge the cycle.

**NOTE:** The first read datum is not available until acknowledged by **Bus2IP\_MstRdAck**.

Datums are then sent per every active **Bus2IP\_MstRdAck**. Since the bus slave on the other end of the transaction can throttle the data rate, it is possible that gaps may exist such as that shown between cycle 3 and cycle K-1.

There is no limit to the number of burst cycles that can be accomplished other than the practical utilization of the bus. It is recommended that only cache-aligned bursts be done in order to maximize system performance.

The IP deasserts the **IP2Bus\_MstBurst** signal when the second to last datum is transferred. This is required to inform the IPIF logic that the next datum is the last datum of the transfer, and to get off the bus efficiently. **Figure 3-27** indicates no gap in the **Bus2IP\_MstWrAcks** for the last two cycles. However, in some cases the datums may be separated by several clocks. In this case, the **IP2Bus\_MstBurst** signal must be terminated in the same cycle that the second to last datum is acknowledged.

### General Burst Cycle Issues

When issuing burst cycle, the IPIF Master module must be capable of providing or receiving a datum per **Bus2IP\_Clk**. The bus permits slaves to respond each cycle of a burst operation, so there is no means to tell the slave to hold off. Only slaves can throttle transactions.



## Master Signal List

**Table 3-9** shows the names, direction, and a brief description of the signals that connect to the IP from the IPIF Master module.

**Table 3-9: Signals for the IPIF Master Module**

| Name                     | Direction | Description                                                                                                                                                                              |
|--------------------------|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Bus2IP_Clk</b>        | From IPIF | Clock source (from global buffer)                                                                                                                                                        |
| <b>Bus2IP_Reset</b>      | From IPIF | Active-high synchronous reset source (from global buffer)                                                                                                                                |
| <b>IP2Bus_Intr[0:i]</b>  | To IPIF   | Interrupt input from IP to IPIF                                                                                                                                                          |
| <b>IP2Bus_Addr[0:31]</b> | To IPIF   | 32-bit address of location Master wants to read or write                                                                                                                                 |
| <b>IP2Bus_MstBE[0:b]</b> | To IPIF   | Master byte enable, 1 = byte late valid<br>(where b = IPIF_NUMBER_OF_BYTE_ENABLES -1)                                                                                                    |
| <b>IP2Bus_MstWrReq</b>   | To IPIF   | Master requests write access to the bus, single <b>Bus2IP_Clk</b> high                                                                                                                   |
| <b>IP2Bus_MstRdReq</b>   | To IPIF   | Master requests read access to the bus, single <b>Bus2IP_Clk</b> high                                                                                                                    |
| <b>IP2Bus_MstBurst</b>   | To IPIF   | Master requests burst access to bus, goes low at data acknowledge for second to last datum                                                                                               |
| <b>IP2Bus_MstBusLock</b> | To IPIF   | Master requires bus to lock, not allowing release of this master until transfer is complete                                                                                              |
| <b>Bus2IP_MstWrAck</b>   | From IPIF | Acknowledge from bus that Master write datum has been accepted, single <b>Bus2IP_Clk</b> high                                                                                            |
| <b>Bus2IP_MstRdAck</b>   | From IPIF | Acknowledge from bus that Master read datum has been presented, single <b>Bus2IP_Clk</b> high                                                                                            |
| <b>Bus2IP_MstError</b>   | From IPIF | Bus tells IP that last transfer was in error, valid during <b>MstRdAck</b> or <b>MstWrAck</b> only                                                                                       |
| <b>Bus2IP_MstTimeout</b> | From IPIF | Bus tells IP that transfer timed out at bus slave. (Data may be unknown.) Valid in place of <b>MstRdAck</b> or <b>MstWrAck</b> . If valid at same time as ack, then data transferred OK. |
| <b>Bus2IP_MstRetry</b>   | From IPIF | Bus tells IP to get off bus now. Valid in place of <b>MstRdAck</b> or <b>MstWrAck</b><br><br><b>NOTE:</b> Acks are OK, but data must be assumed not have been transferred in any case.   |

## Master Parameters

**Table 3-10** shows the parameters that are associated with the IPIF Master module..

**Table 3-10: Parameters for the IPIF Master Module**

| Affects                    | Parameter                                                                                                                                                       | Value            | Type    |
|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|---------|
| General IPIF               | IPIF_DATA_BUS_WIDTH<br>Sets the size of the data bus for IPIF (where “n” in <b>Bus2IP_Data[0:n]</b> or <b>IP2Bus_Data[0:n]</b> is equal to IPIF_DATA_BUS_WIDTH) | 8, 16, or 32     | number  |
|                            | IPIF_NUMBER_OF_BYTE_ENABLES<br>Sets the number of byte enables                                                                                                  | 1, 2, or 4       | number  |
|                            | IPIF_NUMBER_OF_INTR<br>Sets the number of interrupts the IP provides to the IPIF (where “i” in <b>IP2Bus_Intr[0:i]</b> is equal to IPIF_NUMBER_OF_INTR)         | 0 to 8           | number  |
|                            | IPIF_INTR_ID<br>Sets the unique interrupt ID for this IPIF                                                                                                      | 16 bits          | number  |
|                            | IPIF_HAS_INTC<br>Sets whether the IPIF has a built-in interrupt controller (See <b>Figure 3-5</b> for more information)                                         | 0 or 1, 1 = true | boolean |
|                            | IP_HAS_OWN_INTC<br>Sets whether the IP has its own interrupt controller (See <b>Figure 3-5</b> for more information)                                            | 0 or 1, 1 = true | boolean |
| General IPIF Master Module | TBD                                                                                                                                                             |                  |         |

## IPIF Parameterization

Table 3-11 includes parameters and their accompanying descriptions for all modules.

Table 3-11: Complete Parameters for All IPIF Modules

| Affects                                 | Parameter                                                                                                                                                          | Value            | Type    | Module              |
|-----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|---------|---------------------|
| General IPIF                            | IPIF_DATA_BUS_WIDTH<br>Sets the size of the data bus for IPIF<br>(where “n” in <b>Bus2IP_Data[0:n]</b> or <b>IP2Bus_Data[0:n]</b> is equal to IPIF_DATA_BUS_WIDTH) | 8, 16, or 32     | number  | All                 |
|                                         | IPIF_NUMBER_OF_BYTE_ENABLES<br>Sets the number of byte enables                                                                                                     | 1, 2, or 4       | number  |                     |
|                                         | IPIF_NUMBER_OF_INTR<br>Sets the number of interrupts the IP provides to the IPIF<br>(where “i” in <b>IP2Bus_Intr[0:i]</b> is equal to IPIF_NUMBER_OF_INTR)         | 0 to 8           | number  |                     |
|                                         | IPIF_INTR_ID<br>Sets the unique interrupt ID for this IPIF                                                                                                         | 16 bits          | number  |                     |
|                                         | IPIF_HAS_INTC<br>Sets whether the IPIF has a built-in interrupt controller<br>(See Figure 3-5 for more information)                                                | 0 or 1, 1 = true | boolean |                     |
|                                         | IP_HAS_OWN_INTC<br>Sets whether the IP has its own interrupt controller<br>(See Figure 3-5 for more information)                                                   | 0 or 1, 1 = true | boolean |                     |
|                                         |                                                                                                                                                                    |                  |         |                     |
| General IPIF Slave DMA Handshake Module | DMA_HNDSHK_RESPONSE_TIME_MIN<br>Sets the minimum number of <b>Bus2IP_Clk</b> cycles in which the IPIF will respond with a DMA_ACK                                  | 0 to 255         | number  | Slave DMA Handshake |
|                                         | DMA_HNDSHK_DATA_VALID_WIDTH<br>Sets the number of <b>Bus2IP_Clk</b> cycles that the data will remain valid during DMA handshaking                                  | 0 to 255         | number  |                     |

Table 3-11: Complete Parameters for All IPIF Modules (Continued)

| Affects                                            | Parameter                                                                                                                                                             | Value                                     | Type   | Module                 |
|----------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|--------|------------------------|
| General IPIF Slave Control Register Module         | IPIF_NUMBER_OF_REGS<br>Sets the total number of registers at the bus width set by IPIF_DATA_BUS_WIDTH                                                                 | 1 to 255                                  | number | Slave Control Register |
|                                                    | IPIF_REG_BASE_ADDR<br>Sets the base address where IPIF registers will start in memory                                                                                 | 32-bit decode                             | number |                        |
|                                                    | IPIF_REG_BASE_ADDR_BIT_ENBL[0:31]<br>Allows specification of which address bits to decode in IPIF_REG_BASE_ADDR                                                       | 1 = decode respective address bit         | mask   |                        |
| IPIF Slave Control Register (Arrays of Parameters) | IPIF_REGx_NUMBER_OF_BITS<br>Defines the number of bits for each register (where x = 0 to IP_NUMBER_OF_REGS -1)                                                        | ordinal 0 to 32                           | number |                        |
|                                                    | IPIF_REGx_DATA_BIT_VALID_MASK[0:n]<br>Defines which bits in each register are physically present (where x = 0 to IP_NUMBER_OF_REGS -1 and n = IPIF_DATA_BUS_WIDTH -1) | 1 = bit position is used in this register | mask   |                        |
|                                                    | IPIF_REGx_READABLE_BITS[0:n]<br>Defines which bits in each register are readable by the IPIF (where x = 0 to IP_NUMBER_OF_REGS -1 and n = IPIF_DATA_BUS_WIDTH -1)     | 1 = bit position will be readable by IPIF | mask   |                        |
|                                                    | IPIF_REGx_WRITEABLE_BITS[0:n]<br>Defines which bits in each register are writable by the IPIF (where x = 0 to IP_NUMBER_OF_REGS -1 and n = IPIF_DATA_BUS_WIDTH -1)    | 1 = bit position will be writable by IPIF | mask   |                        |

Table 3-11: Complete Parameters for All IPIF Modules (Continued)

| Affects                                       | Parameter                                                                                                                                                                            | Value                             | Type   | Module     |
|-----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|--------|------------|
| General IPIF Slave SRAM Module                | IPIF_NUMBER_OF_SRAM_DECODER_REGIONS<br>Defines the number of decoded regions of memory address space for the IPIF Slave SRAM module                                                  | 1 to 4 decoded regions allowed    | number | Slave SRAM |
|                                               | IPIF_SLV_SRAM_ADDR_BUS_LSB<br>Sets the LSB index number for the address provided by the IPIF to the IP                                                                               | ordinal 0 to 31                   | number |            |
|                                               | IPIF_SLV_SRAM_ADDR_BUS_MSB<br>Sets the MSB index number for the address provided by the IPIF to the IP                                                                               | ordinal 0 to 31                   | number |            |
| IPIF Slave SRAM Module (Arrays of Parameters) | IPIF_SRAMd_BASE_ADDR<br>Sets the base address of the “d” region for the Slave SRAM module<br>(where d = 0 to IPIF_NUMBER_SRAM_DECODER_REGIONS –1)                                    | 32-bit decode of one region       | number |            |
|                                               | IPIF_SRAMd_BASE_ADDR_BIT_ENBL[0:31]<br>Allows specification of which bits address bits to decode in the IPIF_SRAMd_BASE_ADDR<br>(where d = 0 to IPIF_NUMBER_SRAM_DECODER_REGIONS –1) | 1 = decode respective address bit | mask   |            |

Table 3-11: Complete Parameters for All IPIF Modules (Continued)

| Affects                        | Parameter                                                                                                             | Value                             | Type    | Module     |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------|-----------------------------------|---------|------------|
| General IPIF Slave FIFO Module | IPIF_RD_FIFO_DEPTH<br>Sets the depth of the Read FIFO in IPIF_RD_FIFO_DATA_WIDTH units                                | 1 to 2048, in units of data width | number  | Slave FIFO |
|                                | IPIF_RD_FIFO_DATA_WIDTH<br>Sets the width of the Read FIFO data bus                                                   | 1 to 32 bits                      | number  |            |
|                                | IPIF_RD_FIFO_TRIG_THRESHOLD<br>Sets the trigger point of the Read FIFO if packet processing                           | 1 to 2048                         | number  |            |
|                                | IPIF_WR_FIFO_DEPTH<br>Sets the depth of the Write FIFO in IPIF_WR_FIFO_DATA_WIDTH units                               | 1 to 2048, in units of data width | number  |            |
|                                | IPIF_WR_FIFO_DATA_WIDTH<br>Sets the width of the Write FIFO data bus                                                  | 1 to 32 bits                      | number  |            |
|                                | IPIF_WR_FIFO_TRIG_THRESHOLD<br>Sets the trigger point of the Write FIFO if packet processing                          | 1 to 2048                         | number  |            |
|                                | IPIF_PACKET_FIFO_MODE_ENBL<br>Enables the IPIF Slave FIFO module with packet FIFO functionality                       | 0 or 1, 1 = true                  | boolean |            |
|                                | IPIF_WR_FIFO_NUMBER_OF_OCC_BITS<br>Sets the total number of bits in the <b>FIFO2IP_WrFIFO_Occupancy[0:fo]</b> signals | 0 TO 12                           | number  |            |
|                                | IPIF_WR_FIFO_NUMBER_OF_VAC_BITS<br>Sets the total number of bits in the <b>FIFO2IP_WrFIFO_Vacancy[0:fv]</b> signals   | 0 TO 12                           | number  |            |
| IPIF Master Module             | TBD                                                                                                                   |                                   |         | Master     |

## IPIF Signals

Table 3-12 includes signals and their accompanying descriptions for all modules.

Table 3-12: IPIF Signals Connecting to IP for All Modules

| Name             | Dir       | Description                                               | Slave DMA Handshake | Slave Ctrl Reg | Slave SRAM | Slave FIFO | Mstr |
|------------------|-----------|-----------------------------------------------------------|---------------------|----------------|------------|------------|------|
| Bus2IP_Clk       | From IPIF | Clock source (from global buffer)                         | x                   | x              | x          | x          | x    |
| Bus2IP_Reset     | From IPIF | Active-high synchronous reset source (from global buffer) | x                   | x              | x          | x          | x    |
| IP2Bus_Intr[0:i] | To IPIF   | Interrupt input from IP to IPIF                           | x                   | x              | x          | x          | x    |

Table 3-12: IPIF Signals Connecting to IP for All Modules (Continued)

| Name                     | Dir          | Description                                                                                                                                                         | Slave<br>DMA<br>Hand-<br>shake | Slave<br>Ctrl<br>Reg | Slave<br>SRAM | Slave<br>FIFO | Mstr |
|--------------------------|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------|----------------------|---------------|---------------|------|
| IP2Bus_Error             | To<br>IPIF   | Error signal from IP to IPIF (Valid only during a data acknowledge cycle)                                                                                           | x                              | x                    | x             | x             |      |
| IP2Bus_Retry             | To<br>IPIF   | Indicates IP wants master to retry the cycle                                                                                                                        | x                              | x                    | x             | x             |      |
| IP2Bus_ToutSup           | To<br>IPIF   | Forces the suppression of watch dog timeout on the bus                                                                                                              | x                              | x                    | x             | x             |      |
| Bus2IP_Data[0:n]         | From<br>IPIF | IPIF Write data (where n = IPIF_DATA_BUS_WIDTH - 1)                                                                                                                 | x                              | x                    | x             |               |      |
| IP2Bus_Data[0:n]         | To<br>IPIF   | IPIF Read data (where n = IPIF_DATA_BUS_WIDTH - 1)                                                                                                                  | x                              | x                    | x             |               |      |
| Bus2IP_BE[0:b]           | From<br>IPIF | Byte enable, 1 = byte lane valid (where b = IPIF_NUMBER_OF_BYTE_ENABLES - 1)                                                                                        | x                              | x                    | x             |               |      |
| Bus2IP_WrReq             | From<br>IPIF | Write request from IPIF to IP, single clock high                                                                                                                    |                                | x                    | x             |               |      |
| IP2Bus_WrAck             | To<br>IPIF   | Acknowledge that write data has been taken from Bus2IP_Data[0:n], single Bus2IP_Clk high                                                                            |                                | x                    | x             |               |      |
| Bus2IP_RdReq             | From<br>IPIF | Read request from IPIF to IP, single clock high                                                                                                                     |                                | x                    | x             |               |      |
| IP2Bus_RdAck             | To<br>IPIF   | Acknowledge that read data has been placed on IP2Bus_Data[0:n], single Bus2IP_Clk high                                                                              |                                | x                    | x             |               |      |
| IP2Bus_DMA_Req           | To<br>IPIF   | DMA handshake transfer request from IP                                                                                                                              | x                              |                      |               |               |      |
| Bus2IP_DMA_Ack           | From<br>IPIF | DMA handshake transfer acknowledge from IPIF                                                                                                                        | x                              |                      |               |               |      |
| Bus2IP_RegCE(r)          | From<br>IPIF | Clock enable of decoded "r" register (where r = 0 to IPIF_NUMBER_OF_REGS - 1)                                                                                       |                                | x                    |               |               |      |
| Bus2IP_Addr[al:ah]       | From<br>IPIF | IPIF Slave SRAM address, where:<br>al = IPIF_SLV_SRAM_ADDR_BUS_MSB<br>ah = IPIF_SLV_SRAM_ADDR_BUS_LSB<br>(al is a lower number than ah due to big-endian numbering) |                                |                      | x             |               |      |
| Bus2IP_SRAM_CE           | From<br>IPIF | Clock enable of decoded SRAM address space, high for entire bus cycle                                                                                               |                                |                      | x             |               |      |
| IP2Bus_Clk (optional)    | To<br>IPIF   | Optional clock source to control IPIF Slave FIFO module signals (not yet available)                                                                                 |                                |                      |               | x             |      |
| Bus2IP_FIFO_CE           | From<br>IPIF | Clock enable of decoded IPIF Slave FIFO address space, high for entire bus cycle                                                                                    |                                |                      |               | x             |      |
| FIFO2IP_WrFIFO_Data[0:n] | From<br>IPIF | IPIF write FIFO data (where n = IPIF_WR_FIFO_DATA_WIDTH - 1)                                                                                                        |                                |                      |               | x             |      |
| FIFO2IP_WrFIFO_Empty     | From<br>IPIF | IPIF write FIFO is empty when high                                                                                                                                  |                                |                      |               | x             |      |
| FIFO2IP_WrFIFO_WrReq     | From<br>IPIF | IPIF write FIFO request, single Bus2IP_Clk (or IP2Bus_Clk) high per datum                                                                                           |                                |                      |               | x             |      |
| IP2FIFO_WrFIFO_WrAck     | To<br>IPIF   | IPIF write FIFO acknowledge, single Bus2IP_Clk (or IP2Bus_Clk) high per datum                                                                                       |                                |                      |               | x             |      |

Table 3-12: IPIF Signals Connecting to IP for All Modules (Continued)

| Name                           | Dir          | Description                                                                                                                                               | Slave<br>DMA<br>Hand-<br>shake | Slave<br>Ctrl<br>Reg | Slave<br>SRAM | Slave<br>FIFO | Mstr |
|--------------------------------|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------|----------------------|---------------|---------------|------|
| IP2FIFO_RdFIFO_Data[0:n]       | To<br>IPIF   | IPIF read FIFO data<br>(where n = IPIF_RD_FIFO_DATA_WIDTH -1)                                                                                             |                                |                      |               | x             |      |
| FIFO2IP_RdFIFO_Full            | From<br>IPIF | IPIF write FIFO is empty when this signal is high                                                                                                         |                                |                      |               | x             |      |
| FIFO2IP_RdFIFO_WrReq           | To<br>IPIF   | IPIF read FIFO request, single <b>Bus2IP_Clk</b><br>(or <b>IP2Bus_Clk</b> ) high per datum                                                                |                                |                      |               | x             |      |
| IP2FIFO_RdFIFO_WrAck           | From<br>IPIF | IPIF read FIFO acknowledge, single <b>Bus2IP_Clk</b><br>(or <b>IP2Bus_Clk</b> ) high per datum                                                            |                                |                      |               | x             |      |
| IP2FIFO_WrFIFO_Mark            | To<br>IPIF   | Marks datum as first datum of packet, high during<br><b>IP2FIFO_WrFIFO_WrAck</b> only                                                                     |                                |                      |               | x             |      |
| IP2FIFO_WrFIFO_Restore         | To<br>IPIF   | Restores write FIFO to marked location, next request<br>will be at marked data (this signal is valid high only<br>during an <b>IP2FIFO_WrFIFO_WrAck</b> ) |                                |                      |               | x             |      |
| IP2FIFO_WrFIFO_Release         | To<br>IPIF   | Releases the marked position, write FIFO can now<br>accumulate data from mark forward                                                                     |                                |                      |               | x             |      |
| FIFO2IP_WrFIFO_Occupancy[0:fo] | From<br>IPIF | Indicates how much data is in the write FIFO (where<br>fo = IPIF_WR_FIFO_NUMBER_OF_OCC_BITS -1)                                                           |                                |                      |               | x             |      |
| IP2FIFO_RdFIFO_Mark            | To<br>IPIF   | Marks datum as first datum of packet, high during<br><b>IP2FIFO_RdFIFO_WrAck</b> only                                                                     |                                |                      |               | x             |      |
| IP2FIFO_RdFIFO_Restore         | To<br>IPIF   | Restores write FIFO to marked location, next request<br>will be at marked data (this signal is valid high only<br>during an <b>IP2FIFO_RdFIFO_WrAck</b> ) |                                |                      |               | x             |      |
| IP2FIFO_RdFIFO_Release         | To<br>IPIF   | Releases the marked position, read FIFO can now<br>accumulate data from mark forward                                                                      |                                |                      |               | x             |      |
| FIFO2IP_WrFIFO_Vacancy[0:fv]   | From<br>IPIF | Indicates how much data is in the read FIFO (where<br>fv = IPIF_WR_FIFO_NUMBER_OF_VAC_BITS -1)                                                            |                                |                      |               | x             |      |
| IP2Bus_Addr[0:31]              | To<br>IPIF   | 32-bit address of location Master wants to read or<br>write                                                                                               |                                |                      |               |               | x    |
| IP2Bus_MstBE[0:b]              | To<br>IPIF   | Master byte enable, 1 = byte late valid (where b =<br>IPIF_NUMBER_OF_BYTE_ENABLES -1)                                                                     |                                |                      |               |               | x    |
| IP2Bus_MstWrReq                | To<br>IPIF   | Master requests write access to the bus, single<br><b>Bus2IP_Clk</b> high                                                                                 |                                |                      |               |               | x    |
| IP2Bus_MstRdReq                | To<br>IPIF   | Master requests read access to the bus, single<br><b>Bus2IP_Clk</b> high                                                                                  |                                |                      |               |               | x    |
| IP2Bus_MstBurst                | To<br>IPIF   | Master requests burst access to bus, goes low at data<br>acknowledge for second to last datum                                                             |                                |                      |               |               | x    |
| IP2Bus_MstBusLock              | To<br>IPIF   | Master requires bus to lock, not allowing release of<br>this master until transfer is complete                                                            |                                |                      |               |               | x    |
| Bus2IP_MstWrAck                | From<br>IPIF | Acknowledge from bus that Master write datum has<br>been accepted, single <b>Bus2IP_Clk</b> high                                                          |                                |                      |               |               | x    |
| Bus2IP_MstRdAck                | From<br>IPIF | Acknowledge from bus that Master read datum has<br>been presented, single <b>Bus2IP_Clk</b> high                                                          |                                |                      |               |               | x    |



Table 3-12: IPIF Signals Connecting to IP for All Modules (Continued)

| Name                     | Dir          | Description                                                                                                                                                                              | Slave<br>DMA<br>Hand-<br>shake | Slave<br>Ctrl<br>Reg | Slave<br>SRAM | Slave<br>FIFO | Mstr |
|--------------------------|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------|----------------------|---------------|---------------|------|
| <b>Bus2IP_MstError</b>   | From<br>IPIF | Bus tells IP that last transfer was in error, valid during <b>MstRdAck</b> or <b>MstWrAck</b> only                                                                                       |                                |                      |               |               | x    |
| <b>Bus2IP_MstTimeout</b> | From<br>IPIF | Bus tells IP that transfer timed out at bus slave. (Data may be unknown.) Valid in place of <b>MstRdAck</b> or <b>MstWrAck</b> . If valid at same time as ack, then data transferred OK. |                                |                      |               |               | x    |
| <b>Bus2IP_MstRetry</b>   | From<br>IPIF | Bus tells IP to get off bus now. Valid in place of <b>MstRdAck</b> or <b>MstWrAck</b><br><br><b>NOTE:</b> B Acks are OK, but data must be assumed not have been transferred in any case. |                                |                      |               |               | x    |

# OPB Arbiter

---

## Summary

This document will provide the design specification for the On-Chip Peripheral Bus (OPB) Arbiter point solution.

## Introduction

The OPB Arbiter design described in this document has been designed incorporating the features described in the 32-bit implementation of the IBM *On-Chip Peripheral Bus Arbiter Core User's Manual*, version 1.5.

## OPB Arbiter Overview

### Features

The OPB Arbiter is a soft IP core designed for Xilinx FPGAs and contains the following features:

- OPB 8-bit slave interface
- OPB Arbitration
  - arbitrates between four OPB masters (the number of masters will be parameterizable in the deliverable IP design)
  - arbitration priorities among masters programmable via register write
  - dynamic priority re-ordering mode implementing a true least recent used (LRU) algorithm enabled via a control register write
- Two bus parking modes selectable via control register write:
  - park on selected OPB master (specified in control register)
  - park on last OPB master which was granted OPB access
- Watchdog timer, which asserts the OPB time-out signal if a slave response is not detected within 16 clock cycles
- System clock frequency of 100 MHz

### OPB Arbitration Protocol

OPB Bus arbitration proceeds by the following protocol:

1. An OPB master asserts its bus request signal.
2. The OPB Arbiter receives the request and outputs an individual grant signal to each master according to its priority and the state of the other requests.

3. An OPB master samples its grant signal at the rising edge of the OPB clock. In the following cycle, the OPB master may initiate a data transfer between the master and a slave by asserting its select signal.

The OPB Arbiter only issues a bus grant signal during valid arbitration cycles, which are defined as either:

- *Idle*, which means that the OPB\_select and OPB\_busLock are deasserted, indicating that no data transfer is in progress, or
- *Overlapped arbitration cycle*, which means that the OPB\_xferAck is asserted, indicating the final cycle in a data transfer, and OPB\_busLock is not asserted. Arbitration in this cycle allows another master to begin a transfer in the following cycle, avoiding the need for a “dead” cycle on the bus.

Figure 4-1 shows the typical OPB arbitration protocol.

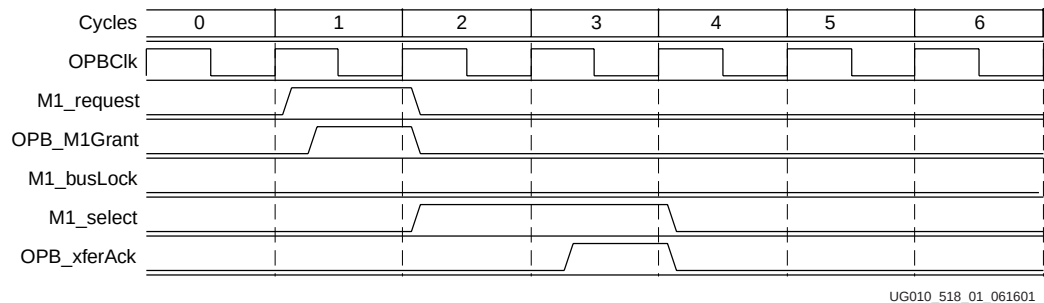


Figure 4-1: OPB Basic Bus Arbitration

An OPB master device need not deassert its request upon receipt of a bus grant signal if it has multiple bus transfer cycle to perform. Figure 4-2 shows an OPB arbitration cycle in which an OPB master asserts bus request continuously for four data transfer cycles. Bus grant is asserted during valid arbitration cycles.

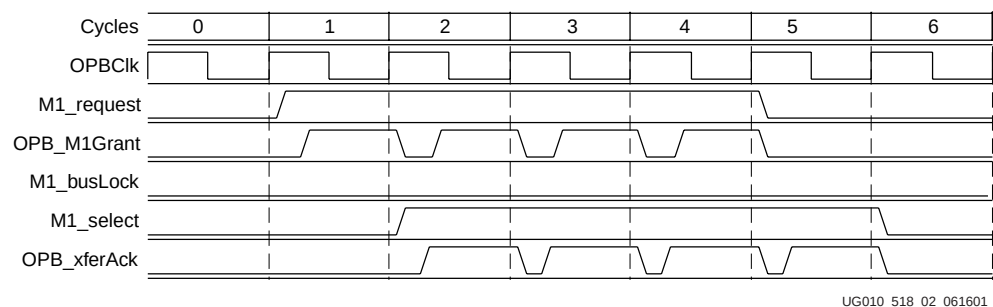
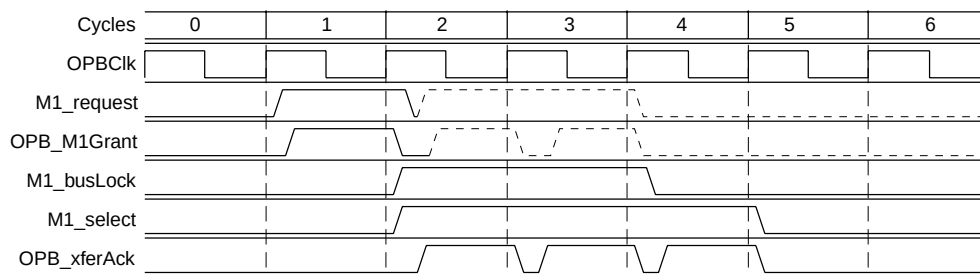


Figure 4-2: Continuous Master Bus Request

Even if an OPB master asserts request continuously, it will not necessarily receive a valid grant signal. Other OPB masters with higher bus priority may request the OPB and will be granted according to OPB Arbiter priority. If an OPB master device needs a non-interruptible sequence of bus cycles, it can use the bus lock signal for this purpose.

If an OPB master asserts the bus lock signal upon assuming control of the bus, the OPB Arbiter will continue to grant the OPB to the master which locked the bus as shown in Figure 4-3. Grant signals will be generated during valid arbitration cycles if the master asserts its request signal. The master which asserted bus lock will retain control of the bus until bus lock is deasserted for at least one complete cycle. Bus request and grant signals have no effect on bus arbitration. The OPB master may proceed with data transfer cycles while asserting bus lock without engaging in bus arbitration and without regard to the

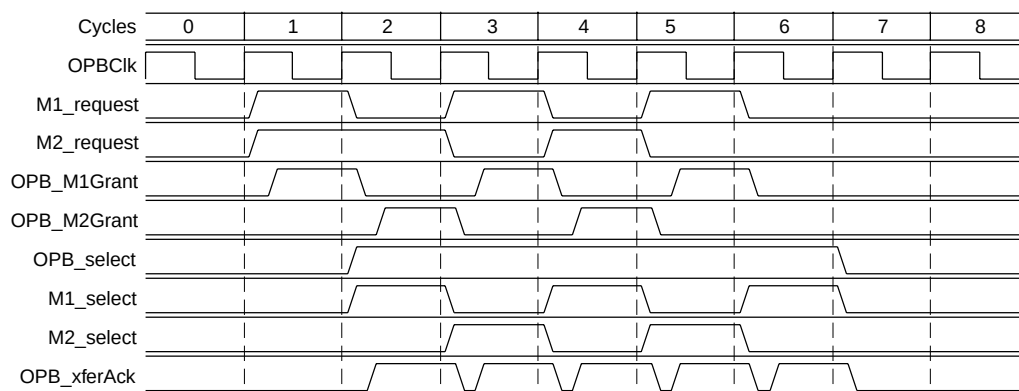
state of the request and grant signals. The OPB Arbiter will detect the bus lock signal and will continue to grant the bus to the current master, regardless of other (higher priority) requests.



UG010\_518\_03\_061601

Figure 4-3: Bus Locking

Figure 4-4 shows multiple bus request or overlapped bus arbitration. Both OPB Master 1 and OPB Master 2 simultaneously request the bus. Master 1 has a higher priority and is granted the bus. During cycle 2, Master 1 completes its first transaction and Master 2 is granted the bus for cycle 3. Thus, during cycle 2, the arbitration for the bus is overlapped with a data transfer. This overlapped bus arbitration improves the bandwidth of the bus.



UG010\_518\_04\_061601

Figure 4-4: Multiple Bus Request Arbitration

## OPB Arbiter I/O Signals

The I/O signals for the OPB Arbiter are listed in Table 4-1. The interfaces referenced in this table are shown in Figure 4-7 in the OPB Arbiter block diagram.

Table 4-1: OPB Arbiter I/O Signals

| Signal Name    | Interface | I/O | Description                  |
|----------------|-----------|-----|------------------------------|
| ARB_DBus[0:31] | IPIF      | O   | Arbiter output data bus      |
| ARB_XferAck    | IPIF      | O   | Arbiter transfer acknowledge |
| OPB_Abus[0:31] | IPIF      | I   | OPB address bus              |
| OPB_BE[0:3]    | IPIF      | O   | OPB byte enables             |
| OPB_Dbus[0:31] | IPIF      | I   | OPB data bus                 |

Table 4-1: OPB Arbiter I/O Signals (Continued)

| Signal Name | Interface                           | I/O | Description                                                                                 |
|-------------|-------------------------------------|-----|---------------------------------------------------------------------------------------------|
| OPB_RNW     | IPIF                                | I   | Read not Write<br>(OR of all master RNW signals)                                            |
| M0_request  | Arbitration Logic                   | I   | Request from OPB Master 0                                                                   |
| M1_request  | Arbitration Logic                   | I   | Request from OPB Master 1                                                                   |
| M2_request  | Arbitration Logic                   | I   | Request from OPB Master 2                                                                   |
| M3_request  | Arbitration Logic                   | I   | Request from OPB Master 3                                                                   |
| OPB_xferAck | Arbitration Logic                   | I   | Transfer Acknowledge indicating end<br>of data transfer cycle<br>(OR of all slave xferAcks) |
| OPB_select  | Arbitration Logic<br>Watchdog Timer | I   | Master has taken control of the bus<br>(OR of all master selects)                           |
| OPB_retry   | Watchdog Timer                      | I   | Bus cycle retry<br>(OR of all slave retrys)                                                 |
| OPB_toutSup | Watchdog Timer                      | I   | Suppress timeout<br>(OR of all slave toutSups)                                              |
| OPB_timeout | Watchdog Timer                      | O   | Timeout signal for OPB                                                                      |
| OPB_buslock | Park/Lock Logic                     | I   | Bus lock<br>(OR of all master buslocks)                                                     |
| OPB_M0Grant | Park/Lock Logic                     | O   | Grant to OPB Master 0<br>(default highest priority)                                         |
| OPB_M1Grant | Park/Lock Logic                     | O   | Grant to OPB Master 1                                                                       |
| OPB_M2Grant | Park/Lock Logic                     | O   | Grant to OPB Master 2                                                                       |
| OPB_M3Grant | Park/Lock Logic                     | O   | Grant to OPB Master 3<br>(default lowest priority)                                          |
| OPB_clk     | System                              | I   | System clock                                                                                |
| IPIF_Rst    | System                              | I   | System Reset<br>(active High)                                                               |

## OPB Arbiter Register Descriptions

The OPB Arbiter contains two byte-wide addressable registers for read/write operations as shown in Table 4-2. The base address for these registers is set in the parameter C\_ARB\_BAR which represents the first 29 bits of the OPB address of these registers.

Table 4-2: OPB Arbiter Registers

| Register Name                 | OPB Address     | Access     |
|-------------------------------|-----------------|------------|
| OPB Arbiter Priority Register | C_ARB_BAR & 000 | Read/Write |
| OPB Arbiter Control Register  | C_ARB_BAR & 100 | Read/write |

## OPB Arbiter Priority Register

The four 2-bit fields of the 8-bit Priority register hold the master IDs of the OPB masters. As shown in [Figure 4-5](#) and [Table 4-3](#), each master's relative priority is determined by its ID's location within this register. Bits 0 and 1 hold the ID of the master with highest priority (level 0), bits 2 and 3 hold the ID of the master with medium high priority (level 1), bits 4 and 5 hold the ID of the master with medium low priority (level 2), and bits 6 and 7 hold the ID of the master with lowest priority (level 3). This register is reset to a value of 00011011 which assigns level 0 (highest) priority to Master 0, level 1 (medium high) priority to Master 1, level 2 (medium low) priority to Master 2, and level 3 (lowest) priority to Master 3.

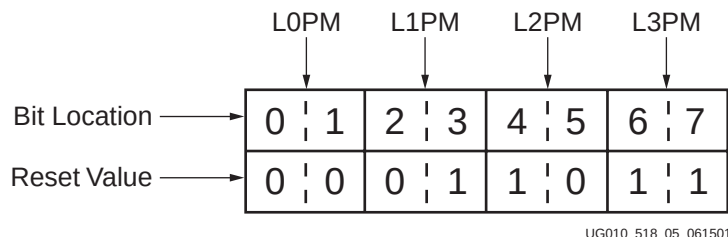


Figure 4-5: OPB Arbiter Priority Register

Table 4-3: OPB Arbiter Priority Register Bit Definitions

| Bit(s) | Name | Core Access | Reset Value | Description                                                                                                                                                                                                                                                                                                                              |
|--------|------|-------------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0-1    | L0PM | Read/Write  | 00          | Level 0 Priority Master ID. This field contains the ID of the master with the highest priority. <ul style="list-style-type: none"> <li>00 - Master 0 has high priority</li> <li>01 - Master 1 has high priority</li> <li>10 - Master 2 has high priority</li> <li>11 - Master 3 has high priority</li> </ul>                             |
| 2-3    | L1PM | Read/Write  | 01          | Level 1 Priority Master ID. This field contains the ID of the master with medium high priority. <ul style="list-style-type: none"> <li>00 - Master 0 has medium high priority</li> <li>01 - Master 1 has medium high priority</li> <li>10 - Master 2 has medium high priority</li> <li>11 - Master 3 has medium high priority</li> </ul> |
| 4-5    | L2PM | Read/Write  | 10          | Level 2 Priority Master ID. This field contains the ID of the master with medium low priority. <ul style="list-style-type: none"> <li>00 - Master 0 has medium low priority</li> <li>01 - Master 1 has medium low priority</li> <li>10 - Master 2 has medium low priority</li> <li>11 - Master 3 has medium low priority</li> </ul>      |
| 6-7    | L3PM | Read/Write  | 11          | Level 3 Priority Master ID. This field contains the ID of the master with the lowest priority. <ul style="list-style-type: none"> <li>00 - Master 0 has low priority</li> <li>01 - Master 1 has low priority</li> <li>10 - Master 2 has low priority</li> <li>11 - Master 3 has low priority</li> </ul>                                  |

The values of the priority fields in the priority register must be unique; the same master ID can not be loaded into more than one field of the priority register.

The priority register can be accessed from the OPB for read and write operations.

## OPB Arbiter Control Register

Update of the Priority Register is controlled by fields in the 8-bit Control Register as shown in Figure 4-6. Table 4-4 shows the Control Register bit definitions.

The Control register is reset to a value of 00000000, which disables dynamic priority and bus parking. If this register is not modified after reset, the OPB Arbiter will operate using a fixed priority among the OPB bus masters based on the value of the OPB Arbiter Priority Register.

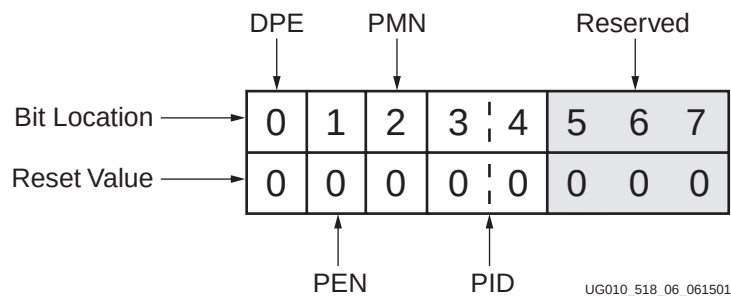


Figure 4-6: OPB Arbiter Control Register

Table 4-4: OPB Arbiter Control Register Bit Definitions

| Bit(s) | Name | Core Access | Reset Value | Description                                                                                                                                                                                                                                                                                                                                                                            |
|--------|------|-------------|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0      | DPE  | Read/Write  | 0           | <b>Dynamic Priority Enable.</b> Enables dynamic priority algorithm and update of the priority register. <ul style="list-style-type: none"> <li>0 - dynamic priority disabled</li> <li>1 - dynamic priority enabled</li> </ul>                                                                                                                                                          |
| 1      | PEN  | Read/Write  | 0           | <b>Park Enable.</b> Enables parking on a master when no other masters have requests asserted. <ul style="list-style-type: none"> <li>0 - parking disabled</li> <li>1 - parking enabled</li> </ul>                                                                                                                                                                                      |
| 2      | PMN  | Read/Write  | 0           | <b>Park On Master Not Last.</b> When parking is enabled, this bit determines if the arbiter parks on the master who was last granted the bus or on the master specified by the Parked Master ID bits. <ul style="list-style-type: none"> <li>0 - park on the master who had just was granted the bus</li> <li>1 - park on the master specified by the Parked Master ID bits</li> </ul> |

Table 4-4: OPB Arbiter Control Register Bit Definitions (Continued)

| Bit(s) | Name | Core Access | Reset Value | Description                                                                                                                                                                                                                                                                                                        |
|--------|------|-------------|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3:4    | PID  | Read/Write  | 00          | <b>Parked Master ID.</b> These bits contain the ID of the master to park on if parking is enabled and the Park On Master Not Last bit is set. <ul style="list-style-type: none"> <li>00 - park on Master 0</li> <li>01 - park on Master 1</li> <li>10 - park on Master 2</li> <li>11 - park on Master 3</li> </ul> |
| 5:7    |      |             |             | Reserved                                                                                                                                                                                                                                                                                                           |

The OPB Arbiter Control Register can be accessed from the OPB bus for read and write operations.

## OPB Arbiter Block Diagram

The top-level block diagram for the OPB Arbiter is shown in [Figure 4-7](#). The IPIF block is the OPB bus interface block that handles the OPB bus protocol for reading and writing the Priority Register and the Control Register within the OPB Arbiter. The IP2OPB data mux contains the data multiplexor required to output data to the OPB bus during a read cycle. The Arbitration Logic block determines which incoming request has the highest priority and the Park /Lock Logic block determines which master should be granted the bus based on this priority as well as whether the bus is locked or if bus parking is enabled. The Watchdog Timer asserts the OPB timeout signal if a slave response (OPB\_xferAck, OPB\_retry, or OPB\_toutSup) has not been received within 16 clock cycles of the master taking control of the bus.

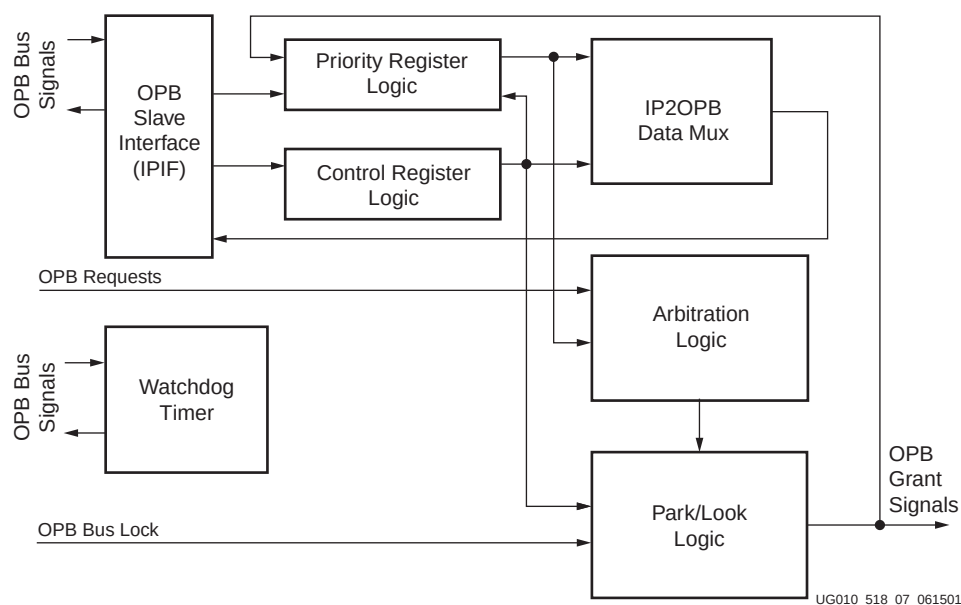


Figure 4-7: OPB Arbiter Top-level Block Diagram

The following sections describe each module in the block diagram.



## OPB Slave Interface (IPIF)

The IPIF block implements a byte-wide slave interface to the OPB. Its address in the OPB memory map is determined by setting the parameter C\_ARB\_BAR. The upper 29 most significant bits of the OPB address bus are compared to this value during valid OPB transfer cycles. If there is a match, the lower three least significant bits are used to index one of the OPB Arbiter's two addressable registers for read and write operations as shown in Table 4-2.

The IPIF block outputs a register clock enable for the register which was addressed along with a write or read request depending on the type of data transfer specified by the master. When the data transfer is complete, the IPIF block generates the transfer acknowledge. The data transfer protocol from the IPIF to the rest of the arbiter circuitry for OPB read and write operations is shown in Figure 4-8. Please refer to Chapter 3, **IPIF Specification**, for more detailed information.

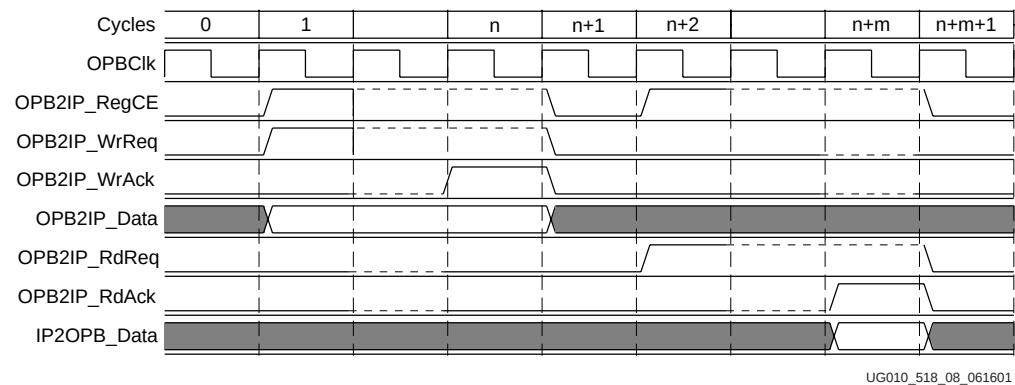


Figure 4-8: IPIF Data Transfer Protocol

## Priority Register Logic

The Priority Register logic contains the priority register and the logic to update the priority of the OPB masters if dynamic priority has been enabled in the Control Register. Descriptions of the OPB Arbiter Priority Register and the OPB Arbiter Control Register are found in section **OPB Arbiter Register Descriptions**, page 1388.

### Priority Register Update Logic

The Priority Register is loaded at reset and may be updated via a write to the OPB Arbiter core as an OPB slave or by enabling dynamic priority mode. Dynamic priority mode is enabled by writing a 1 to the DPEN bit of the control register. Disabling dynamic priority mode by setting the DPEN bit to 0 effectively freezes the value of the Priority register (unless updated by an OPB write to the register) and thus its ordering of arbitration priorities among the four attached master devices. Setting the master priorities by software and not allowing them to update results in a static assignment of priority among the four masters.

Upon reset, the Priority register contains the reset values shown in Table 4-3 and the dynamic priority bit is disabled. Thus if neither the Priority register or the Control register are written, the OPB Arbiter will operate in a fixed priority mode with relative priorities

determined by the connection of master devices to the request/grant signals. Fixed priority arbitration is shown in **Figure 4-9**.

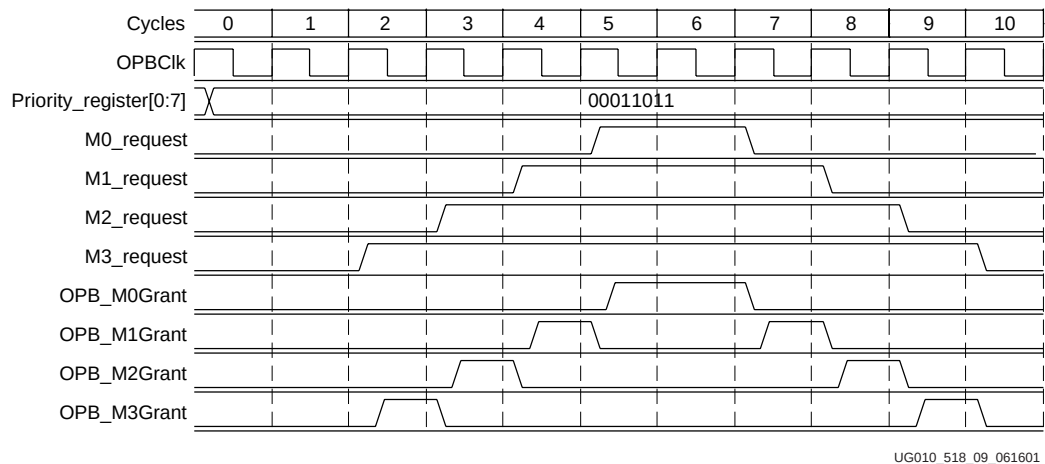


Figure 4-9: Fixed Priority Arbitration

When dynamic priority mode is enabled, the Priority Register reorders its contents after every request-grant cycle, moving the ID of the most recently granted master to the lowest priority slot of the register and moving all other master IDs up one level of priority. The dynamic priority mode operation results in an implementation of the least recently used (LRU) algorithm. The lowest priority master ID will be the one which was granted the bus most recently, and the highest priority master ID will be the one which was granted the bus the least frequently in the recent past. Dynamic priority arbitration is shown in **Figure 4-10**.

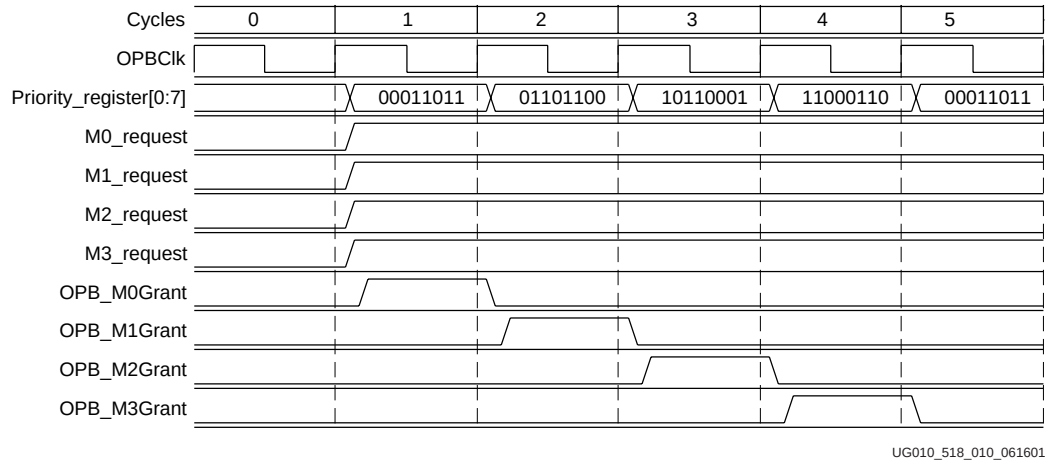


Figure 4-10: Dynamic Priority Arbitration

The values to be loaded into the fields of the Priority register are either the value written to the register from the OPB or a shift of the master id from the next lowest priority field. In the case of the low priority field of the Priority register, the ID of the master last granted the bus is loaded into this field. The master ID of the master granted the bus is loaded into the lowest priority field and the IDs in all other slots up to the priority of the one just granted move up one position in priority. The priority slots above the one just granted hold their master ID values.

## Block Diagram

The block diagram for the Priority register and the register update logic is shown in Figure 4-11.

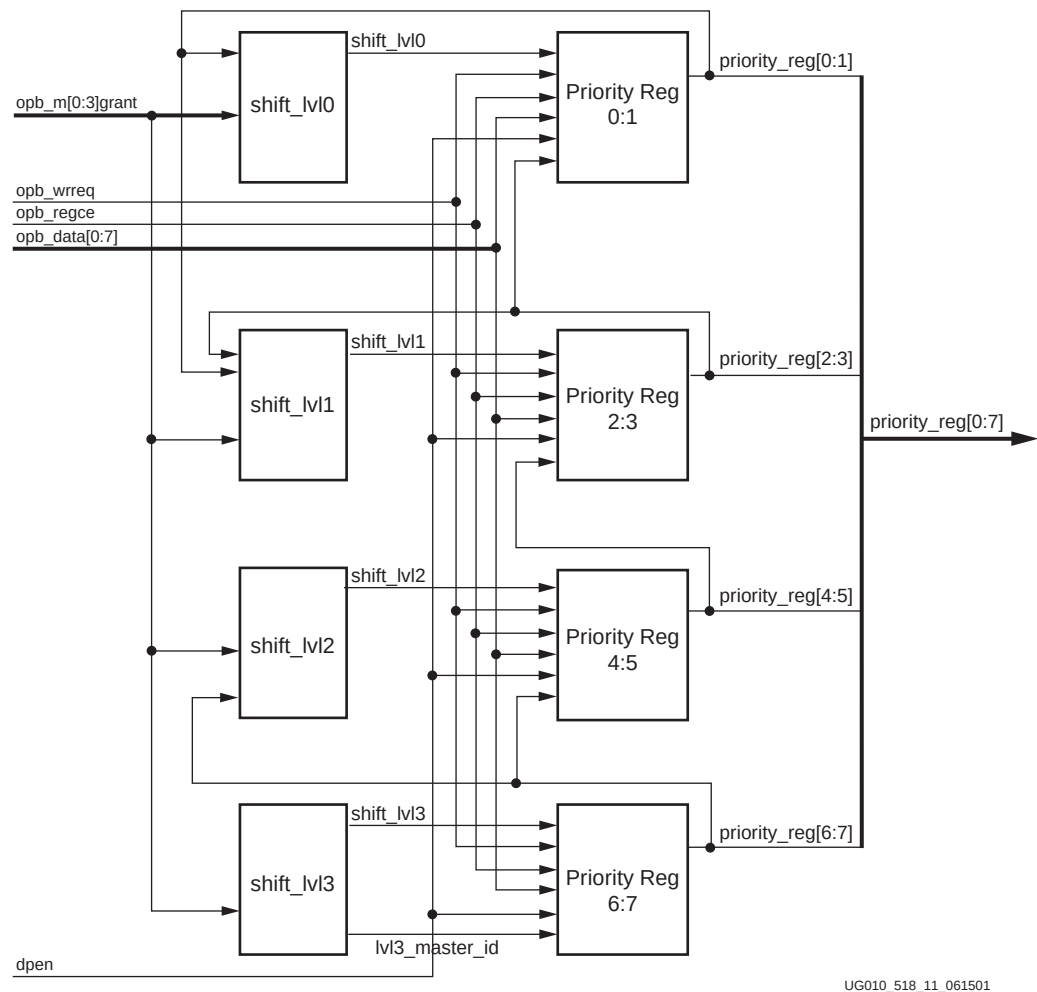


Figure 4-11: Priority Register Logic

## Control Register Logic

The Control Register Logic block simply contains the OPB Arbiter Control Register described in section **OPB Arbiter Control Register**, page 1390.

## IP2OPB Data Mux

When a read of either the OPB Arbiter Priority Register or the OPB Arbiter Control Register is requested on the OPB, the IP2OPB Data Mux outputs the requested data to the IPIF which sends this data to the OPB with the required protocol.

## Arbitration Logic

Figure 4-12 depicts the arbitration logic implemented in the OPB Arbiter.

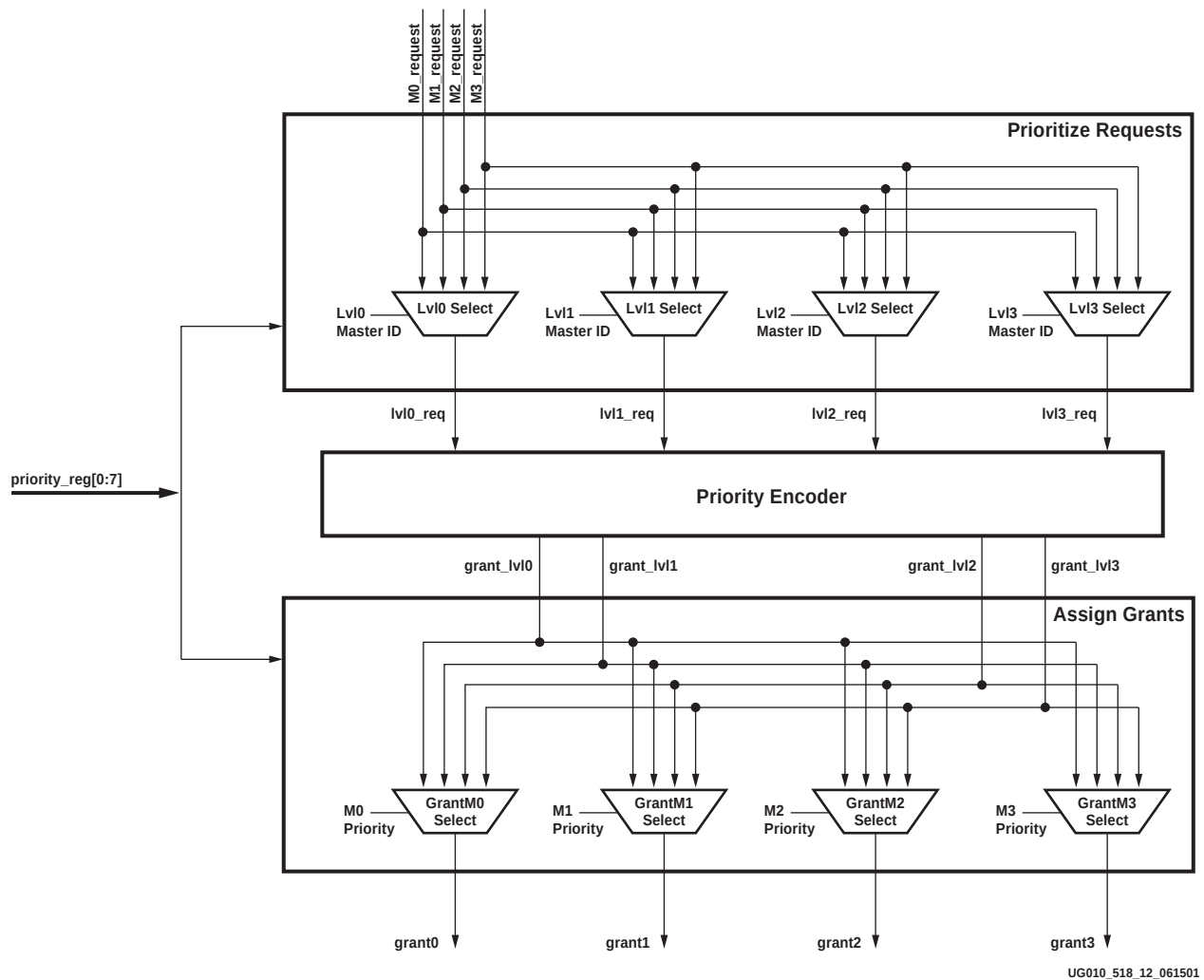


Figure 4-12: Arbitration Logic

All four master request signals are input to the Prioritize Request block which consists of four, 4-to-1 multiplexors. The select code for each multiplexor is the master ID for each of the 4 priority levels from the Priority register. Therefore, the master's requests are prioritized into the signals lv0\_req, lv1\_req, lv2\_req, and lv3\_req based on the requesting masters' priorities.

The prioritized request signals are then input into the priority encoder which determines which priority grant is asserted, i.e., grant\_lv0, grant\_lv1, grant\_lv2, and grant\_lv3.

The prioritized grant signals are then input to the Assign Grants block to determine which master's grant signal is asserted based on the priority of that master, again determined by examination of the master IDs in the priority register. This block consists of four 4-to-1 multiplexors (one for each master) with the master's priority code as the multiplexor's selects. Each prioritized grant signal is connected to all four multiplexors and the master's priority code selects the appropriate prioritized grant signal to be output to that master.

The arbitration logic also contains the logic for detecting valid arbitration cycles which is input to the Park/Lock logic. Valid arbitration cycles are defined as when either the OPB\_select signal is deasserted indicating no data transfer is in progress or when

OPB\_XferAck is asserted, indicating the final cycle in a data transfer and OPB\_busLock is not asserted.

## Park/Lock Logic

The Park/Lock logic processes the raw grant outputs from the Arbitration Logic. It provides for the parking and locking functionality of the OPB Arbiter and generates the final grant signals which are sent to the OPB master devices during valid arbitration cycles as shown in Figure 4-13.

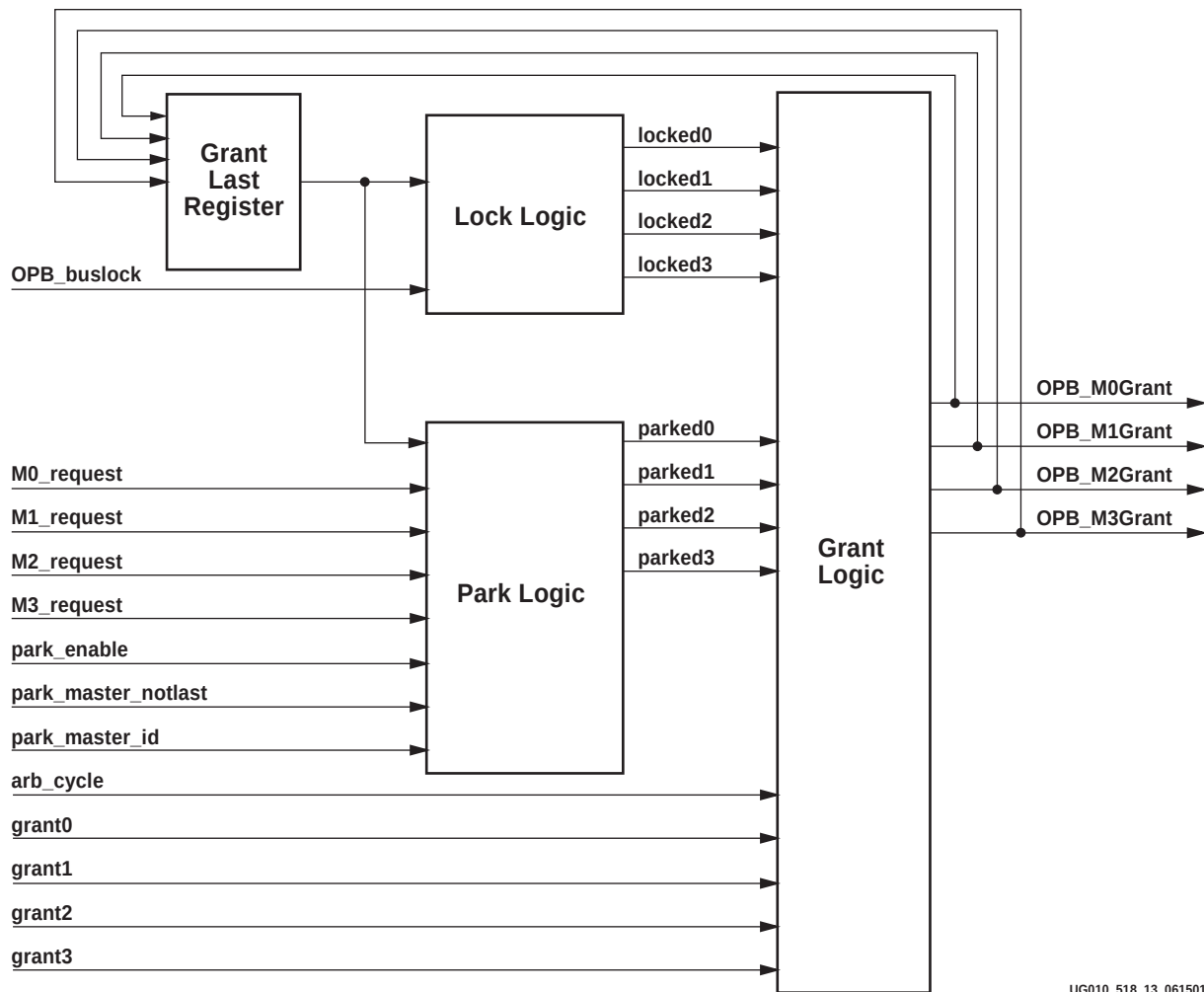


Figure 4-13: Park/Lock Logic

### Grant Last Register

The Grant Last Register is a 4-bit register which holds the state of the grant outputs from the last request/grant arbitration cycle. This information is used to determine which master currently has control of the bus to implement bus locking and park on last master parking.

### Lock Logic

The OPB Arbiter implements bus locking as defined in the OPB specification. Mn\_busLock signals from all attached masters are ORed together to form OPB\_busLock, which is an input to the OPB Arbiter. When OPB\_busLock is asserted, bus arbitration is locked to the last granted master (as indicated by the Grant Last register). All other master Grant

outputs are gated off and will not be asserted, regardless of the state of the Request inputs or the programmed priorities.

The locked master's grant signal will be asserted in response to its request signal during valid arbitration cycles. However, the locked master need not assert its request or receive an asserted grant signal to control the bus. The master owns the bus by virtue of asserting Mn\_busLock after being granted the bus and before another valid arbitration cycle. Bus locking is shown in [Figure 4-3](#).

## Park Logic

The OPB Arbiter supports bus parking, where a master's grant signal is asserted during valid arbitration cycles, when no other master devices are requesting. This reduces latency for the parked master eliminating the need for a request/grant cycle when initiating a new OPB transfer.

Asserting a grant signal for parking is considered an arbitration, since it determines which device controls the bus. If dynamic priority mode is enabled, the ID of the parked master will be shifted to the lowest priority slot of the Priority Register. The master will remain parked (its grant signal asserted) so long as no other master asserts a request signal. If the parked master and another master assert request at the same time, the other master will win arbitration because the parked master's priority was lowered due to the parked condition, even though it might never have utilized the parked grant signal to initiate a transfer.

Bus parking is enabled or disabled by the value of the Park Enable control bit in the OPB Arbiter Control Register (see [Table 4-4](#)). The bus can either be parked on the master who was last granted the bus, or on a specified master as indicated by the Park Master ID bits in the OPB Arbiter Control Register.

### Park on Master Not Last

When bus parking is enabled (bit PEN = 1 in the OPB Arbiter Control Register) and Park on Master Not Last is selected (bit PMNL = 1 in the OPB Arbiter Control Register), the bus will be parked on the master whose ID is contained in the Park Master ID (PMID) field of the OPB Arbiter Control Register. This master's grant signal will be asserted during valid arbitration cycles when no other master's request signal is asserted as shown in [Figure 4-14](#).

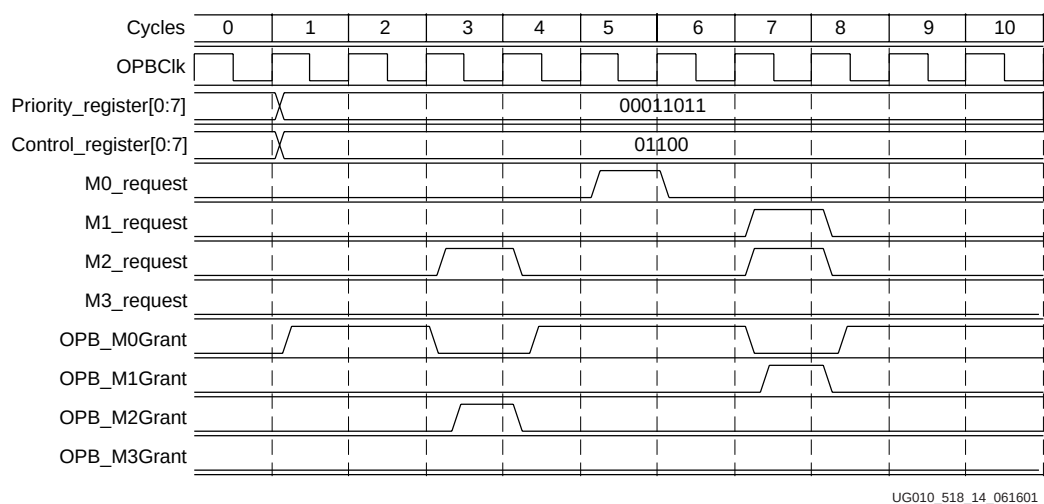


Figure 4-14: Bus Parking on Master Not Last

### Park on Last Master

When bus parking is enabled (bit PEN = 1 in the OPB Arbiter Control Register) and Park on Master Not Last is negated (bit PMNL = 0 in the OPB Arbiter Control Register), the bus will be parked on the master which was most recently granted the bus as indicated by the Grant Last register. This master's grant signal will be asserted during valid arbitration cycles when no other master's request signal is asserted as shown in [Figure 4-15](#).

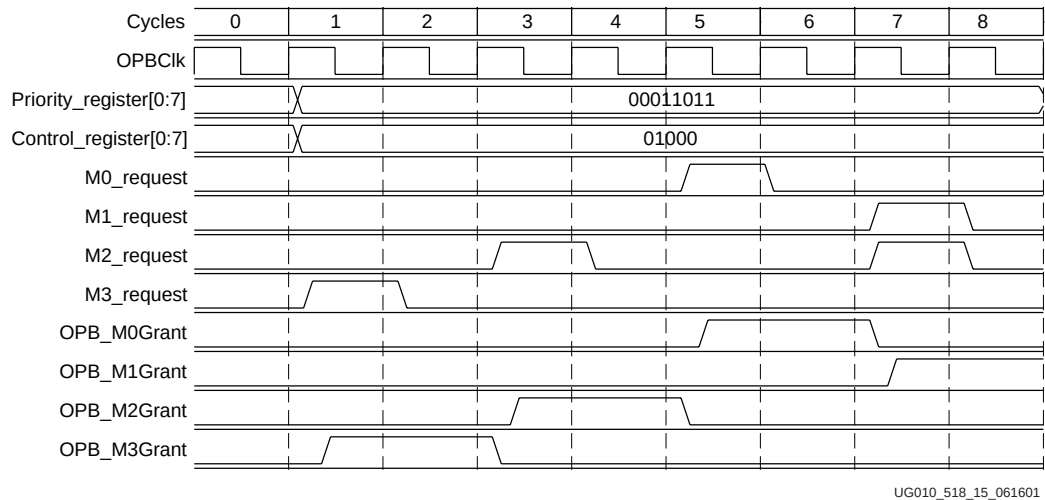


Figure 4-15: Bus Parking on Last Master

### Watchdog Timer

The watchdog timer contains the OPB Arbiter logic which generates the OPB\_timeout signal within the 16th cycle following the assertion of OPB\_select if there is no response from a slave (OPB\_xferAck or OPB\_retry) and if toutSup is not asserted by an addressed slave device to suppress the timeout.

Upon assertion of OPB\_timeout, the master device which initiated the transfer cycle must terminate the transfer by deasserting Mn\_select in the cycle following the assertion of OPB\_timeout as shown in [Figure 4-16](#). If OPB\_busLock is not asserted, the OPB Arbiter will perform a bus arbitration in the cycle in which OPB\_select is deasserted. If OPB\_busLock is asserted, the requesting master retains control of the OPB, but must still deassert Mn\_select following the assertion of OPB\_timeout for at least one cycle.

If OPB\_xferAck or OPB\_retry are asserted in the 16th cycle following the assertion of Mn\_select coincident to the assertion of OPB\_timeout, the master device should ignore OPB\_timeout, and respond to the slave's OPB\_xferAck or OPB\_retry signal.

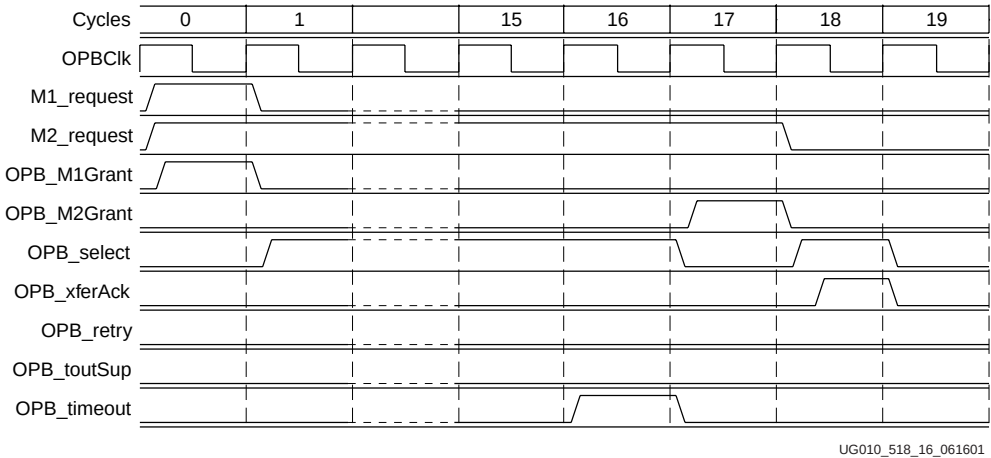


Figure 4-16: OPB Timeout Error

To prevent a bus timeout, an OPB slave must assert OPB\_toutSup (OR of all slave's Sln\_ToutSup) within 16 cycles from the assertion of OPB\_select. OPB\_toutSup will be used by the OPB Arbiter to suspend the timeout counter and suppress the assertion of OPB\_timeout. OPB\_toutSup must remain asserted until the slave can complete the requested operation. OPB timeout error suppression is shown in Figure 4-17.

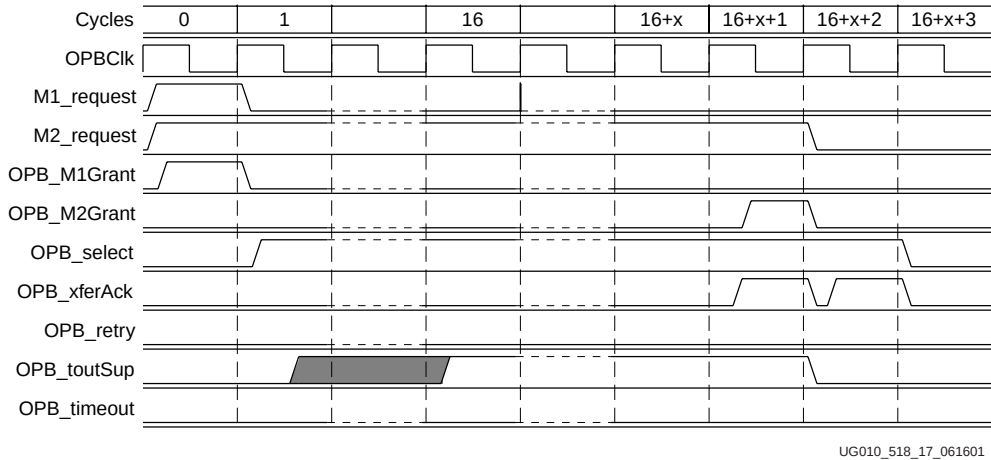


Figure 4-17: OPB Timeout Error Suppression





## OPB Configuration ROM Module

### Overview

The Configuration Read-Only Memory (CROM) module is a design example of an OPB slave device that uses the Xilinx Intellectual Property InterFace (IPIF) module. It consists of a 512 x 32-bit block RAM (BRAM) connected to an IPIF Slave SRAM Interface. The CROM module can act as a generic memory device, but is intended to be used for hardware discovery purposes. At boot time, the CPU would read the CROM module to identify the OPB devices in the system and call the appropriate device drivers. The CROM Structure Specification describes the organization and description of the memory contents in detail. Though the CROM module is intended to be a read-only device, write logic is included for system debugging and to better illustrate an IPIF-based device.

### Related Documents

The following documents provide additional information:

- *CROM Structure Specification*
- *IBM CoreConnect™ 64-Bit On-Chip Peripheral Bus: Architecture Specifications, Version 2.1*
- *Virtex-II Pro™ Platform FPGAs (Advance Product Specification)*
- *Using Data2BRAM*

### Features

- 32-bit OPB slave utilizing a 32-bit IPIF Slave SRAM interface
- 2 KB of memory space
- Supports reads and writes with byte enables

### Module Port Interface

Table 5-1: Global Signals

| Name    | Direction | Description      |
|---------|-----------|------------------|
| OPB_Clk | Input     | OPB system clock |
| OPB_Rst | Input     | OPB system reset |

Table 5-2: OPB Slave Signals

| Name           | Direction | Description                |
|----------------|-----------|----------------------------|
| OPB_ABus[0:31] | Input     | OPB address bus            |
| OPB_BE[0:3]    | Input     | OPB byte enables           |
| OPB_DBus[0:31] | Input     | OPB data bus               |
| OPB_RNW        | Input     | OPB read not write         |
| OPB_select     | Input     | OPB select                 |
| OPB_seqAddr    | Input     | OPB sequential address     |
| S1_DBus[0:31]  | Output    | Slave data bus             |
| S1_DBusEn      | Output    | Slave data bus enable      |
| S1_errAck      | Output    | Slave error acknowledge    |
| S1_retry       | Output    | Slave bus cycle retry      |
| S1_toutSup     | Output    | Slave timeout suppress     |
| S1_xferAck     | Output    | Slave transfer acknowledge |

Table 5-3: Parameters

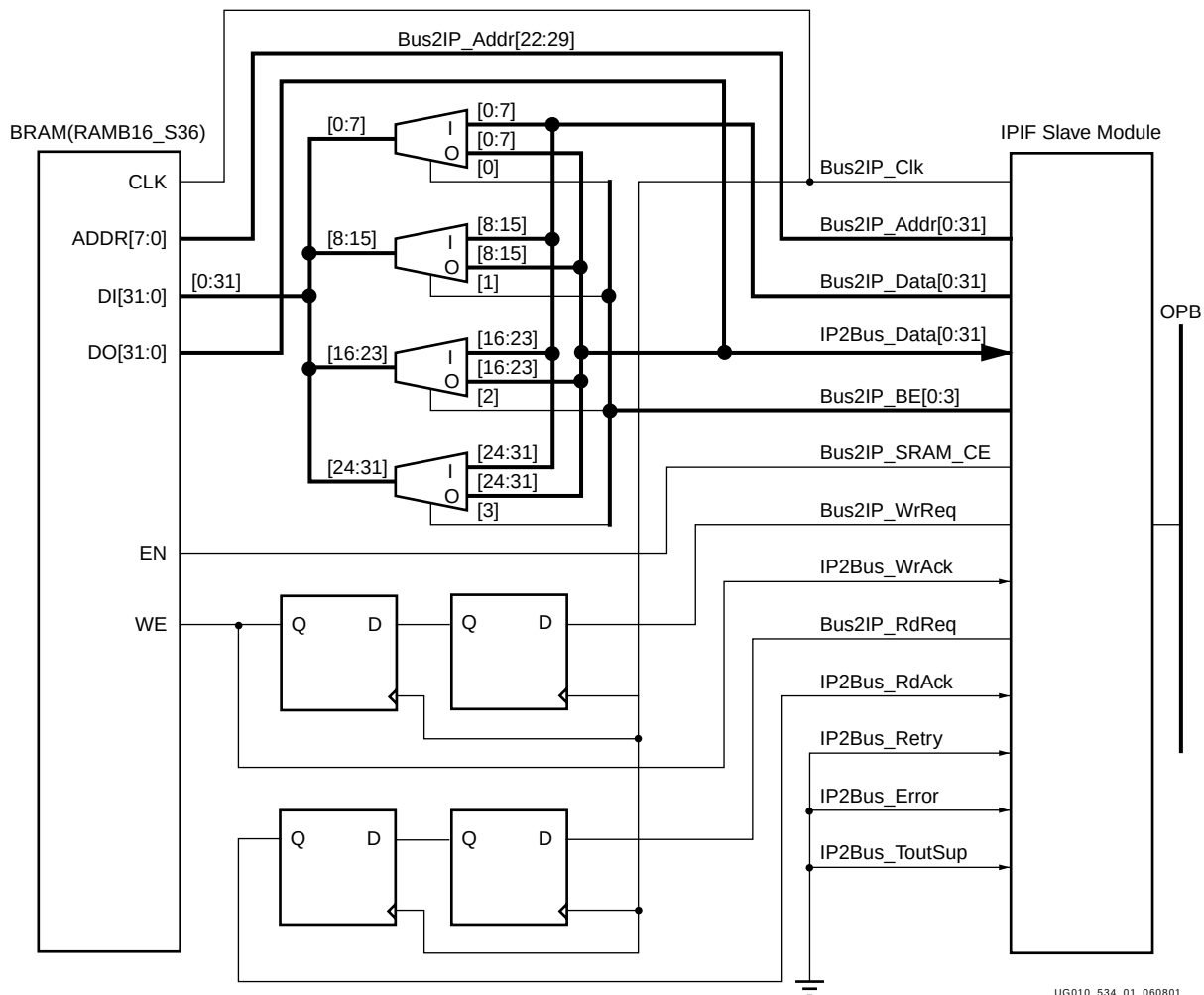
| Name        | Direction | Description                                                         |
|-------------|-----------|---------------------------------------------------------------------|
| C_BASE_ADDR | N/A       | 32-bit base address of CROM<br>(must be aligned to a 2 kB boundary) |

## Implementation

The CROM module contains a minimal amount of glue logic to connect a BRAM to the 32-bit IPIF Slave SRAM module. It demonstrates how quickly and easily an OPB peripheral device can be developed using IPIF modules.

Figure 5-1, page 1403 shows a conceptual view of the logic necessary to connect the BRAM to the IPIF module. The IP2Bus\_RdAck / IP2Bus\_WrAck signals are generated with a fixed, 2-cycle delay after the corresponding Bus2IP\_RdReq / Bus2IP\_WrReq signals are asserted. The first cycle drives the address information to the BRAM. The data from BRAM is available in the second cycle. The data read from BRAM can be sent directly back to the IPIF module to complete read transactions.

Write transactions use a read-modify-write type of operation to handle write byte enables since the BRAM has only one write enable pin. Each byte lane of data read from BRAM is multiplexed with write data from the IPIF module according to the byte enable bits. The resulting data can then be written to memory to complete write transactions with byte enables. To save power, the BRAM is enabled using the Bus2IP\_SRAM\_CE pin so it can be powered down when there is no activity.



UG010\_534\_01\_060801

Figure 5-1: Schematic of CROM Device Implemented with IPIF Module

## Design Extensions and Improvements

The CROM controller can easily be modified to support different memory sizes by adding and reorganizing the memory. This example supports write operations, but the designer could disable the write signal to the BRAM to build a true read-only memory. The unused memory port of the BRAM can also be connected to user logic to make use of the dual-port capability of BRAMs. The contents of the BRAM can be initialized for simulation or downloaded with the help of the Data2BRAM program.



## OPB GPIO Module

---

### Overview

The General Purpose Input/Output (GPIO) module is a design example of an On-Chip Peripheral Bus (OPB) slave device that uses the Xilinx Intellectual Property InterFace (IPIF) module. The GPIO provides register-based control over 32 external pins. Each pin can be either read from or actively driven.

### Related Documents

The following documents provide additional information

- *IPIF Specification*
- *IBM CoreConnect™ 64-Bit On-Chip Peripheral Bus: Architecture Specifications, Version 2.1*
- *Virtex-II Pro™ Platform FPGAs (Advance Product Specification)*

### Features

- 32-bit OPB slave with the ability to use a 32-bit IPIF Slave SRAM module or an IPIF Slave Control Register module
- Permits simple, full software control over 32 external pins
- Supports reads and writes with byte enables

### Module Port Interface

Table 6-1: Global Signals

| Name    | Direction | Description      |
|---------|-----------|------------------|
| OPB_Clk | Input     | OPB system clock |
| OPB_Rst | Input     | OPB system reset |

Table 6-2: OPB Slave Signals

| Name           | Direction | Description                |
|----------------|-----------|----------------------------|
| OPB_ABus[0:31] | Input     | OPB address bus            |
| OPB_BE[0:3]    | Input     | OPB byte enables           |
| OPB_DBus[0:31] | Input     | OPB data bus               |
| OPB_RNW        | Input     | OPB read not write         |
| OPB_select     | Input     | OPB select                 |
| OPB_seqAddr    | Input     | OPB sequential address     |
| S1_DBus[0:31]  | Output    | Slave data bus             |
| S1_DBusEn      | Output    | Slave data bus enable      |
| S1_errAck      | Output    | Slave error acknowledge    |
| S1_retry       | Output    | Slave bus cycle retry      |
| S1_toutSup     | Output    | Slave timeout suppress     |
| S1_xferAck     | Output    | Slave transfer acknowledge |

Table 6-3: External I/O Pins

| Name         | Direction | Description                |
|--------------|-----------|----------------------------|
| GPIO_I[0:31] | Input     | Input from Interface pins  |
| GPIO_O[0:31] | Output    | Output to Interface pins   |
| GPIO_T[0:31] | Output    | 3-state for Interface pins |

Table 6-4: Parameters

| Name        | Direction | Description                                                         |
|-------------|-----------|---------------------------------------------------------------------|
| C_BASE_ADDR | N/A       | 32-bit base address of GPIO (must be aligned to an 8-byte boundary) |

## Implementation

The GPIO module has two 32-bit registers: one register to control the TBUF for each pin and another register to access the I/O pins. The GPIO contains a minimal amount of logic when used with a 32-Bit IPIF Slave SRAM module or with an IPIF Slave Control Register module. It demonstrates how quickly and easily an OPB peripheral device can be developed using IPIF modules.

Figure 6-1 shows a conceptual view of the logic necessary to build the GPIO module using the IPIF Slave SRAM module. Figure 6-2, page 1408, shows an example using the IPIF Slave Control Register module which makes the design even simpler. The IP2Bus\_RdAck / IP2Bus\_WrAck signals are directly connected to the corresponding Bus2IP\_RdReq / Bus2IP\_WrReq signals since it only takes one clock cycle to read or write the GPIO registers. The GPIO registers support byte enables during writes.

To drive an external I/O pin, the output enable for that pin is asserted, allowing the pin to be driven high or low. If the output enable for a given pin is deasserted, the pin is put in a

high impedance state that allows an external device to drive the pin. The CPU can then read the pin to see what value was detected. Note that the designer should ensure that bus contention with an external device does not take place or that sufficient protection is added to prevent damage.

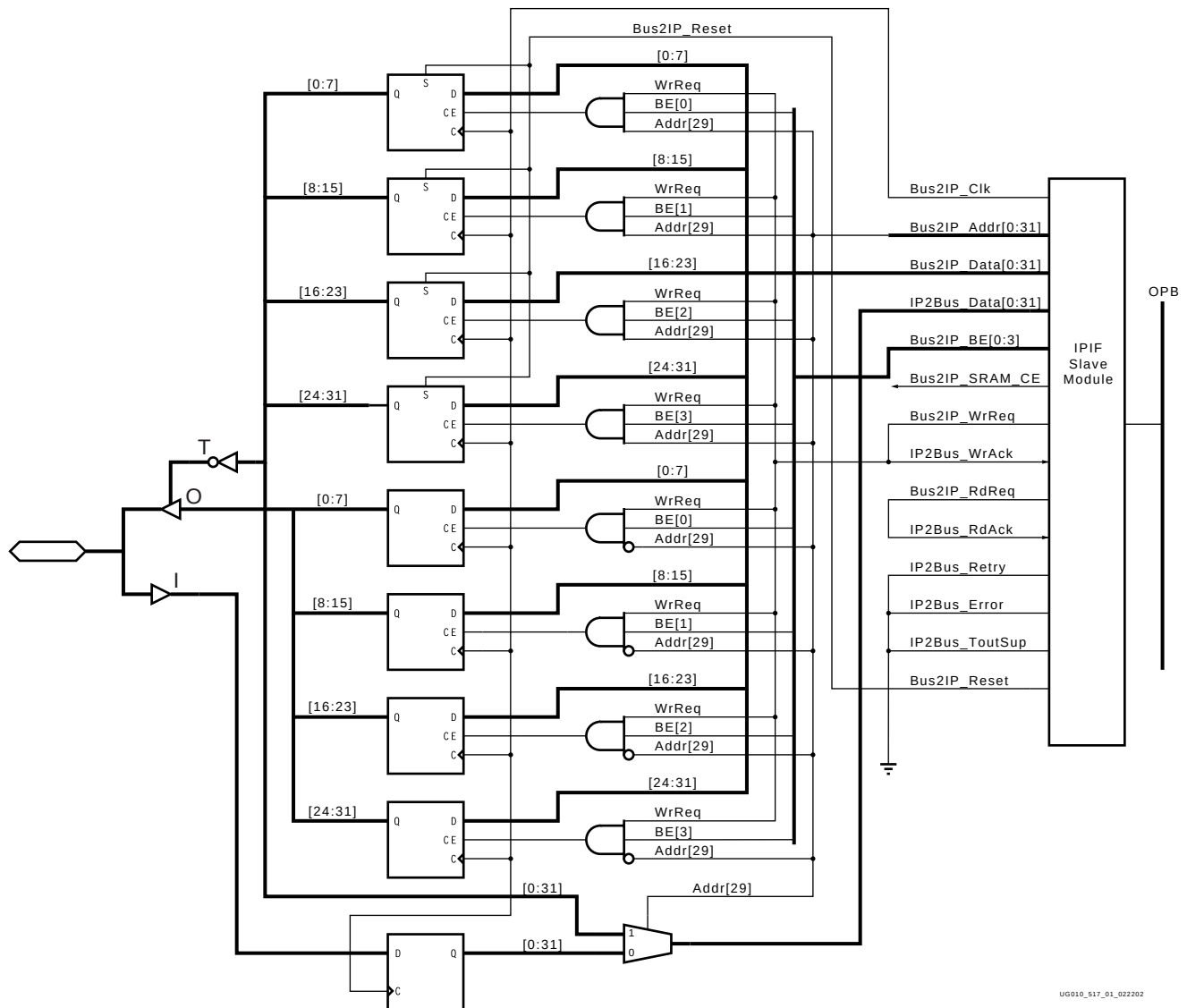


Figure 6-1: Schematic of GPIO Module Using the IPIF Slave SRAM Module



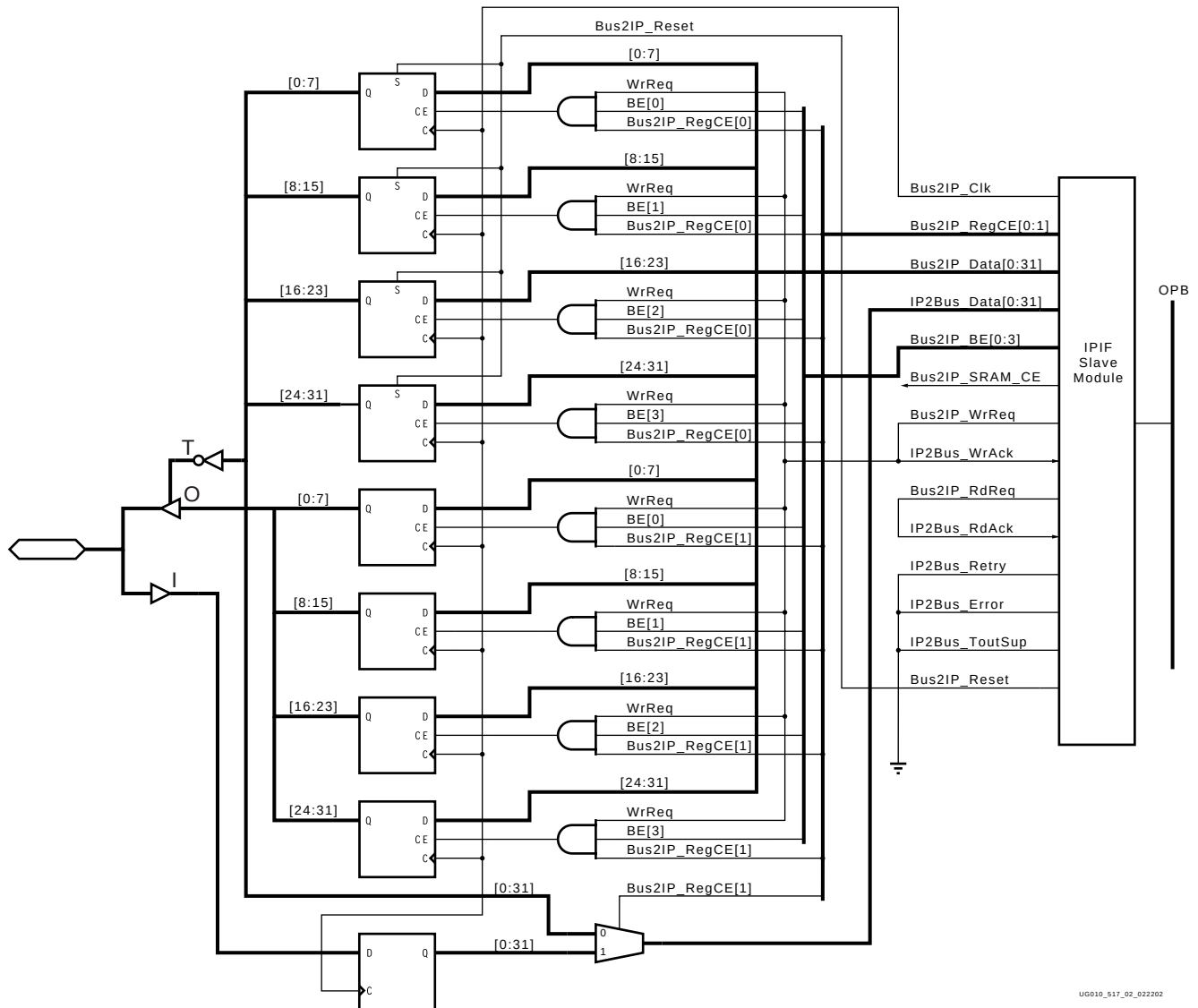


Figure 6-2: Schematic of GPIO Module Using the IPIF Slave Control Register Module

## Memory Map

Table 6-5: Register Definitions

| Register Address | Size           | Description                                                                                                                                                                                                                       |
|------------------|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Base Address + 0 | 32 Bits [0:31] | Wr: Logic value to be driven externally (corresponding output enable must be set high)<br>Rd: Value being driven out (if corresponding output enable is high)<br>Or value sensed/received (if corresponding output enable is low) |
| Base Address + 4 | 32 Bits [0:31] | Rd/Wr: Output Enables for all 32 external pins<br>(1 = Pin is actively driven out; 0 = High Impedance)                                                                                                                            |



## OPB LCD Controller

### Overview

The OPB LCD Controller provides a simple memory-mapped interface to access the registers in a generic character Liquid Crystal Display (LCD) module. The design consists of some control logic connected to an OPB-based Intellectual Property Interface (IPIF) module. The IPIF abstracts the OPB slave interface into a simple SRAM protocol interface that is easier to design with.

### Related Documents

- *IBM CoreConnect™ 64-Bit On-Chip Peripheral Bus:Architecture Specifications*
- *Virtex-II Pro™ Platform FPGAs (Advance Product Specification)*
- *IPIF Specification*

### Features

- 32-bit OPB slave utilizing a 32-bit IPIF Slave SRAM interface
- Implements 8-bit read/write interface found in many character LCD modules

### Module Port Interfaces

Table 7-1: Global Signals

| Name    | Direction | Description      |
|---------|-----------|------------------|
| OPB_Clk | Input     | OPB system clock |
| OPB_Rst | Input     | OPB system reset |

Table 7-2: OPB Slave Signals

| Name           | Direction | Description      |
|----------------|-----------|------------------|
| OPB_ABus[0:31] | Input     | OPB address bus  |
| OPB_BE[0:3]    | Input     | OPB byte enables |
| OPB_DBus[0:31] | Input     | OPB data bus     |

Table 7-2: OPB Slave Signals (Continued)

| Name          | Direction | Description                |
|---------------|-----------|----------------------------|
| OPB_RNW       | Input     | OPB read not write         |
| OPB_select    | Input     | OPB select                 |
| OPB_seqAddr   | Input     | OPB sequential address     |
| S1_DBus[0:31] | Output    | Slave data bus             |
| S1_DBusEn     | Output    | Slave data bus enable      |
| S1_errAck     | Output    | Slave error acknowledge    |
| S1_retry      | Output    | Slave bus cycle retry      |
| S1_toutSup    | Output    | Slave timeout suppress     |
| S1_xferAck    | Output    | Slave transfer acknowledge |

Table 7-3: External I/O Pins

| Name          | Direction | Description         |
|---------------|-----------|---------------------|
| LCD_E         | Output    | Enable Pin          |
| LCD_Rd_Wr_n   | Output    | Read/Write Strobe   |
| LCD_RS        | Output    | Register Select     |
| LCD_DB_I[7:0] | Input     | Data Input          |
| LCD_DB_O[7:0] | Output    | Data Output         |
| LCD_DB_T[7:0] | Output    | Data 3-State Enable |

Table 7-4: Parameters

| Name        | Direction | Description                                                                   |
|-------------|-----------|-------------------------------------------------------------------------------|
| C_BASE_ADDR | N/A       | 32-bit base address of LCD Controller (must be aligned to an 8-byte boundary) |

## Implementation

### Design Overview

Figure 7-1, page 1413 shows a conceptual overview of the design of the LCD Controller. The control logic consists of a counter-based state machine. When an OPB transaction is received, the counter begins counting up toward a fixed stopping point. As the counter reaches various values, the LCD interface pins are toggled as necessary to generate a read or write command to the LCD's registers. When the counter reaches its stopping value, the read or write acknowledge is returned to the IPIF to end the transaction. Given the slow speed of an LCD read/write cycle relative to the OPB clock frequency, the **IP2Bus\_ToutSup** pin is asserted to prevent timeouts from occurring over OPB.

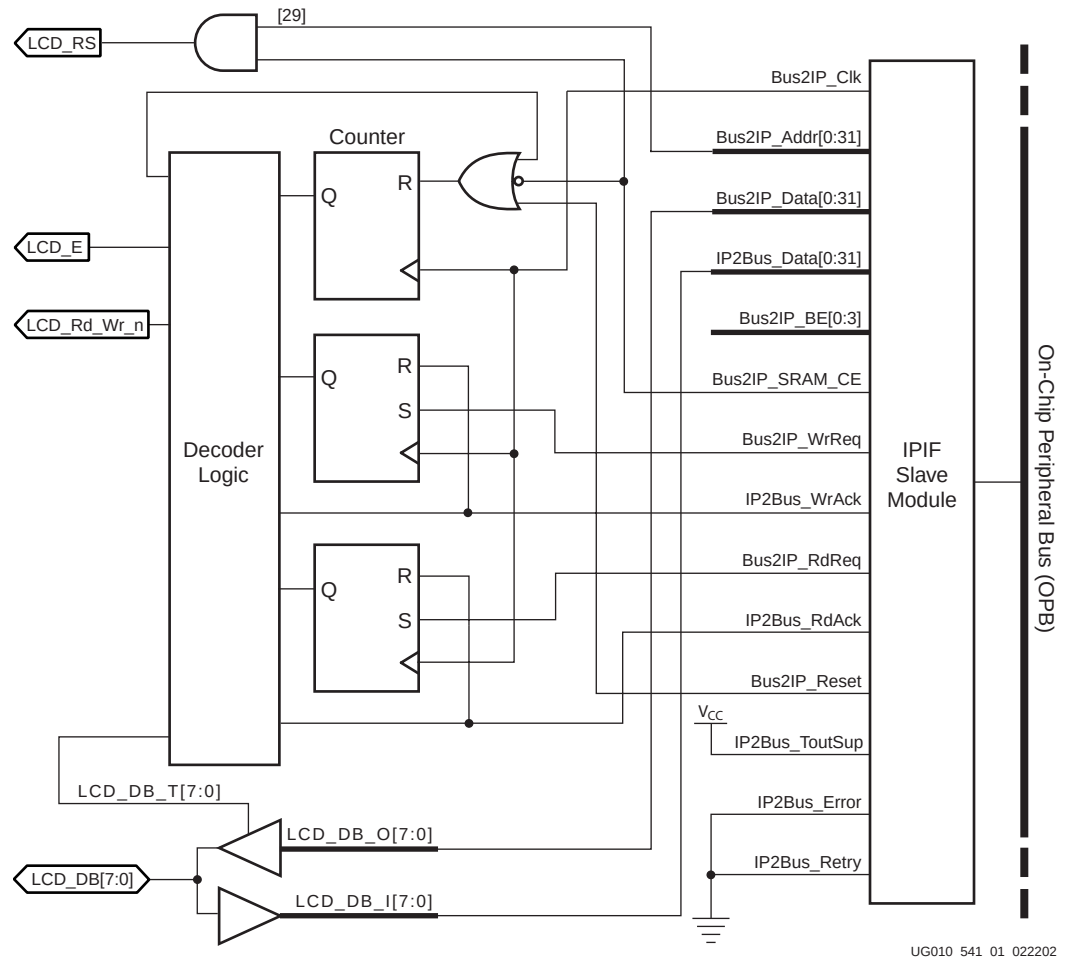


Figure 7-1: Schematic of LCD Controller Using the IPIF Slave SRAM Module

## Read/Write Timing

The timing of the LCD's control signals is taken from an Optrex character LCD module (<http://www.optrex.com>) and assumes the OPB clock frequency is less than 133 MHz. The timing parameters used here should be similar to those of other manufacturers, but the user should refer to the appropriate product datasheets for exact timing specifications. Some signal timing values may need to be adjusted to reflect the specific OPB clock frequency and LCD manufacturer, but it is easy to do so with the counter-based design.

The figures following show the timing for a read (Figure 7-2) and write (Figure 7-3) operation, with the timing expressed in units of OPB clock cycles. The user should refer to the LCD controller datasheets for a description of the software commands to initialize, configure, and display characters on the LCD screen.

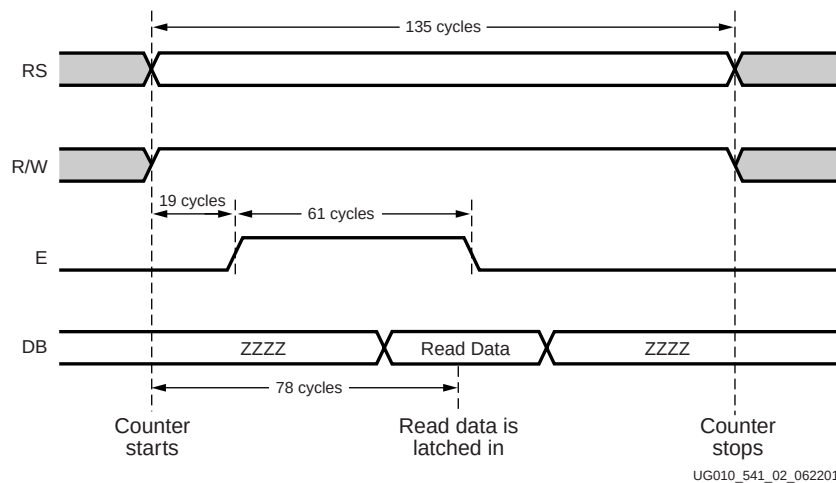


Figure 7-2: Read Cycle Timing Diagram

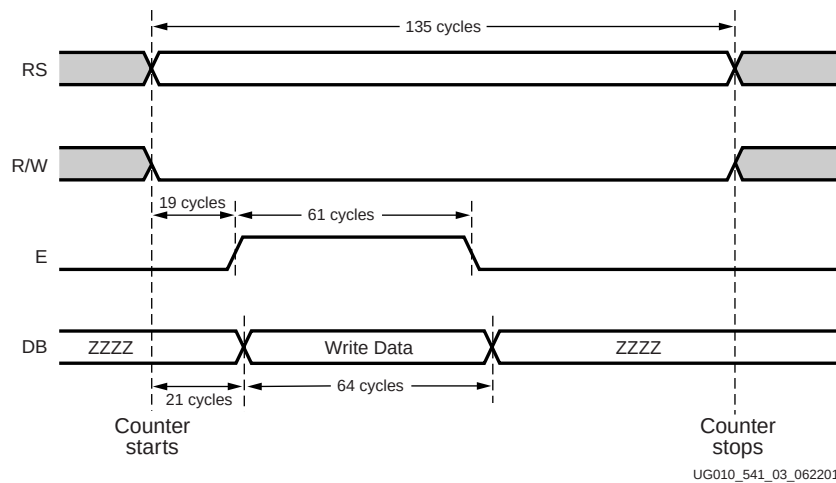


Figure 7-3: Write Cycle Timing Diagram

## Memory Map

The LCD Controller has an 8-bit data bus on its external interface. To make the implementation less complex, however, these memory-mapped registers are aligned to 32-bit address boundaries over OPB. The OPB base address can be programmed to any 8-byte-aligned address by setting the C\_BASE\_ADDR parameter of the module.

Table 7-5: Register Definitions

| Register Address | Size         | Description                                                                                                                           |
|------------------|--------------|---------------------------------------------------------------------------------------------------------------------------------------|
| Base Address + 0 | 8 bits [0:7] | <b>Rd:</b> Reads the LCD control register contents with RS set to 0<br><b>Wr:</b> Writes to the LCD control register with RS set to 0 |
| Base Address + 4 | 8 bits [0:7] | <b>Rd:</b> Reads the LCD control register contents with RS set to 1<br><b>Wr:</b> Writes to the LCD control register with RS set to 1 |

**Notes:**

1. The unused bits [8:31] are undefined for reads, and are ignored for writes





# OPB 16450 UART Controller

---

## Overview

This document provides the design specification for the Virtex-II Pro™ X16450 Universal Asynchronous Receiver/Transmitter (UART) Intellectual Property (IP) module.

The X16450 UART Controller is designed to incorporate the 16450-mode features described in National Semiconductor PC16550D UART data sheet (June, 1995). The PC16550D data sheet is referenced throughout this document and should be used as the authoritative specification. Differences between the PC16550D (in 16450 mode) and the X16450 UART Controller design implementation are highlighted and explained in **Specification Exceptions, page 1426**.

## Related Documents

- *Intellectual Property InterFace (IPIF) Specification*
- *IBM CoreConnect™ 64-Bit On-Chip Peripheral Bus: Architecture Specifications*
- *National Semiconductor PC16550D UART data sheet (June, 1995),*  
<http://www.national.com/pf/PC/PC16550D.html>

## Features

The X16450 UART Controller is a soft IP core designed for Xilinx FPGAs and contains the following features:

- Hardware and software register-compatible with standard 16450 UARTs
- Implements all standard serial interface protocols
  - 5, 6, 7, or 8 bits per character
  - Odd, even, or no parity detection and generation
  - 1, 1.5, or 2 stop bit detection and generation
  - Internal baud rate generator
  - Modem control functions
  - False start bit detection and recovery
  - Prioritized transmit, receive, line status, and modem control interrupts
  - Line break detection and generation
  - Internal loopback diagnostic functionality
- Registers
  - Receiver Buffer Register (read only)
  - Transmitter Holding Register (write only)
  - Interrupt Enable Register

- Interrupt Identification Register (read only)
- Line Control Register
- Modem Control Register
- Line Status Register
- Modem Status Register
- Scratch Register
- Divisor Latch (least significant byte)
- Divisor Latch (most significant byte)
- 100 MHz system clock

## UART Background

The X16450 UART Controller implements the hardware and software functionality of the National Semiconductor 16550 UART, operating in the 16450 mode.

The X16450 UART Controller performs parallel-to-serial conversion on characters received from the CPU and serial-to-parallel conversion on characters received from a modem or microprocessor peripheral. It is capable of transmitting and receiving 5-, 6-, 7-, or 8-bit characters with 1, 1.5, or 2 stop bits and odd, even, or no parity. The X16450 UART Controller can transmit and receive independently. The device can be configured and its status monitored by way of the internal register set. The X16450 UART Controller is capable of signaling receiver, transmitter, and modem control interrupts. These interrupts are prioritized. They can be masked and can be identified by reading an internal register.

## UART I/O Signals

The I/O signals for the UART are listed in [Table 8-1](#). The interfaces referenced in this table are shown in [Table 8-1](#) in the UART block diagram.

**Table 8-1: UART I/O Signals**

| Grouping          |     | Signal Name    | Interface | I/O | Initial State | Description                               |
|-------------------|-----|----------------|-----------|-----|---------------|-------------------------------------------|
| OPB Slave Signals | P1  | opb_abus[0:31] | IPIF      | I   |               | OPB Address Bus                           |
|                   | P2  | opb_be[0:3]    | IPIF      | I   |               | OPB Byte Enable                           |
|                   | P3  | opb_dbus[0:31] | IPIF      | I   |               | OPB Data Bus                              |
|                   | P4  | opb_rnw        | IPIF      | I   |               | Read Not Write                            |
|                   | P5  | opb_Select     | IPIF      | I   |               | OPB Select                                |
|                   | P6  | opb_seqaddr    | IPIF      | I   |               | OPB sequential address (unused)           |
|                   | P7  | sln_dbus[0:31] | IPIF      | O   | 0             | Output Data Bus                           |
|                   | P8  | sln_errack     | IPIF      | O   | 0             | Slave Error Acknowledge (always inactive) |
|                   | P9  | sln_retry      | IPIF      | O   | 0             | Slave Bus Cycle Retry (always inactive)   |
|                   | P10 | Sln_ToutSup    | IPIF      | O   | 0             | Slave Time Out Suppress (always inactive) |
|                   | P11 | Sln_XferAck    | IPIF      | O   | 0             | Slave Transfer Acknowledge                |
|                   | P12 | ip2intc_irpt   | IPIF      | O   | 0             | IPIF interrupt Output                     |

Table 8-1: UART I/O Signals (Continued)

| Grouping     |     | Signal Name | Interface | I/O | Initial State | Description                                                                           |
|--------------|-----|-------------|-----------|-----|---------------|---------------------------------------------------------------------------------------|
| UART Signals | P13 | baudoutn    | Serial    | O   | 1             | Transmitter Clock                                                                     |
|              | P14 | rclk        | Serial    | I   |               | Receiver 16x Clock (unused, internally connected to <b>baudoutn</b> )                 |
|              | P15 | sin         | Serial    | I   |               | Serial Data Input                                                                     |
|              | P16 | sout        | Serial    | O   | 1             | Serial Data Output                                                                    |
|              | P17 | xin         | Serial    | I   |               | Baud Rate Generator reference clock (unused, internally connected to <b>opb_clk</b> ) |
|              | P18 | xout        | Serial    | O   | ~XIN          | Inverted XIN                                                                          |
|              | P19 | ctsn        | Modem     | I   |               | ClearToSend (active low)                                                              |
|              | P20 | dcdn        | Modem     | I   |               | Data Carrier Detect (active low)                                                      |
|              | P21 | dsrn        | Modem     | I   |               | Data Set Ready (active low)                                                           |
|              | P22 | dtrn        | Modem     | O   | 1             | Data Terminal Ready (active low)                                                      |
|              | P23 | rin         | Modem     | I   |               | Ring Indicator (active low)                                                           |
|              | P24 | rtsn        | Modem     | O   | 1             | Request To Send (active low)                                                          |
|              | P25 | ddis        | User      | O   | 1             | Driver Disable. Low when CPU is reading OPB UART                                      |
|              | P26 | out1n       | User      | O   | 1             | User controlled output                                                                |
|              | P27 | out2n       | User      | O   | 1             | User controlled output                                                                |
|              | P28 | rxrdyn      | User      | O   | 1             | DMA control signal                                                                    |
|              | P29 | txrdyn      | User      | O   | 1             | DMA control signal                                                                    |
| System       | P30 | opb_clk     | System    | I   |               | System clock                                                                          |
|              | P31 | opb_rst     | System    | I   |               | System Reset (active high)                                                            |
|              | P32 | freeze      | System    | I   |               | Freezes UART for software debug (active high)                                         |

## UART Register Definition

### UART Interface (IPIF)

The internal registers of the OPB 16450 UART are offset from the C\_BASEADDR base address. Additionally, some of the internal registers are accessible only when bit 7 of the Line Control Register (LCR) is set. The UART internal register set is described in [Table 8-2](#).

Table 8-2: UART Registers

| Name                                    | LCR[7] + Address + C_UART_BAR <sup>(1)</sup> | Access     |
|-----------------------------------------|----------------------------------------------|------------|
| Receiver Buffer Register (RBR)          | 0 + 00011                                    | Read       |
| Transmitter Holding Register (THR)      | 0 + 00011                                    | Write      |
| Interrupt Enable Register (IER)         | 0 + 00111                                    | Read/Write |
| Interrupt Identification Register (IIR) | 0 + 01011                                    | Read       |
| Line Control Register (LCR)             | X + 01111                                    | Read/Write |
| Modem Control Register (MCR)            | X + 10011                                    | Read/Write |
| Line Status Register (LSR)              | X + 10111                                    | Read/Write |
| Modem Status Register (MSR)             | X + 11011                                    | Read/Write |
| Scratch Register (SCR)                  | X + 11111                                    | Read/Write |
| Divisor Register (LS)                   | 1 + 00011                                    | Read/Write |
| Divisor Register (LM)                   | 1 + 00111                                    | Read/Write |

**Notes:**

1. X denotes "don't care"

The data transfer protocol from the OPB to the rest of the circuitry is shown in [Figure 8-1](#).

### UART Register Logic

#### Receiver Buffer Register

The Receiver Buffer Register contains the last received character, as shown in [Table 8-3](#).

Table 8-3: Receiver Buffer Register Bit Definitions

| Bit Location | Name | Access | Reset Value | Description             |
|--------------|------|--------|-------------|-------------------------|
| [7:0]        | RBR  | Read   | 00000000    | Last received character |

## Transmitter Buffer Register

The Transmitter Holding Register contains the character to be transmitted next, as shown in [Table 8-4](#).

**Table 8-4: Transmitter Holding Register Bit Definitions**

| Bit Location | Name | Access | Reset Value | Description                                |
|--------------|------|--------|-------------|--------------------------------------------|
| [7:0]        | THR  | Write  | 11111111    | Holds the character to be transmitted next |

## Interrupt Enable Register

The Interrupt Enable Register contains the bits which enable interrupts, as shown in [Table 8-5](#).

**Table 8-5: Interrupt Enable Register Bit Definitions**

| Bit Location | Name  | Access     | Reset Value <sup>(1)</sup> | Description                                                                                                                                                                 |
|--------------|-------|------------|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [7:4]        |       | Read/Write | <b>0000</b>                | Always returns 0000.                                                                                                                                                        |
| [3]          | EDSSI | Read/Write | 0                          | Enable Modem Status Interrupt.<br>0 → Disables Modem Status Interrupts.<br>1 → Enables Modem Status Interrupts.                                                             |
| [2]          | ELSI  | Read/Write | 0                          | Enable Receiver Line Status Interrupt.<br>0 → Disables Receiver Line Status Interrupts.<br>1 → Enables Receiver Line Status Interrupts.                                     |
| [1]          | ETBEI | Read/Write | 0                          | Enable Transmitter Holding Register Empty Interrupt.<br>0 → Disables Transmitter Holding Register Empty Interrupts.<br>1 → Enables Transmitter Holding Register Interrupts. |
| [0]          | ERBFI | Read/Write | 0                          | Enable Received Data Available Interrupt.<br>0 → Disables Received Data Available Interrupts.<br>1 → Enables Received Data Available Interrupts.                            |

**Notes:**

1. **Boldfaced** bits are permanently low. Writing to these bits is allowed. Reading always returns "0".

## Interrupt Identification Register

The Interrupt Identification Register contains the priority interrupt identification, as shown in [Table 8-6](#).

**Table 8-6: Interrupt Identification Register Bit Definitions**

| Bit Location | Name    | Access | Reset Value <sup>(1)</sup> | Description                                                                                                                                                                                                                                |
|--------------|---------|--------|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [7:4]        |         | Read   | <b>00000</b>               | Always returns 00000.                                                                                                                                                                                                                      |
| [3:1]        | INTID2  | Read   | 000                        | Interrupt ID. <sup>(2)</sup><br>011 → Receiver Line Status (highest).<br>010 → Received Data Available (second).<br>110 → Character Timeout (second).<br>001 → Transmitter Holding Register Empty (third).<br>000 → Modem Status (fourth). |
| [0]          | INTPEND | Read   | 1                          | Interrupt Pending.<br>Interrupt is pending when cleared.                                                                                                                                                                                   |

**Notes:**

- Boldfaced** bits are permanently low. Reading these bits always returns "0".
- If bit 0 is cleared. See National Semiconductor PC16550D data sheet for more detail.

## Line Control Register

The Line Control Register contains the serial communication configuration bits, as shown in [Table 8-7](#).

**Table 8-7: Line Control Register Bit Definitions**

| Bit Location | Name         | Access     | Reset Value | Description                                                                    |
|--------------|--------------|------------|-------------|--------------------------------------------------------------------------------|
| [7]          | DLAB         | Read/Write | 0           | Divisor Latch Access Bit.<br>1 → Allows access to the Divisor Latch Registers. |
| [6]          | Set Break    | Read/Write | 0           | Set Break.<br>1 → Sets SOUT to "0".                                            |
| [5]          | Stick Parity | Read/Write | 0           | Stick Parity.<br>1 → Forces parity to "1" or "0" based on bits 3 and 4.        |
| [4]          | EPS          | Read/Write | 0           | Even Parity Select.<br>1 → Selects Even parity.<br>0 → Selects Odd parity.     |
| [3]          | PEN          | Read/Write | 0           | Parity Enable.<br>1 → Enables parity.                                          |

Table 8-7: Line Control Register Bit Definitions (Continued)

| Bit Location | Name | Access     | Reset Value | Description                                                                                                                 |
|--------------|------|------------|-------------|-----------------------------------------------------------------------------------------------------------------------------|
| [2]          | STB  | Read/Write | 0           | Number of Stop Bits.<br>0 → 1 stop bit.<br>1 → 2 stop bits, or 1.5 if five bits/character selected.                         |
| [1:0]        | WLS  | Read/Write | 00          | Word Length Select.<br>00 → 5 bits/character.<br>01 → 6 bits/character.<br>10 → 7 bits/character.<br>11 → 8 bits/character. |

## Modem Control Register

The Modem Control Register, shown in [Table 8-8](#), contains the modem signaling configuration bits.

Table 8-8: Modem Control Register Bit Definitions

| Bit Location | Name | Access     | Reset Value <sup>(1)</sup> | Description                                                           |
|--------------|------|------------|----------------------------|-----------------------------------------------------------------------|
| [7:5]        |      | Read/Write | <b>000</b>                 | Always returns "000".                                                 |
| [4]          | Loop | Read/Write | 0                          | Loop Back.<br>1 → Enables loopback.                                   |
| [3]          | Out2 | Read/Write | 0                          | User Output 2.<br>1 → Drives OUT2N low.<br>0 → Drives OUT2N high.     |
| [2]          | Out1 | Read/Write | 0                          | User Output 1.<br>1 → Drives OUT1N low.<br>0 → Drives OUT1N high.     |
| [1]          | RTS  | Read/Write | 0                          | Request To Send.<br>1 → Drives RTSN low.<br>0 → Drives RTSN high.     |
| [0]          | DTR  | Read/Write | 0                          | Data Terminal Ready.<br>1 → Drives DTRN low.<br>0 → Drives DTRN high. |

### Notes:

1. **Boldfaced** bits are permanently low. Writing to these bits is allowed. Reading always returns "0".



## Line Status Register

The Line Status Register, shown in [Table 8-9](#), contains the current status of the receiver and transmitter.

**Table 8-9: Line Status Register Bit Definitions**

| Bit Location | Name | Access     | Reset Value | Description                                                                                                  |
|--------------|------|------------|-------------|--------------------------------------------------------------------------------------------------------------|
| [7]          |      | Read/Write | 0           | Always returns "0".                                                                                          |
| [6]          | TEMT | Read/Write | 0           | Transmitter Empty.                                                                                           |
| [5]          | THRE | Read/Write | 1           | Transmitter Holding Register Empty.                                                                          |
| [4]          | BI   | Read/Write | 1           | Break Interrupt.<br>Set when SIN is held low for an entire character time.                                   |
| [3]          | FE   | Read/Write | 0           | Framing Error.<br>Character missing a stop bit.<br>Receiver resynchronizes with next character, if possible. |
| [2]          | PE   | Read/Write | 0           | Parity Error.                                                                                                |
| [1]          | OE   | Read/Write | 0           | Overrun Error.<br>RBR not read before next character is received.                                            |
| [0]          | DR   | Read/Write | 0           | Data Ready.                                                                                                  |

**Notes:**

1. **Boldfaced** bits are permanently low. Writing to these bits is allowed. Reading always returns "0".

## Modem Status Register

The Modem Status Register, shown in [Table 8-10](#), contains the current state of the Modem interface.

**Table 8-10: Modem Status Register Bit Definitions**

| Bit Location | Name | Access     | Reset Value <sup>(1)</sup> | Description                                       |
|--------------|------|------------|----------------------------|---------------------------------------------------|
| [7]          | DCD  | Read/Write | X                          | Data Carrier Detect.<br>Complement of DCDN input. |
| [6]          | RI   | Read/Write | X                          | Ring Indicator.<br>Complement of RIN input.       |
| [5]          | DSR  | Read/Write | X                          | Data Set Ready.<br>Complement of DSRN input.      |
| [4]          | CTS  | Read/Write | X                          | Clear To Send.<br>Complement of CTSN input.       |

Table 8-10: Modem Status Register Bit Definitions (Continued)

| Bit Location | Name | Access     | Reset Value <sup>(1)</sup> | Description                                                            |
|--------------|------|------------|----------------------------|------------------------------------------------------------------------|
| [3]          | DDCD | Read/Write | 0                          | Delta Data Carrier Detect.<br>Change in DCDN since last MSR read.      |
| [2]          | TERI | Read/Write | 0                          | Trailing Edge Ring Indicator.<br>RIN has changed from a low to a high. |
| [1]          | DDSR | Read/Write | 0                          | Delta Data Set Ready.<br>Change in DSRN since last MSR read.           |
| [0]          | DCTS | Read/Write | 0                          | Delta Clear To Send.<br>Change in CTSN since last MSR read.            |

**Notes:**

1. X represents bit driven by external input.

## Scratch Register

The Scratch Register, shown in Table 8-11, can be used to hold user data.

Table 8-11: Scratch Register Bit Definitions

| Bit Location | Name    | Access     | Reset Value | Description |
|--------------|---------|------------|-------------|-------------|
| [7:0]        | Scratch | Read/Write | 00000000    | Scratch.    |

## Divisor (Least Significant Byte) Register

The Divisor (Least Significant Byte) Register, shown in Table 8-12, holds the least significant byte of the baud rate generator counter.

Table 8-12: Divisor (Least Significant Byte) Register Bit Definitions

| Bit Location | Name | Access     | Reset Value | Description                     |
|--------------|------|------------|-------------|---------------------------------|
| [7:0]        | DLL  | Read/Write | 00000000    | Divisor Least Significant Byte. |

## Divisor (Most Significant Byte) Register

The Divisor (Most Significant Byte) Register, shown in Table 8-13, holds the most significant byte of the baud rate generator counter.

Table 8-13: Divisor (Most Significant Byte) Register Bit Definitions

| Bit Location | Name | Access     | Reset Value | Description                    |
|--------------|------|------------|-------------|--------------------------------|
| [7:0]        | DLM  | Read/Write | 00000000    | Divisor Most Significant Byte. |

## UART Block Diagram

The top-level block diagram for the UART is shown in [Figure 8-1](#).

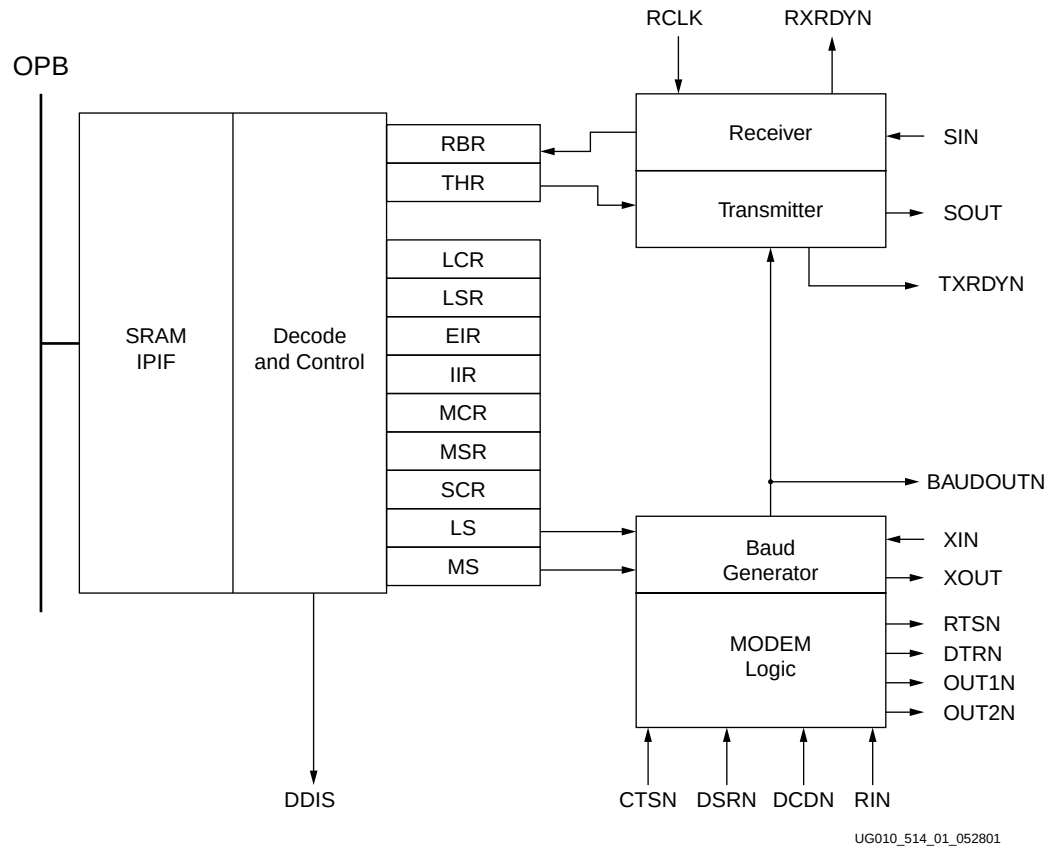


Figure 8-1: UART Top-Level Block Diagram

## Specification Exceptions

### System Clock

The asynchronous microprocessor interface of the National Semiconductor PC16550D is synchronized to the system clock input of the X16450 UART Controller.

### Register Addresses

All internal registers reside on a 32-bit word boundary; not on 8-bit byte boundaries. The Virtex-II Pro software driver for the X16450 UART Controller module takes this into account. Other 16450 UART drivers may not.

# OPB 16550 UART Controller

---

## Overview

This document provides the design specification for the Virtex-II Pro™ X16550 Universal Asynchronous Receiver/Transmitter (UART) Intellectual Property (IP) module.

The X16550 UART Controller incorporates the features, including FIFO mode, described in National Semiconductor PC16550D UART data sheet (June, 1995). The National Semiconductor PC16550D data sheet is referenced throughout this document and should be used as the authoritative specification. Differences between the PC16550D and the X16550 UART Controller design implementation are highlighted and explained in **Specification Exceptions, page 1437**.

## Related Documents

- *Intellectual Property InterFace (IPIF) Specification*
- *IBM CoreConnect™ 64-Bit On-Chip Peripheral Bus: Architecture Specifications*
- *National Semiconductor PC16550D UART data sheet (June, 1995),*  
<http://www.national.com/pf/PC/PC16550D.html>

## Features

The X16550 UART Controller is a soft IP core designed for Xilinx FPGAs and contains the following features:

- Hardware and software register-compatible with all standard 16550 UARTs
- Implements all standard serial interface protocols
  - 5, 6, 7, or 8 bits per character
  - Odd, even or no parity detection and generation
  - 1, 1.5, or 2 stop bit detection and generation
  - Internal baud rate generator
  - Modem control functions
  - False start bit detection and recovery
  - Prioritized transmit, receive, line status, and modem control interrupts
  - Line break detection and generation
  - Internal loopback diagnostic functionality
  - Independent 16 word transmit and receive FIFOs
- Registers
  - Receiver Buffer Register (read only)
  - Transmitter Holding Register (write only)
  - Interrupt Enable Register
  - Interrupt Identification Register (read only)

- FIFO Control Register (read/write) See **Specification Exceptions**
- Line Control Register
- Modem Control Register
- Line Status Register
- Modem Status Register
- Scratch Register
- Divisor Latch (least significant byte)
- Divisor Latch (most significant byte)
- 100 MHz system clock

## UART Background

The X16550 UART Controller implements the hardware and software functionality of the National Semiconductor 16550 UART.

The X16550 UART Controller performs parallel-to-serial conversion on characters received from the CPU and serial-to-parallel conversion on characters received from a modem or microprocessor peripheral. It is capable of transmitting and receiving 5-, 6-, 7- or 8-bit characters with 1, 1.5, or 2 stop bits and odd, even, or no parity. The X16550 UART Controller can transmit and receive independently. The device can be configured and its status monitored by way of the internal register set. It is capable of signaling receiver, transmitter, and modem control interrupts. These interrupts can be masked, are prioritized, and can be identified by reading an internal register. The device contains a 16-bit programmable baud rate generator and independent 16-word transmit and receive FIFOs. The FIFOs can be enabled or disabled through software control.

## UART I/O Signals

The I/O signals for the UART are listed in **Table 9-1**. The interfaces referenced in this table are shown in **Table 9-1** in the UART block diagram.

**Table 9-1: UART I/O Signals**

| Grouping          |     | Signal Name    | Interface | I/O | Initial State | Description                               |
|-------------------|-----|----------------|-----------|-----|---------------|-------------------------------------------|
| OPB Slave Signals | P1  | opb_abus[0:31] | IPIF      | I   |               | OPB Address Bus                           |
|                   | P2  | opb_be[0:3]    | IPIF      | I   |               | OPB Byte Enable                           |
|                   | P3  | opb_dbus[0:31] | IPIF      | I   |               | OPB Data Bus                              |
|                   | P4  | opb_rnw        | IPIF      | I   |               | Read Not Write                            |
|                   | P5  | opb_select     | IPIF      | I   |               | OPB select                                |
|                   | P6  | opb_seqaddr    | IPIF      | I   |               | OPB sequential address (unused)           |
|                   | P7  | sln_dbus[0:31] | IPIF      | O   | 0             | Output Data Bus                           |
|                   | P8  | sln_errack     | IPIF      | O   | 0             | Slave Error Acknowledge (always inactive) |
|                   | P9  | sln_retry      | IPIF      | O   | 0             | Slave Bus Cycle Retry (always inactive)   |
|                   | P10 | sln_toutsup    | IPIF      | O   | 0             | Slave Time Out Suppress (always inactive) |
|                   | P11 | sln_xferack    | IPIF      | O   | 0             | Slave Transfer Acknowledge                |
|                   | P12 | ip2intc_irpt   | IPIF      | O   | 0             | IPIF interrupt Output                     |

Table 9-1: UART I/O Signals (Continued)

| Grouping     |     | Signal Name | Interface | I/O | Initial State | Description                                                                           |
|--------------|-----|-------------|-----------|-----|---------------|---------------------------------------------------------------------------------------|
| UART Signals | P13 | baudoutn    | Serial    | O   | 1             | Transmitter Clock                                                                     |
|              | P14 | rclk        | Serial    | I   |               | Receiver 16x Clock (unused, internally connected to <b>baudoutn</b> )                 |
|              | P15 | sin         | Serial    | I   |               | Serial Data Input                                                                     |
|              | P16 | sout        | Serial    | O   | 1             | Serial Data Output                                                                    |
|              | P17 | xin         | Serial    | I   |               | Baud Rate Generator reference clock (unused, internally connected to <b>opb_clk</b> ) |
|              | P18 | xout        | Serial    | O   | ~XIN          | Inverted XIN                                                                          |
|              | P19 | ctsn        | Modem     | I   |               | ClearToSend (active low)                                                              |
|              | P20 | dcdn        | Modem     | I   |               | Data Carrier Detect (active low)                                                      |
|              | P21 | dsrn        | Modem     | I   |               | Data Set Ready (active low)                                                           |
|              | P22 | dtrn        | Modem     | O   | 1             | Data Terminal Ready (active low)                                                      |
|              | P23 | rin         | Modem     | I   |               | Ring Indicator (active low)                                                           |
|              | P24 | rtsn        | Modem     | O   | 1             | Request To Send (active low)                                                          |
|              | P25 | ddis        | User      | O   | 1             | Driver Disable. Low when CPU is reading OPB UART                                      |
|              | P26 | out1n       | User      | O   | 1             | User controlled output                                                                |
|              | P27 | our2n       | User      | O   | 1             | User controlled output                                                                |
|              | P28 | rxrdyn      | User      | O   | 1             | DMA control signal                                                                    |
|              | P29 | txrdyn      | User      | O   | 1             | DMA control signal                                                                    |
| System       | P30 | opb_clk     | System    | I   |               | System clock                                                                          |
|              | P31 | opb_rst     | System    | I   |               | System Reset (active high)                                                            |
|              | P32 | freeze      | System    | I   |               | Freezes UART for software debug (active high)                                         |

## UART Register Definition

### UART Interface (IPIF)

The internal registers of the OPB 16550 UART are offset from the C\_BASEADDR base address. Additionally, some of the internal registers are accessible only when bit 7 of the Line Control Register (LCR) is set. The UART internal register set is described in [Table 9-2](#).

Table 9-2: UART Registers

| Name                                    | LCR[7] + Address + C_UART_BAR <sup>(1)</sup> | Access     |
|-----------------------------------------|----------------------------------------------|------------|
| Receiver Buffer Register (RBR)          | 0 + 00011                                    | Read       |
| Transmitter Holding Register (THR)      | 0 + 00011                                    | Write      |
| Interrupt Enable Register (IER)         | 0 + 00111                                    | Read/Write |
| Interrupt Identification Register (IIR) | 0 + 01011                                    | Read       |
| FIFO Control Register (FCR)             | 0 + 01011                                    | Write      |
| FIFO Control Register <sup>(2)</sup>    | 1 + 01011                                    | Read       |
| Line Control Register (LCR)             | X + 01111                                    | Read/Write |
| Modem Control Register (MCR)            | X + 10011                                    | Read/Write |
| Line Status Register (LSR)              | X + 10111                                    | Read/Write |
| Modem Status Register (MSR)             | X + 11011                                    | Read/Write |
| Scratch Register (SCR)                  | X + 11111                                    | Read/Write |
| Divisor Register (LS)                   | 1 + 00011                                    | Read/Write |
| Divisor Register (LM)                   | 1 + 00111                                    | Read/Write |

**Notes:**

1. X denotes "don't care"
2. FIFO Control Register is Write Only in the National PC16550D

The data transfer protocol from the OPB to the rest of the circuitry is shown in [Figure 9-1](#).

### UART Register Logic

#### Receiver Buffer Register

The Receiver Buffer Register contains the last received character, as shown in [Table 9-3](#).

Table 9-3: Receiver Buffer Register Bit Definitions

| Bit Location | Name | Access | Reset Value | Description             |
|--------------|------|--------|-------------|-------------------------|
| [7:0]        | RBR  | Read   | 00000000    | Last received character |

## Transmitter Holding Register

The Transmitter Holding Register contains the character to be transmitted next, as shown in [Table 9-4](#).

**Table 9-4: Transmitter Holding Register Bit Definitions**

| Bit Location | Name | Access | Reset Value | Description                                |
|--------------|------|--------|-------------|--------------------------------------------|
| [7:0]        | THR  | Write  | 11111111    | Holds the character to be transmitted next |

## Interrupt Enable Register

The Interrupt Enable Register contains the bits which enable interrupts, as shown in [Table 9-5](#).

**Table 9-5: Interrupt Enable Register Bit Definitions**

| Bit Location | Name  | Access     | Reset Value <sup>(1)</sup> | Description                                                                                                                                                                 |
|--------------|-------|------------|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [7:4]        |       | Read/Write | <b>0000</b>                | Always returns 0000.                                                                                                                                                        |
| [3]          | EDSSI | Read/Write | 0                          | Enable Modem Status Interrupt.<br>0 → Disables Modem Status Interrupts.<br>1 → Enables Modem Status Interrupts.                                                             |
| [2]          | ELSI  | Read/Write | 0                          | Enable Receiver Line Status Interrupt.<br>0 → Disables Receiver Line Status Interrupts.<br>1 → Enables Receiver Line Status Interrupts.                                     |
| [1]          | ETBEI | Read/Write | 0                          | Enable Transmitter Holding Register Empty Interrupt.<br>0 → Disables Transmitter Holding Register Empty Interrupts.<br>1 → Enables Transmitter Holding Register Interrupts. |
| [0]          | ERBFI | Read/Write | 0                          | Enable Received Data Available Interrupt.<br>0 → Disables Received Data Available Interrupts.<br>1 → Enables Received Data Available Interrupts.                            |

**Notes:**

1. **Boldfaced** bits are permanently low. Writing to these bits is allowed. Reading always returns "0".



## Interrupt Identification Register

The Interrupt Identification Register contains the priority interrupt identification, as shown in [Table 9-6](#).

**Table 9-6: Interrupt Identification Register Bit Definitions**

| Bit Location | Name    | Access | Reset Value <sup>(1)</sup> | Description                                                                                                                                                                                                                                |
|--------------|---------|--------|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [7:6]        | FIFOEN  | Read   | 00                         | FIFOs enabled.<br>Always zero if not in FIFO mode.                                                                                                                                                                                         |
| [5:4]        |         | Read   | 00                         | Always returns "00".                                                                                                                                                                                                                       |
| [3:1]        | INTID2  | Read   | 000                        | Interrupt ID. <sup>(2)</sup><br>011 → Receiver Line Status (highest).<br>010 → Received Data Available (second).<br>110 → Character Timeout (second).<br>001 → Transmitter Holding Register Empty (third).<br>000 → Modem Status (fourth). |
| [0]          | INTPEND | Read   | 1                          | Interrupt Pending.<br>Interrupt is pending when cleared.                                                                                                                                                                                   |

**Notes:**

1. **Boldfaced** bits are permanently low. Reading these bits always returns "0".
2. If bit 0 is cleared. See National Semiconductor PC16550D data sheet for more detail.

## FIFO Control Register

The FIFO Control Register contains the FIFO configuration bit, as shown in [Table 9-7](#).

**Table 9-7: FIFO Control Register Bit Definitions**

| Bit Location | Name                    | Access     | Reset Value | Description                                                                                  |
|--------------|-------------------------|------------|-------------|----------------------------------------------------------------------------------------------|
| [7:6]        | RCVR FIFO Trigger Level | Read/Write | 00          | RCVR FIFO Trigger Level.<br>00 → 1 byte.<br>01 → 4 bytes.<br>10 → 8 bytes.<br>11 → 14 bytes. |
| [5:4]        | Reserved                | Read/Write | 00          | Always returns "00"                                                                          |
| [3]          | DMA Mode Select         | Read/Write | 0           | DMA Mode Select.<br>0 → Mode 0.<br>1 → Mode 1.                                               |
| [2]          | XMIT FIFO Reset         | Read/Write | 0           | Transmitter FIFO Reset.<br>1 → Resets XMIT FIFO.                                             |

Table 9-7: FIFO Control Register Bit Definitions (Continued)

| Bit Location | Name               | Access     | Reset Value | Description                                      |
|--------------|--------------------|------------|-------------|--------------------------------------------------|
| [1]          | RCVR<br>FIFO Reset | Read/Write | 0           | Transmitter FIFO Reset.<br>1 → Resets RCVR FIFO. |
| [0]          | FIFOEN             | Read/Write | 0           | FIFO Enable.<br>1 → Enables FIFOs.               |

**Notes:**

1. **Boldfaced** bits are permanently low. Reading these bits always returns "0".

## Line Control Register

As shown in Table 9-8, the Line Control Register contains the serial communication configuration bits.

Table 9-8: Line Control Register Bit Definitions

| Bit Location | Name         | Access     | Reset Value | Description                                                                                                                 |
|--------------|--------------|------------|-------------|-----------------------------------------------------------------------------------------------------------------------------|
| [7]          | DLAB         | Read/Write | 0           | Divisor Latch Access Bit.<br>1 → Allows access to the Divisor Latch Registers and reading of the FIFO Control Register.     |
| [6]          | Set Break    | Read/Write | 0           | Set Break.<br>1 → Sets SOUT to "0".                                                                                         |
| [5]          | Stick Parity | Read/Write | 0           | Stick Parity.<br>1 → Forces parity to "1" or "0" based on bits 3 and 4.                                                     |
| [4]          | EPS          | Read/Write | 0           | Even Parity Select.<br>1 → Selects Even parity.<br>0 → Selects Odd parity.                                                  |
| [3]          | PEN          | Read/Write | 0           | Parity Enable.<br>1 → Enables parity.                                                                                       |
| [2]          | STB          | Read/Write | 0           | Number of Stop Bits.<br>0 → 1 Stop bit.<br>1 → 2 Stop bits, or 1.5 if five bits/character selected.                         |
| [1:0]        | WLS          | Read/Write | 00          | Word Length Select.<br>00 → 5 bits/character.<br>01 → 6 bits/character.<br>10 → 7 bits/character.<br>11 → 8 bits/character. |

## Modem Control Register

The Modem Control Register, shown in [Table 9-9](#), contains the modem signaling configuration bits.

**Table 9-9: Modem Control Register Bit Definitions**

| Bit Location | Name | Access     | Reset Value <sup>(1)</sup> | Description                                                           |
|--------------|------|------------|----------------------------|-----------------------------------------------------------------------|
| [7:5]        |      | Read/Write | <b>000</b>                 | Always returns "000".                                                 |
| [4]          | Loop | Read/Write | 0                          | Loop Back.<br>1 → Enables loopback.                                   |
| [3]          | Out2 | Read/Write | 0                          | User Output 2.<br>1 → Drives OUT2N low.<br>0 → Drives OUT2N high.     |
| [2]          | Out1 | Read/Write | 0                          | User Output 1.<br>1 → Drives OUT1N low.<br>0 → Drives OUT1N high.     |
| [1]          | RTS  | Read/Write | 0                          | Request To Send.<br>1 → Drives RTSN low.<br>0 → Drives RTSN high.     |
| [0]          | DTR  | Read/Write | 0                          | Data Terminal Ready.<br>1 → Drives DTRN low.<br>0 → Drives DTRN high. |

**Notes:**

1. **Boldfaced** bits are permanently low. Writing to these bits is allowed. Reading always returns "0".

## Line Status Register

As shown in [Table 9-10](#), the Line Status Register contains the current status of the receiver and transmitter.

**Table 9-10: Line Status Register Bit Definitions**

| Bit Location | Name               | Access     | Reset Value | Description                                                                |
|--------------|--------------------|------------|-------------|----------------------------------------------------------------------------|
| [7]          | Error in RCVR FIFO | Read/Write | 0           | Error in RCVR FIFO.<br>RCVR FIFO contains at least one receiver error.     |
| [6]          | TEMT               | Read/Write | 0           | Transmitter Empty.                                                         |
| [5]          | THRE               | Read/Write | 1           | Transmitter Holding Register Empty.                                        |
| [4]          | BI                 | Read/Write | 1           | Break Interrupt.<br>Set when SIN is held low for an entire character time. |

Table 9-10: Line Status Register Bit Definitions (Continued)

| Bit Location | Name | Access     | Reset Value | Description                                                                                                  |
|--------------|------|------------|-------------|--------------------------------------------------------------------------------------------------------------|
| [3]          | FE   | Read/Write | 0           | Framing Error.<br>Character missing a stop bit.<br>Receiver resynchronizes with next character, if possible. |
| [2]          | PE   | Read/Write | 0           | Parity Error.                                                                                                |
| [1]          | OE   | Read/Write | 0           | Overrun Error.<br>RBR not read before next character is received.                                            |
| [0]          | DR   | Read/Write | 0           | Data Ready.                                                                                                  |

## Modem Status Register

The Modem Status Register, shown in Table 9-11, contains the current state of the Modem interface.

Table 9-11: Modem Status Register Bit Definitions

| Bit Location | Name | Access     | Reset Value <sup>(1)</sup> | Description                                                            |
|--------------|------|------------|----------------------------|------------------------------------------------------------------------|
| [7]          | DCD  | Read/Write | X                          | Data Carrier Detect.<br>Complement of DCDN input.                      |
| [6]          | RI   | Read/Write | X                          | Ring Indicator.<br>Complement of RIN input.                            |
| [5]          | DSR  | Read/Write | X                          | Data Set Ready.<br>Complement of DSRN input.                           |
| [4]          | CTS  | Read/Write | X                          | Clear To Send.<br>Complement of CTSN input.                            |
| [3]          | DDCD | Read/Write | 0                          | Delta Data Carrier Detect.<br>Change in DCDN since last MSR read.      |
| [2]          | TERI | Read/Write | 0                          | Trailing Edge Ring Indicator.<br>RIN has changed from a low to a high. |
| [1]          | DDSR | Read/Write | 0                          | Delta Data Set Ready.<br>Change in DSRN since last MSR read.           |
| [0]          | DCTS | Read/Write | 0                          | Delta Clear To Send.<br>Change in CTSN since last MSR read.            |

### Notes:

1. X represents bit driven by external input.

## Scratch Register

The Scratch Register, shown in [Table 9-12](#), can be used to hold user data.

**Table 9-12: Scratch Register Bit Definitions**

| Bit Location | Name    | Access     | Reset Value | Description |
|--------------|---------|------------|-------------|-------------|
| [7:0]        | Scratch | Read/Write | 00000000    | Scratch.    |

## Divisor (Least Significant Byte) Register

The Divisor (Least Significant Byte) Register, shown in [Table 9-13](#), holds the least significant byte of the baud rate generator counter.

**Table 9-13: Divisor (Least Significant Byte) Register Bit Definitions**

| Bit Location | Name | Access     | Reset Value | Description                     |
|--------------|------|------------|-------------|---------------------------------|
| [7:0]        | DLL  | Read/Write | 00000000    | Divisor Least Significant Byte. |

## Divisor (Most Significant Byte) Register

The Divisor (Most Significant Byte) Register, shown in [Table 9-14](#), holds the most significant byte of the baud rate generator counter.

**Table 9-14: Divisor (Most Significant Byte) Register Bit Definitions**

| Bit Location | Name | Access     | Reset Value | Description                    |
|--------------|------|------------|-------------|--------------------------------|
| [7:0]        | DLM  | Read/Write | 00000000    | Divisor Most Significant Byte. |

## UART Block Diagram

The top-level block diagram for the UART is shown in [Figure 9-1](#).

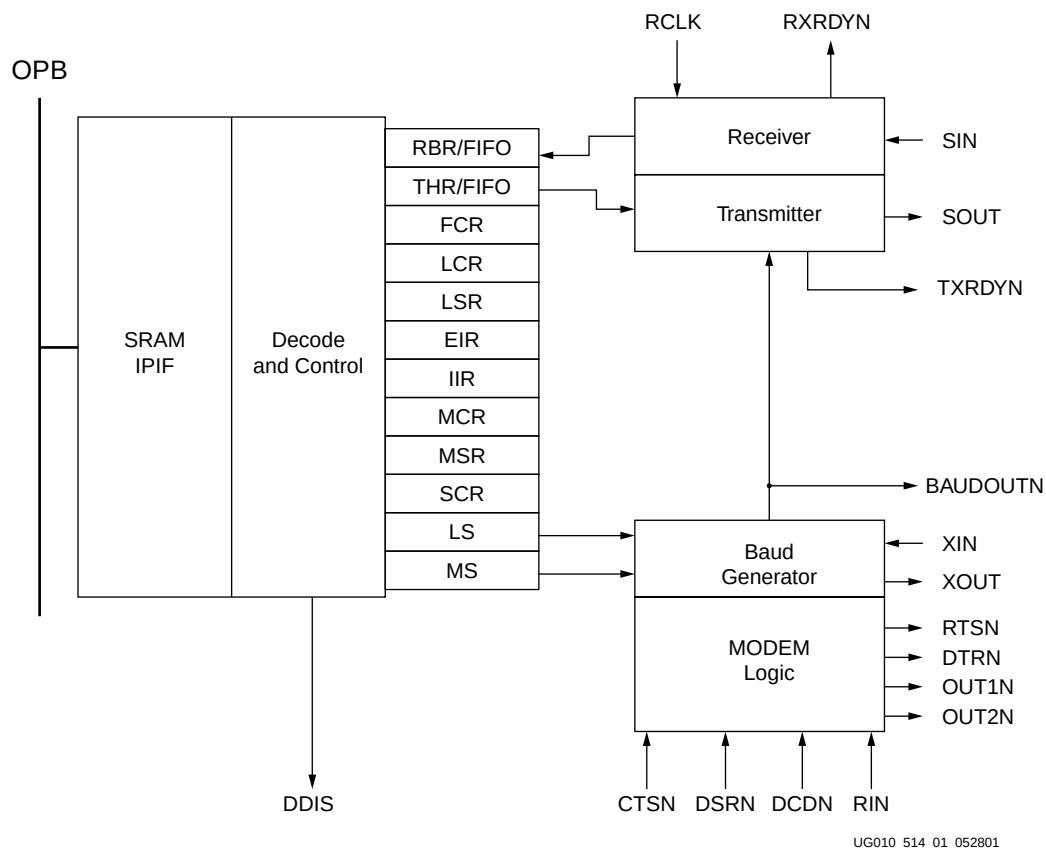


Figure 9-1: UART Top-Level Block Diagram

## Specification Exceptions

### FIFO Control Register

The FIFO control register is a read/write register. Read access is controlled by setting Line Control Register bit 7.

### System Clock

The asynchronous microprocessor interface of the National Semiconductor PC16550D is synchronized to the system clock input of the X16550 UART Controller.

### Register Addresses

All internal registers reside on a 32-bit word boundary; not on 8-bit byte boundaries. The Virtex-II Pro software driver for the X16550 UART Controller takes this into account. Other 16550 UART drivers may not.



# OPB IIC Controller

---

## Important Note Regarding Philips I<sup>2</sup>C-Bus Technology

The I<sup>2</sup>C-Bus technology is the patented intellectual property of Koninklijke Philips N.V. *Xilinx does not and cannot convey to end users any license to use the I<sup>2</sup>C technology.* End users must obtain their own licensing directly from Philips. For more information on obtaining licensing to use I<sup>2</sup>C, go to <http://www.semiconductors.philips.com/buses/i2c/licensing/>.

## Overview

This document provides the design specification for the Virtex-II Pro™ On-Chip Peripheral Bus (OPB) IIC Controller Intellectual Property (IP) module.

This specification defines the architecture and interface requirements of the OPB IIC Controller. This includes registers that the user must initialize for proper operation. This module supports standard mode features, 100 Kb/s, and seven-bit addressing of the Philips I<sup>2</sup>C bus, version 2.1, January 2000 release.

## Related Documents

- *IPIF Specification*
- *Philips Semiconductors I<sup>2</sup>C-Bus Specification*, version 2.1, January 2000
- *IBM CoreConnect™ 64-Bit On-Chip Peripheral Bus: Architecture Specifications*

## Features

The IIC bus interface is a soft IP core designed for Xilinx FPGAs, and contains the following features:

- Master or Slave operation
- Multi-master operation
- Software selectable acknowledge bit
- Arbitration lost interrupt with automatic mode switching from Master to Slave
- Calling address identification interrupt with automatic mode switching from Master to Slave
- START and STOP signal generation/detection
- Repeated START signal generation
- Acknowledge bit generation/detection
- Bus busy detection
- 100 KHz clock (with 100 MHz OPB)
- General Call Enable or Disable
- Transmit and Receive FIFOs - 16 bytes deep
- Throttling



## IIC Background

This section describes the main protocol of the IIC bus. For more details and timing diagrams, see the IIC specification at <http://www.semiconductors.philips.com/buses/i2c/>.

The IIC bus consists of two wires, serial data (SDA) and serial clock (SCL), which carry information between the devices connected to the bus. The number of devices connected to the same bus is limited only by a maximum bus capacitance of 400 pF. Both the SDA and SCL lines are bidirectional lines, connected to a positive supply voltage via a pull-up resistor. When the bus is free, both lines are High. The output stages of devices connected to the bus must have an open-drain or open-collector in order to perform the wired-AND function.

Each device on the bus has a unique address and can operate as either a transmitter or receiver. In addition, devices can also be configured as Masters or Slaves. A Master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. Any other device that is being addressed is considered a Slave. The IIC protocol defines an arbitration procedure that insures that if more than one Master simultaneously tries to control the bus, only one is allowed to do so and the message is not corrupted. The arbitration and clock synchronization procedures defined in the Philips I2C specification are supported by the IIC Bus Interface module.

Data transfers on the IIC bus are initiated with a START condition and are terminated with a STOP condition. Normal data on the SDA line must be stable during the High period of the clock, including required setup and hold times. The High or Low state of the data line can only change when SCL is Low. The START condition is a unique case and is defined by a High-to-Low transition on the SDA line while SCL is High. Likewise, the STOP condition is a unique case and is defined by a Low-to-High transition on the SDA line while SCL is High. The definitions of data, START, and STOP insure that the START and STOP conditions will never be confused as data. This is shown in Figure 10-1.

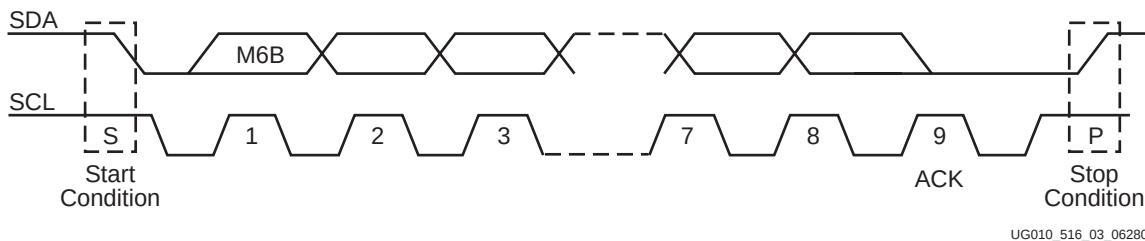


Figure 10-1: Data Transfer on the IIC Bus

Each data packet on the IIC bus consists of eight bits of data followed by an acknowledge bit so one complete data byte transfer requires nine clock pulses. Data is transferred with the most significant bit first (MSB). The transmitter releases the SDA line during the acknowledge bit and the receiver of the data transfer must drive the SDA line low during the acknowledge bit to acknowledge receipt of the data. If a Slave-receiver does not drive the SDA line Low during the acknowledge bit, this indicates that the Slave-receiver was unable to accept the data and the Master can then generate a STOP condition to abort the transfer. If the Master-receiver does not generate an acknowledge, this indicates to the Slave-transmitter that this byte was the last byte of the transfer.

Standard communication on the bus between a Master and a Slave is composed of four parts: START, Slave address, data transfer, and STOP. A seven bit address is initiated as follows. After the START condition, a Slave address is sent. This address is seven bits long followed by an eighth-bit which is the read/write bit. A "1" indicates a request for data (read) and a "0" indicates a data transmission (write). Only the Slave with the calling address that matches the address transmitted by the Master responds by sending back an acknowledge bit by pulling the SDA line Low on the ninth clock.

Once successful Slave addressing is achieved, the data transfer can proceed byte-by-byte as specified by the read/write bit. The Master can terminate the communication by generating a STOP signal to free the bus. However, the Master may generate a START signal without generating a STOP signal first. This is called a repeated START.

## IIC I/O Signals

The I/O signals for the IIC are listed in [Table 10-1](#).

**Table 10-1: IIC I/O Signals**

| Grouping    |     | Signal Name    | Interface | I/O | Initial State | Description                                                    |
|-------------|-----|----------------|-----------|-----|---------------|----------------------------------------------------------------|
| IIC Signals | P1  | Sda_I          | System    | I   | 0             | IIC Serial Data Input from 3-state buffer.                     |
|             | P2  | Sda_O          | System    | O   | 0             | IIC Serial Data Output to 3-state buffer                       |
|             | P3  | Sda_T          | System    | O   | 0             | IIC Serial Data Output Enable to 3-state buffer                |
|             | P4  | Scl_I          | System    | I   | 0             | IIC Serial Clock Input from 3-state buffer.                    |
|             | P5  | Scl_O          | System    | O   | 0             | IIC Serial Clock Output to 3-state buffer.                     |
|             | P6  | Scl_T          | System    | O   | 0             | IIC Serial Clock Output Enable to 3-state buffer               |
| OPB Signals | P7  | Sln_DBus[0:31] | IPIF      | O   | 0             | IIC output data bus                                            |
|             | P8  | Sln_xferAck    | IPIF      | O   | 0             | IIC transfer acknowledge                                       |
|             | P9  | Sln_Retry      | IPIF      | O   | 0             | IIC retry                                                      |
|             | P10 | Sln_ToutSup    | IPIF      | O   | 0             | IIC timeout suppress                                           |
|             | P11 | Sln_ErrAck     | IPIF      | O   | 0             | IIC error acknowledge                                          |
|             | P12 | OPB_ABus[0:31] | IPIF      | I   |               | OPB address bus                                                |
|             | P13 | OPB_BE[0:3]    | IPIF      | I   |               | OPB byte enables                                               |
|             | P14 | OPB_DBus[0:31] | IPIF      | I   |               | OPB data bus                                                   |
|             | P15 | OPB_RNW        | IPIF      | I   |               | Read not Write (OR of all master RNW signals)                  |
|             | P16 | OPB_select     | IPIF      | I   |               | Master has taken control of the bus (OR of all master selects) |
|             | P17 | OPB_seqAddr    | IPIF      | I   |               | OPB sequential address                                         |
| System      | P18 | OPB_Clk        | System    | I   |               | System clock                                                   |
|             | P19 | OPB_Rst        | System    | I   |               | System Reset (active high)                                     |
|             | P20 | IP2INTC_Irpt   | System    | O   |               | System Interrupt                                               |
|             | P21 | Freeze         | System    | I   |               | System Freeze Input (Unconnected in IIC)                       |

## IIC Register Descriptions

The IIC contains addressable registers for read/write operations as shown in [Table 10-2](#). The base address for these registers is set in the parameter C\_BASEADDR. C\_BASEADDR + 0x100 represents the address of the first register in the IIC - in this case, the IIC Control Register. The address of each register is then calculated by an offset to the base address.

The IPIF contains the interrupt registers. The base address for these registers is set by the parameter C\_BASEADDR which represents the address of the first register - in this case, the ISR. The ISR is not used by the IIC design, the location of the first accessible IPIF register is the GIE, offset of 0x1C. The address of each IPIF register is then calculated by an offset to the base address.

[Table 10-2](#) shows all of the IIC registers and the IPIF interrupt registers and their addresses.

Table 10-2: IIC Registers

| Register Name                                      | OPB Address        | Access     |
|----------------------------------------------------|--------------------|------------|
| Control Register (CR)                              | C_BASEADDR + 0x100 | Read/Write |
| Status Register (SR)                               | C_BASEADDR + 0x104 | Read       |
| Transmit FIFO                                      | C_BASEADDR + 0x108 | Read/Write |
| Receive FIFO                                       | C_BASEADDR + 0x10C | Read       |
| IIC Slave Address Register (ADR)                   | C_BASEADDR + 0x110 | Read/Write |
| Transmit FIFO Occupancy                            | C_BASEADDR + 0x114 | Read       |
| Receive FIFO Occupancy                             | C_BASEADDR + 0x118 | Read       |
| Receive FIFO programmable depth interrupt Register | C_BASEADDR + 0x120 | Read/Write |
| GIE                                                | C_BASEADDR + 0x01C | Read/Write |
| IPISR                                              | C_BASEADDR + 0x020 | Read/Write |
| IPIER                                              | C_BASEADDR + 0x028 | Read/Write |
| DeviceSWR                                          | C_BASEADDR + 0x040 | Write Only |

### IIC Control Register

Prior to setting Master Slave Mode Select (MSMS) to a '1', the Tx FIFO should contain the address of the IIC device. All the CR bits can be set at the same time as setting MSMS to a '1' to initiate a bus transaction. When initiating a repeated start condition, the transmit FIFO must be empty, then the repeated start bit should be set to a '1' and then the address of the IIC device should be written to the transmit FIFO. The rest of the FIFO can be filled with data if required. The IIC Control Register is shown in [Table 10-3](#).

Table 10-3: IIC Control Register Bit Definitions

| Bit(s) | Name     | Core Access | Reset Value | Description                                                                                                                                                               |
|--------|----------|-------------|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0      | Reserved |             | 0           | <b>Reserved.</b>                                                                                                                                                          |
| 1      | GC_EN    | Read/Write  | 0           | <b>General Call Enable.</b> Setting this bit high will allow the IIC to respond to a general call address.<br>"0" - General Call Disabled.<br>"1" - General Call Enabled. |

Table 10-3: IIC Control Register Bit Definitions (Continued)

| Bit(s) | Name          | Core Access | Reset Value | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|--------|---------------|-------------|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2      | RSTA          | Read/Write  | 0           | <b>Repeated Start.</b> Writing a "1" to this bit generates a repeated START condition on the bus if the IIC Bus Interface is the current bus Master. Attempting a repeated START at the wrong time, if the bus is owned by another Master, results in a loss of arbitration. This bit is reset when the repeated start occurs. This bit must be set prior to writing the new address to the Tx FIFO .                                                                                             |
| 3      | TXAK          | Read/Write  | 0           | <b>Transmit Acknowledge Enable.</b> This bit specifies the value driven onto the SDA line during acknowledge cycles for both Master and Slave receivers.<br>"1" - ACK bit = "1" - no acknowledge.<br>"0" - ACK bit = "0" - acknowledge.<br>Since Master receivers indicate the end of data reception by not acknowledging the last byte of the transfer, this bit is the means to end a Master receiver transfer. As a slave this bit must be set prior to receiving the byte to not acknowledge. |
| 4      | TX            | Read/Write  | 0           | <b>Transmit/Receive Mode Select.</b> This bit selects the direction of Master/Slave transfers.<br>"1" selects an IIC transmit.<br>"0" selects an IIC receive.<br>This bit does not control the Read/Write bit that is sent on the bus with the address. The Read/Write bit that is sent with an address must be the LSB of the address written into the transmit FIFO.                                                                                                                            |
| 5      | MSMS          | Read/Write  | 0           | <b>Master/Slave Mode Select.</b> When this bit is changed from "0" to "1", the IIC Bus Interface generates a START condition in Master mode. When this bit is cleared, a STOP condition is generated and the IIC Bus Interface switches to Slave mode. When this bit is cleared by the hardware, because arbitration for the bus has been lost, a STOP condition is not generated.                                                                                                                |
| 6      | Tx FIFO Reset | Read/Write  | 0           | <b>Transmit FIFO Reset.</b> This bit must be set if arbitration is lost or if a transmit error occurs to flush the FIFO.<br>"1" resets the Transmit FIFO.<br>"0" Transmit FIFO normal operation.                                                                                                                                                                                                                                                                                                  |
| 7      | EN            | Read/Write  | 0           | <b>IIC Enable.</b> This bit must be set before any other CR bits have any effect.<br>"1" enables the IIC controller.<br>"0" resets and disables the IIC controller.                                                                                                                                                                                                                                                                                                                               |

## Status Register (SR)

This register contains the status of the IIC Bus Interface. This status register is read-only. All bits are cleared upon reset. (Table 10-4)

Table 10-4: Status Register Bits

| Bit(s) | Name          | Core Access | Reset Value | Description                                                                                                                                                                                                                                                                                                                                                                         |
|--------|---------------|-------------|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0      | Tx_FIFO_Empty | Read        | 1           | <b>Transmit FIFO empty.</b> This bit will be set high when the transmit FIFO is empty.                                                                                                                                                                                                                                                                                              |
| 1      | Rc_FIFO_Empty | Read        | 1           | <b>Receive FIFO empty.</b> This bit will be set high when the receive FIFO is empty.                                                                                                                                                                                                                                                                                                |
| 2      | Rc_FIFO_Full  | Read        | 0           | <b>Receive FIFO full.</b> This bit will be set high when the receive FIFO is full. This bit will only be set when all sixteen location in the FIFO are full, regardless of the value written into Rc_FIFO_PIRQ.                                                                                                                                                                     |
| 3      | Tx_FIFO_Full  | Read        | 0           | <b>Transmit FIFO full.</b> This bit will be set high when the transmit FIFO is full.                                                                                                                                                                                                                                                                                                |
| 4      | SRW           | Read        | 0           | <b>Slave Read/Write.</b> When the IIC Bus Interface has been addressed as a Slave (AAS is set), this bit indicates the value of the read/write bit sent by the Master. This bit is only valid when a complete transfer has occurred and no other transfers have been initiated.<br>"1" indicates Master reading from Slave.<br>"0" indicates Master writing to Slave.               |
| 5      | BB            | Read        | 0           | <b>Bus Busy.</b> This bit indicates the status of the IIC bus. This bit is set when a START condition is detected and cleared when a STOP condition is detected.<br>"1" indicates the bus is busy.<br>"0" indicates the bus is idle.                                                                                                                                                |
| 6      | AAS           | Read        | 0           | <b>Addressed as Slave.</b> When the address on the IIC bus matches the Slave address in the Address Register (ADR), the IIC Bus Interface is being addressed as a Slave and switches to Slave mode. This bit is cleared when a stop condition is detected or a repeated start occurs.<br>"1" indicates being addressed as a slave.<br>"0" indicates not being addressed as a slave. |
| 7      | ABGC          | Read        | 0           | <b>Addressed By a General Call.</b> This bit is set high when another master has issued a general call and the general call enable bit is set high, CR(1) = '1'.                                                                                                                                                                                                                    |

## Transmit FIFO

This FIFO contains data to be transmitted on the IIC bus. In transmit mode, data written into this FIFO is output on the IIC bus. Reading of this location will result in reading the current byte being output from the FIFO. (Table 10-5)

Table 10-5: IIC Transmit FIFO

| Bit(s) | Name    | Core Access | Reset Value                  | Description       |
|--------|---------|-------------|------------------------------|-------------------|
| 0 - 7  | D0 - D7 | Read/Write  | Indeterminate <sup>(1)</sup> | IIC Transmit Data |

**Notes:**

1. The value that was available before the reset occurred will still appear on the FIFO outputs

## Receive FIFO

This FIFO contains the data received from the IIC bus. The received IIC data is placed in this FIFO after each complete transfer. The Rc\_FIFO\_OCY must be equal to the Programmable Read Depth Register (Rc\_FIFO\_PIRQ) before throttling will occur. The receive FIFO is read only. Reading this FIFO when it is empty results in indeterminate data being read. (Table 10-6)

Table 10-6: IIC Receive FIFO

| Bit(s) | Name    | Core Access         | Reset Value                  | Description      |
|--------|---------|---------------------|------------------------------|------------------|
| 0 - 7  | D0 - D7 | FIFO -<br>Read Only | Indeterminate <sup>(1)</sup> | IIC Receive Data |

**Notes:**

1. The value that was available before the reset occurred will still appear on the FIFO outputs

## Address Register (ADR)

This field contains the specific Slave address (7-bit address) to be used by the IIC Bus Interface. This register is read/write. (Table 10-7)

Table 10-7: Address Register Bits

| Bit(s) | Name          | Core Access | Reset Value | Description                                              |
|--------|---------------|-------------|-------------|----------------------------------------------------------|
| 7      | Reserved      |             |             | Reserved.                                                |
| 0-6    | Slave Address | Read/Write  | 0x00        | Address used by the IIC Bus Interface when in Slave mode |

## Transmit FIFO Occupancy Register (Tx\_FIFO\_OCY)

This field contains the occupancy value for the Transmit FIFO. Reading this register cannot be used to determine if the FIFO is empty, the Transmit FIFO Empty Interrupt conveys that information. The value read is the binary count value, therefore reading all zeros implies that the first location is filled and reading all ones implies that all sixteen locations are filled. (Table 10-8)

Table 10-8: Transmit FIFO Occupancy Register Bits

| Bit(s) | Name            | Core Access | Reset Value | Description                                                                       |
|--------|-----------------|-------------|-------------|-----------------------------------------------------------------------------------|
| 0-3    | Reserved        |             |             | Reserved.                                                                         |
| 4-7    | Occupancy Value | Read        | 0x0         | Bit 4 is the MSB. A value of 1001 implies that 10 locations in the FIFO are full. |

## Receive FIFO Occupancy Register (Rc\_FIFO\_OCY)

This field contains the occupancy value for the Receive FIFO. This register is read only. Reading this register cannot be used to determine if the FIFO is empty, Rc\_FIFO\_Empty, bit(1) in the status register is used to convey that information. The value read is the binary count value, therefore reading all zeros implies that the first location is filled and reading all ones implies that all sixteen locations are filled. (Table 10-9)

Table 10-9: Receive FIFO Occupancy Register Bits

| Bit(s) | Name            | Core Access | Reset Value | Description                                                                       |
|--------|-----------------|-------------|-------------|-----------------------------------------------------------------------------------|
| 0-3    | Reserved        |             |             | Reserved                                                                          |
| 4-7    | Occupancy Value | Read        | 0x0         | Bit 4 is the MSB. A value of 1001 implies that 10 locations in the FIFO are full. |

## Receive FIFO Programmable Depth Interrupt Register (Rc\_FIFO\_PIRQ)

This field contains the value which will cause the receive FIFO Interrupt to be set. When this value is equal to the Rc\_FIFO\_OCY value, the receive FIFO interrupt will be set and remain set until the equality is no longer true. A read from the receive FIFO will cause the IIC Receive FIFO Interrupt to be cleared. When the Rc\_FIFO\_PIRQ is equal to the Rc\_FIFO\_OCY throttling will also occur to prevent the transmitter from transmitting. This register is read/write. (Table 10-10)

Table 10-10: Receive FIFO Programmable Depth Interrupt Register Bits

| Bit(s) | Name          | Core Access | Reset Value | Description                                                                                                                               |
|--------|---------------|-------------|-------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| 0-3    | Reserved      |             |             |                                                                                                                                           |
| 4-7    | Compare Value | Read/Write  | 0x0         | Bit 4 is the MSB. A value of 1001 implies when ten locations in the receive FIFO have been filled the receive FIFO interrupt will be set. |

## IIC Interrupt Descriptions

### Interrupts

The IIC has eight unique interrupts that are sent to the IPIF. The IPIF has a register that can enable each interrupt independently. The number in the parenthesis is the IPIF interrupt bit number.

#### Interrupt(0) -- Arbitration Lost

Interrupt(0) is the Arbitration Lost interrupt. When arbitration for the IIC bus is lost this interrupt is set. There is no action required by software to clear this interrupt condition

#### Interrupt(1) -- Transmit Error/Slave Transmit Complete

Interrupt(1) is the Transmit Error/Slave Transmit Complete. There is no action required by software to clear this interrupt condition. The following describes the conditions that will cause this interrupt to occur:

IIC operating as a Master Transmitter, Interrupt(1) implies an error. The slave receiver did not acknowledge properly. The MSMS bit in the Control Register will remain high and the IIC will initiate a stop condition which implies that the bus will not be busy.

IIC operating as a Master Receiver, Interrupt(1) implies a transmit complete. This interrupt is caused by setting TXAK high, in the IIC, to indicate to the slave transmitter that the last byte has been transmitted.



IIC operating as a Slave Transmitter, Interrupt(1) implies a transmit complete. This interrupt is caused by the master device to indicate to the IIC that the last byte has been transmitted.

IIC operating as a Slave Receiver, Interrupt(1) implies an error. This interrupt is caused by the IIC (setting TXAK high).

### Interrupt(2) -- Transmit FIFO Empty

Interrupt(2) is the Transmit FIFO . This interrupt is generated only on a transmit throttle condition. Clearing this interrupt requires data to be written to the transmit FIFO . This interrupt will be set and remain set as long as the transmit throttle condition exists.

### Interrupt(3) -- Receive FIFO Full

Interrupt(3) is the Receive FIFO Full interrupt. This interrupt will be set and remain set as long the Rc\_FIFO\_PIRQ is equal to the Rc\_FIFO\_OCY.

Interrupt(4) -- IIC Bus is Not Busy

Interrupt(4) is the IIC Bus is Not Busy. Interrupt(4) is set and will remain set as long as the IIC bus is not busy.

### Interrupt(5) -- Addressed As Slave

Interrupt(5) is the Addressed as Slave interrupt. This interrupt will be set and remain set as long as the IIC is being addressed as a slave.

Interrupt(6) -- Not Addressed As Slave

Interrupt (6) is the Not Addressed As Slave interrupt. This interrupt allows the detection of the end of receive data for a slave receiver when there has been no stop condition (repeated start). The interrupt occurs when a start condition followed by a non-matching slave address is detected. This interrupt will be set whenever the IIC is not addressed as a slave.

### Interrupt(7) -- Transmit FIFO Half Empty

Interrupt(7) is the Transmit FIFO half empty. This interrupt will be set and remain set as long as the MSB of the Tx\_FIFO\_OCY = '0'.

## Flow Descriptions

### Flows

The following is a brief discussion on setting the IIC registers in order to initiate and complete bus transactions. The following must be completed in the order shown to successfully complete an IIC transaction.

#### IIC Master Transmitter, with a Repeated Start

- Write the IIC device address to the Tx FIFO.
- Write data to TX FIFO.
- Write to CR to set MSMS = '1' and TX = '1'.
- Continue writing all data to TX FIFO.
- Wait for Transmit FIFO empty interrupt. This implies the IIC has throttled the bus.
- Write to CR to set RSTA = '1'.
- Write IIC device address to TX FIFO.
- Write all data except last byte to TX FIFO.
- Wait for Transmit FIFO empty interrupt. This implies the IIC has throttled the bus.
- Write to CR to set MSMS = '0'. The IIC will generate a stop condition at the end of the



last byte.

- Write last byte of data to TX FIFO.

### IIC Master Receiver, with a Repeated Start

- Write the IIC device address to the Tx FIFO, Write the programmable read depth interrupt to be 1 less then the total message length.
- Write to CR to set MSMS = '1' and TX = '0'.
- Wait for the Receive FIFO interrupt.
- Write to CR to set TXAK = '1', This will prevent the IIC from acknowledging the next byte received and indicate to the slave transmitter that it has transmitted the final byte of the message. Write the programmable read depth interrupt to be 1. It is important to do this prior to reading data from the FIFO, because as soon a read from the FIFO has occurred, the throttle condition will be removed and the control information must be set up to control the next byte.
- Read all data from receive FIFO.
- Wait for the Receive FIFO interrupt.
- Read data from receive FIFO, Write the programmable read depth interrupt to be 1 less then the total message length.
- Write to CR to set RSTA = '1'. Write IIC device address to TX FIFO.
- Wait for the Receive FIFO interrupt.
- Write to CR to set TXAK = '1', and set MSMS = '0'. This will prevent the IIC from acknowledging the next byte received and indicate to the slave transmitter that it has transmitted the final byte of the message and also generate a stop condition. Write the programmable read depth interrupt to be 1.
- Wait for the Receive FIFO interrupt.
- Read data from receive FIFO.

### IIC Slave Receiver

- Write the programmable read depth interrupt to be equal to the total message length.
- Wait for the Receive FIFO interrupt.
- Read data from receive FIFO.

### IIC Slave Transmitter

- Wait for Addressed as Slave Interrupt.
- Write all data to transmit FIFO.
- Not Addressed as Slave Interrupt may be used to indicate no more data is required by the master receiver.

## Throttle Description

The hardware will throttle the bus to prevent either a receive overrun or a transmit underflow. The throttling is done by holding the SCL line low after the acknowledge bit has been sent. Throttling is independent of Master or Slave operation and only depends upon transmit or receive operation. However, the Master must generate repeated start or stop conditions and that information must be available to the IIC prior to the final byte being transmitted.

Transmit throttling is done whenever Interrupt(2), Transmit FIFO Empty is set. As soon as a new byte is written into the Transmit FIFO, the throttle condition is removed and that byte will be transmitted onto the IIC bus. If that byte is intended to be an address of another device, the repeated start bit must be set prior to writing the address into the Tx

FIFO. This allows the IIC to generate a repeated start condition on the bus before transmitting the address of the new device.

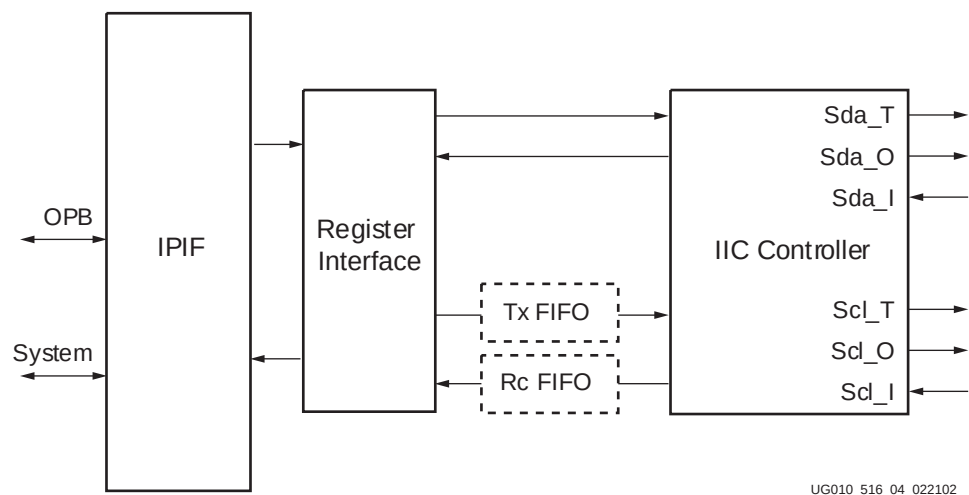
Receive throttling is done whenever the Rc\_FIFO\_PIRQ is set. The throttle condition is removed when a byte from the Receive FIFO is read.

When the IIC is a master and a receive throttle condition exists the IIC will generate a stop condition if the MSMS bit is change from a 1 to a 0. This allows single byte reads from a slave device. The transmit acknowledge must be set up to no-ack the byte. This will command the slave device to release the SDA line so a stop condition can occur.

When the IIC is a master and in a receive throttle condition and the transmit register/FIFO is empty. Setting the repeated start in the Control Register will cause a transmit throttle condition to occur.

## IIC Block Diagram

The top-level block diagram for the IIC is shown in Figure 10-2.



UG010\_516\_04\_022102

Figure 10-2: IIC Top-level Block Diagram

## Specification Exceptions

### Exceptions to the Philips I<sup>2</sup>C-Bus Specification, Version 2.1, January 2000

High-speed mode (Hs-mode) is not currently supported by the IIC IP.

Tri-state buffers are used to perform the wired-AND function inherent in this bus structure.

The Xilinx FPGA device ratings must not be exceeded when interconnecting the IIC to other devices.

Make sure that the correct SelectI/O™ standard is selected for electrical compatibility with other devices.



## OPB to DCR Bridge

### Overview

The OPB to DCR Bridge translates transactions received on its OPB slave interface into DCR master operations. Its design utilizes an Intellectual Property InterFace (IPIF) module to abstract OPB transactions into a simple SRAM style protocol that is easier to design with. The main advantage of using the bridge instead of the CPU to control the DCR bus is that it provides a memory mapped interface that may be preferable to the use of special move to/move from DCR instructions. Since the bridge typically runs at a slower clock frequency than the CPU, its timing requirements are also less stringent. The OPB to DCR Bridge implements a simple and flexible method for communicating with DCR devices.

### Related Documents

- *IPIF Specification*
- *IBM CoreConnect™ 64-Bit On-Chip Peripheral Bus: Architecture Specifications*
- *IBM CoreConnect 32-Bit Device Control Register Bus: Architecture Specifications*
- *Virtex-II Pro™ Platform FPGAs (Advance Product Specification)*

### Features

- 32-bit DCR master with a 10-bit DCR address bus
- Memory-mapped interface from OPB to DCR, no special instructions required
- Increased timing flexibility in typical systems where the OPB clock is slower than the CPU clock
- Allows master devices other than the CPU to access the DCR bus
- Provides a mechanism where CoreConnect systems without a CPU can support DCR devices

### Module Port Interfaces

Table 11-1: Global Signals

| Name    | Direction | Description      |
|---------|-----------|------------------|
| OPB_Clk | Input     | OPB system clock |
| OPB_Rst | Input     | OPB system reset |

Table 11-2: OPB Slave Signals

| Name           | Direction | Description                |
|----------------|-----------|----------------------------|
| OPB_ABus[0:31] | Input     | OPB address bus            |
| OPB_BE[0:3]    | Input     | OPB byte enables           |
| OPB_DBus[0:31] | Input     | OPB data bus               |
| OPB_RNW        | Input     | OPB read not write         |
| OPB_select     | Input     | OPB select                 |
| OPB_seqAddr    | Input     | OPB sequential address     |
| S1_DBus[0:31]  | Output    | Slave data bus             |
| S1_DBusEn      | Output    | Slave data bus enable      |
| S1_errAck      | Output    | Slave error acknowledge    |
| S1_retry       | Output    | Slave bus cycle retry      |
| S1_toutSup     | Output    | Slave timeout suppress     |
| S1_xferAck     | Output    | Slave transfer acknowledge |

Table 11-3: DCR Master Signals

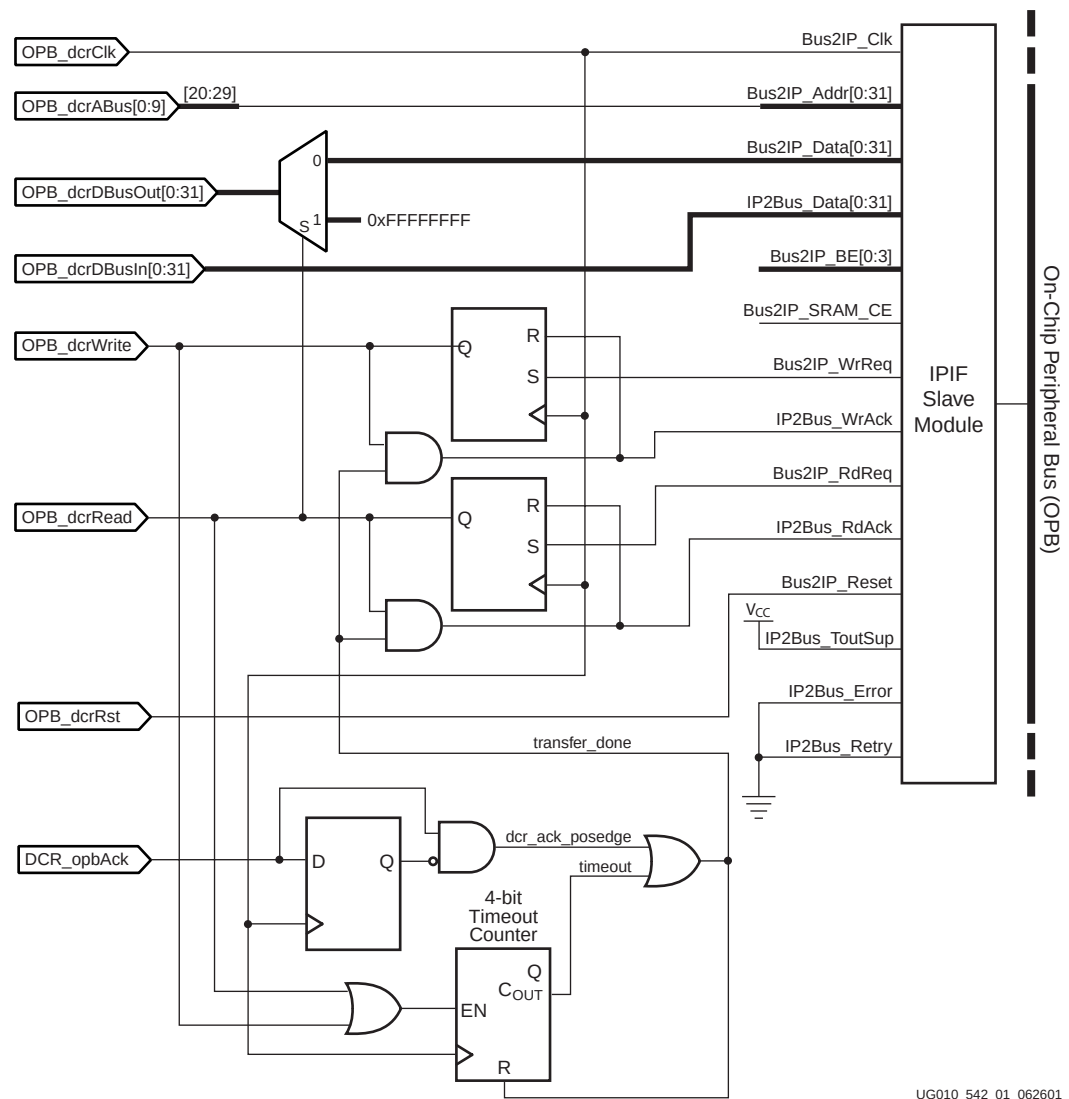
| Signal               | Direction | Description          |
|----------------------|-----------|----------------------|
| DCR_opbAck           | Input     | DCR OPB acknowledge  |
| DCR_opbDBusIn[0:31]  | Input     | DCR OPB data bus in  |
| OPB_dcrABus[0:9]     | Output    | OPB DCR address bus  |
| OPB_dcrClk           | Output    | OPB DCR clock        |
| OPB_dcrDBusOut[0:31] | Output    | OPB DCR data bus out |
| OPB_dcrRead          | Output    | OPB DCR read         |
| OPB_dcrRst           | Output    | OPB DCR reset        |
| OPB_dcrWrite         | Output    | OPB DCR write        |

Table 11-4: Parameters

| Name        | Direction | Description                                                                  |
|-------------|-----------|------------------------------------------------------------------------------|
| C_BASE_ADDR | N/A       | 32-bit base address of OPB to DCR Bridge (must be aligned to a 4KB boundary) |

## Implementation

Figure 11-1 shows a conceptual high level view of the design of the IPIF-based OPB to DCR Bridge.



UG010\_542\_01\_062601

Figure 11-1: OPB to DCR Bridge

The read/write requests from the IPIF are latched by set-reset flip flops which then drive the OPB\_dcrRead/OPB\_dcrWrite signals. Simultaneously, the corresponding DCR address (OPB\_dcrABus) and write data bus out (OPB\_dcrDBusOut) signals are driven. A timeout counter is also enabled. When a DCR slave returns an acknowledge (rising edge of DCR\_opbAck detected) or a DCR timeout occurs after 16 clock cycles, the corresponding Bus2IP\_RdAck and Bus2IP\_WrAck signals are asserted back to the IPIF to end the transaction. The DCR protocol specifies that timeouts are not treated as errors, therefore a DCR timeout terminates the transaction without any errors being reported back to OPB.

For debugging purposes, the data bus out is set to 0xFFFFFFFF during DCR reads so that a read timeout would return 0xFFFFFFFF by default. OPB\_dcrClk and OPB\_dcrRst are directly connected to Bus2IP\_Clk/Bus2IP\_Rst for convenience, even though these signals can be taken directly from the OPB clock and reset lines.

The DCR is organized as a 32-bit bus with 10 bits of address to reference a particular 32-bit DCR register. This 1K x 32 bit DCR address space maps into a 4KB address space on OPB.

Since DCR does not support byte enables, the byte enables are ignored. The user must therefore be careful when accessing DCR registers with 1- to 3-byte instructions, since the inactive byte lanes will carry undefined data. It is recommended that only fullword transactions be used.

## OPB to PCI Bridge

### Overview

The OPB to PCI Bridge translates transactions received on its OPB slave interface into PCI master operations. Its design utilizes an Intellectual Property InterFace (IPIF) module to abstract OPB transactions into a simple SRAM style protocol that is easier to design with. The PCI interface logic utilizes the **Xilinx LogiCORE PCI 64/66** product which helps designers to develop high performance, fully compliant PCI devices. The OPB to PCI Bridge uses the PCI Core to implement a simple 32 bit, 33 MHz PCI master. Configuration, I/O, and Memory transfer types are supported over PCI to provide compatibility with a large number of common PCI devices.

### Related Documents

- *IPIF Specification*
- *IBM CoreConnect 64-Bit On-Chip Peripheral Bus: Architecture Specifications*
- *Virtex-II Pro Platform FPGAs (Advance Product Specification)*

### Features

- 32-bit OPB Slave interface with IPIF-based design
- 32-bit/33 MHz PCI interface that is V2.2 compliant to the PCI specification
- Capable of generating Configuration, I/O, and Memory transactions
- PCI clock can be divided down from the OPB clock by any integer divisor. PCI to OPB clock ratios of 1:1, 1:2, 1:3, etc. are possible

### Module Port Interface

Table 12-1: Global Signals

| Name    | Direction | Description      |
|---------|-----------|------------------|
| OPB_Clk | Input     | OPB system clock |
| OPB_Rst | Input     | OPB system reset |



Table 12-2: OPB Slave Signals

| Name           | Direction | Description                |
|----------------|-----------|----------------------------|
| OPB_ABus[0:31] | Input     | OPB address bus            |
| OPB_BE[0:3]    | Input     | OPB byte enables           |
| OPB_DBus[0:31] | Input     | OPB data bus               |
| OPB_RNW        | Input     | OPB read not write         |
| OPB_select     | Input     | OPB select                 |
| OPB_seqAddr    | Input     | OPB sequential address     |
| S1_DBus[0:31]  | Output    | Slave data bus             |
| S1_DBusEn      | Output    | Slave data bus enable      |
| S1_errAck      | Output    | Slave error acknowledge    |
| S1_retry       | Output    | Slave bus cycle retry      |
| S1_toutSup     | Output    | Slave timeout suppress     |
| S1_xferAck     | Output    | Slave transfer acknowledge |

Table 12-3: PCI Master Signals

| Signal   | Direction    | Description                     |
|----------|--------------|---------------------------------|
| GNT_N    | Input        | PCI Grant                       |
| PCLK_FB  | Input        | PCI Clock External Feedback     |
| PCLK_IN  | Input        | PCI Reference Clock In          |
| IDSEL    | Output       | PCI Identification Select       |
| INTR_A   | Output       | PCI Interrupt A                 |
| PCLK     | Output       | PCI External Clock Out          |
| REQ_N    | Output       | PCI Request Bus Access          |
| RST_N    | Output       | PCI Reset External PCI Devices  |
| ACK64_N  | Input-Output | PCI 64 Bit Transfer Acknowledge |
| AD[63:0] | Input-Output | PCI Address/Data Bus            |
| CBE[7:0] | Input-Output | PCI Command / Byte Enables      |
| DEVSEL_N | Input-Output | PCI Device Select               |
| FRAME_N  | Input-Output | PCI Framing Signal              |
| IRDY_N   | Input-Output | PCI Initiator Ready             |
| PAR      | Input-Output | PCI Parity                      |
| PAR64    | Input-Output | PCI 64 Bit Parity               |
| PERR_N   | Input-Output | PCI Parity Error                |

Table 12-3: PCI Master Signals (Continued)

| Signal  | Direction    | Description                 |
|---------|--------------|-----------------------------|
| REQ64_N | Input-Output | PCI 64 Bit Transfer Request |
| SERR_N  | Input-Output | PCI Error                   |
| STOP_N  | Input-Output | PCI Stop                    |
| TRDY_N  | Input-Output | PCI Target Ready            |

Table 12-4: Parameters

| Name        | Direction | Description                                                                    |
|-------------|-----------|--------------------------------------------------------------------------------|
| C_BASE_ADDR | N/A       | 32-bit base address of OPB to PCI Bridge (must be aligned to a 256MB boundary) |

## Implementation

Figure 12-1 shows a conceptual high-level view of the design OPB to PCI Bridge. The Bridge uses an IPIF to help abstract the OPB interface into a simpler protocol. It also includes a Xilinx PCI Core to simplify the task of building a fully compliant PCI master interface. The interface logic between the IPIF and PCI Core is based on the sample initiator design described in detail by the *LogiCORE PCI Design Guide (Chapter 12)*. The

IPIF signals are decoded into a simple set of signals that control the initiator state machine logic described by the above document.

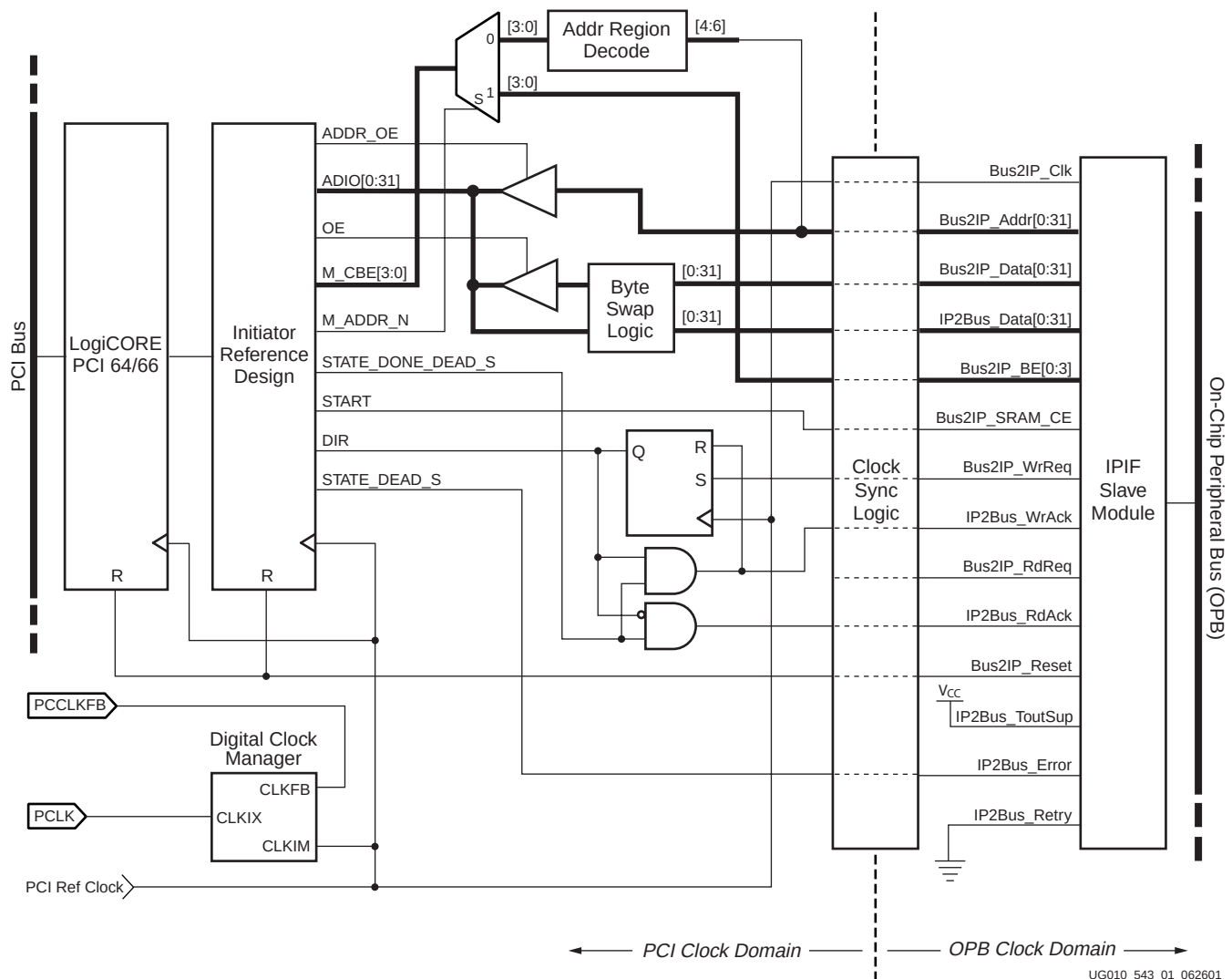


Figure 12-1: OPB to PCI Bridge

The IPIF signals are passed through synchronization logic to allow the OPB and PCI clock domains to be different. The synchronization logic requires that the OPB clock be an integer multiple of the PCI clock and that the rising edges of the two clocks be phase-aligned to each other. In typical systems the PCI clock will operate at 25-33 MHz while the OPB clock would be 50-100 MHz. Supporting different clock frequencies between OPB and PCI prevents the PCI clock from limiting the OPB clock.

PCI errors due to master or target abort conditions are passed back to the OPB as an errAck response. To allow for PCI device discovery without causing CPU machine-check exceptions, errors that occur during configuration cycles are not reported back to OPB. Reading from a non-existent PCI device during a configuration cycle will return data of 0xFFFFFFFF. PCI retry is not fed back to the OPB, but the master will back off the PCI bus and try the transaction again. OPB retry need not be implemented; since deadlock is not possible, the bridge has only one direction of transaction flow.

The OPB to PCI Bridge is designed for PCI systems similar to those found in typical PCs where there is only one PCI master on the bus. (GNT\_N should be tied low in single-

master systems). The bridge does not need any bus arbitration logic if it is the only master device. However, an external PCI bus arbiter is required in multi-master PCI systems.

The bridge is designed for 33 MHz PCI clock frequencies, but should also work at higher frequencies up to 66 MHz by increasing the PCI reference clock frequency. The PCI I/O buffer instantiations may need to be updated so the correct I/O signalling standards are utilized for the given frequency and voltage of the system. The bridge is designed to drive the external PCI system clock and utilizes the Digital Clock Managers available in the Virtex II-Pro FPGA to de-skew the external PCI clock from the internal reference clock.

OPB uses big-endian byte addressing, while PCI uses little-endian byte addressing. To translate data between the two busses, the data bytes and byte enable bits for each 32-bit OPB word are swapped when going to or from PCI.

## Memory Map

The Bridge maps a 256MB OPB address space into four PCI memory regions. Each memory region corresponds to a different transaction type as shown in [Table 12-5](#) below. The base address is user-specified as a parameter during module instantiation.

*Table 12-5: OPB to PCI Bridge Memory Map*

| PCI Transfer Type                   | OPB Address Boundaries (Hex) |                      | PCI Address Boundaries (Hex) |          |
|-------------------------------------|------------------------------|----------------------|------------------------------|----------|
|                                     | Lower                        | Upper                | Lower                        | Upper    |
| Memory                              | Base Addr + 00000000         | Base Addr + 07FFFFFF | 00000000                     | 07FFFFFF |
| I/O                                 | Base Addr + 08000000         | Base Addr + 0BFFFFFF | 00000000                     | 03FFFFFF |
| Configuration (External Device)     | Base Addr + 0C000000         | Base Addr + 0DFFFFFF | 00000000                     | 01FFFFFF |
| Configuration (Self) <sup>(1)</sup> | Base Addr + 0E000000         | Base Addr + 0FFFFFFF | 00000000                     | 01FFFFFF |

**Notes:**

1. Software must first use self-configuration accesses to enable the master interface on the bridge before it can perform any Memory or I/O transactions.

## Xilinx LogiCORE PCI 64/66

Though the bridge is mainly designed for 32 bit/33 MHz applications, the PCI Core supports up to 64 bit/66 MHz PCI transactions, and can even be migrated up to PCI-X functionality. The PCI Core also contains a slave interface that can be utilized to build a two-way OPB ↔ PCI bridge. An evaluation version of the PCI Core is provided with the OPB to DCR Bridge ([\\$V2PRO/source/sw/verilog/pci\\_brg/pci\\_64\\_66\\_eval](#)). It contains full technical documentation, simulation models, and reference designs that the user may work with. Contact your local sales office or go to <http://www.xilinx.com/pci> for more information about the PCI offerings from Xilinx.



## PLB Arbiter

### Overview

This documents the high-performance, 64-bit version Processor Local Bus (PLB) arbiter with a three-cycle-only arbitration feature. The PLB arbiter is a pipelined soft core consisting of a bus control unit, a watchdog timer, and separate address, write, and read data path units.

The PLB arbiter core features:

- PLB address and data steering support for eight masters
- 64-bit and/or 32-bit support for masters and slaves
- PLB address pipelining
- PLB arbitration support for eight masters
- Three cycle arbitration only
- Four levels of dynamic master request priority
- PLB watchdog timer
- PLB architecture compliant

**Figure 13-1** demonstrates how the PLB arbiter is interconnected for the purpose of FPGA development or system-on-a-chip design.

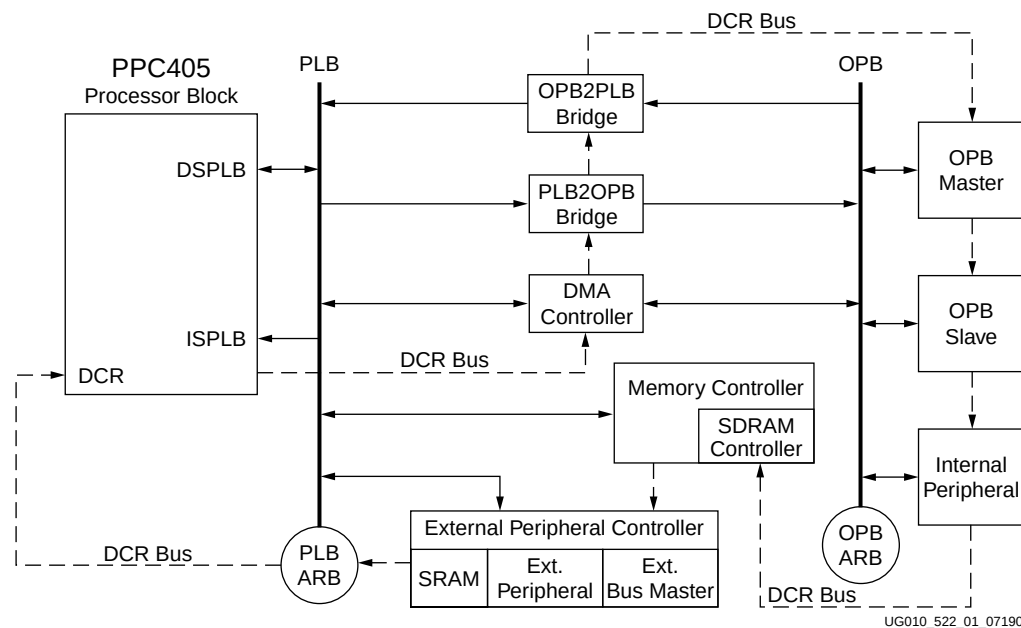


Figure 13-1: PLB Arbiter Interconnection

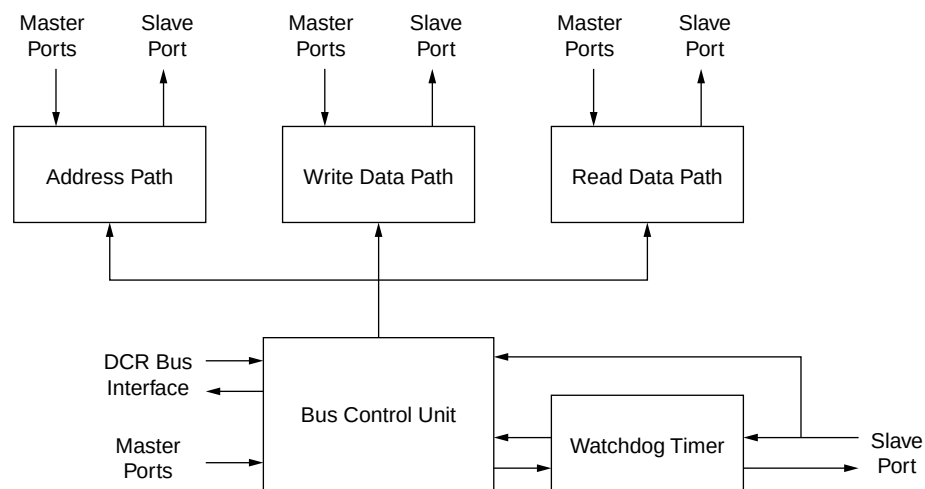
As shown in **Figure 13-1**, the on-chip bus structure provides a link between the processor core (PPC405) and other peripherals consisting of Processor Local Bus (PLB) and On-Chip Peripheral Bus (OPB) master and slave devices.

The PLB is a high-performance bus used to access memory through the bus interface units. The two bus interface units shown in **Figure 13-1**, external peripheral controller and memory controller, are the PLB slaves. The processor core has two PLB master connections, one for instruction cache and one for data cache. Attached to the PLB is also a direct memory access (DMA) controller, which is a PLB master device used to improve data transfer performance in data-intensive applications.

Lower performance peripherals (such as OPB master, slave, and other internal peripherals) are attached to the OPB. A bridge is provided between the PLB and OPB to enable data transfer by PLB masters to and from OPB slaves. The example in **Figure 13-1** shows two bridges, a PLB to OPB bridge (which is a slave on the PLB and a master on the OPB), and an OPB to PLB bridge (which is a slave on the OPB and a master on the PLB). OPB peripherals can also comprise DMA peripherals.

The device control register (DCR) bus is used primarily for accessing status and control registers within the various PLB and OPB masters and slaves. It is meant to off-load the PLB from the lower-performance status and control read and write transfers. The DCR bus architecture allows data transfers among OPB peripherals to occur independently from, and concurrent with, data transfers between the processor and memory, or among other PLB devices.

**Figure 13-2** provides a comprehensive block diagram of the PLB arbiter.



UG010\_522\_02\_070301

**Figure 13-2: PLB Arbiter Block Diagram**

## Bus Control Unit

The PLB arbiter bus control unit consists of a bus arbitration control unit that manages the address and data flow through the PLB and DCRs. The bus arbitration control unit supports arbitration for eight PLB masters. The address and data flow control logic provides address pipelining and address and data steering support for eight PLB masters and any number of PLB slaves.

The PLB arbiter device control registers are used to control and report status from the bus arbitration control unit and address flow control logic. The registers are accessed by using the *move from device* control register (*mf dcr*) and *move to device* control register (*mt dcr*) instructions, which move data between the device control registers and the processor's general purpose registers.

Refer to **PLB Arbiter Registers**, [page 1493](#) for additional information.

## Address Path Unit

The PLB arbiter address path unit contains the necessary muxing to select the master address which will be driven to the slave devices on the PLB address output.

## Read Data Path Unit

The PLB arbiter read data path unit contains the necessary steering logic for the master and slave read data buses.

## Write Data Path Unit

The PLB arbiter write data path unit contains the necessary steering logic for the master and slave write data buses.

## Watchdog Timer

The PLB arbiter watchdog timer provides the necessary handshake to complete the transfer in the event that a master's request times out on the PLB. When handshaking the master, the **M\_type** signals are ignored and all transfers are terminated as if they were normal memory transfers (**M\_type** = 0b000). During burst transfers, the **PLB\_MRdBTerm** and **PLB\_MWdBTerm** signals are driven, when appropriate, in order to minimize the impact on bus latency associated with this type of transfer.

Refer to **PLB Arbiter Operations**, [page 1498](#) to see how the PLB arbiter handles the timed out PLB transfers.

## Clock and Power Management

The PLB arbiter uses a single clock source and does not support multiple clock domains, internal or external, to the core. Therefore, all master and slave devices attached to the PLB arbiter are expected to operate at the frequency of the core.

## Reset Considerations

A logic 1 placed on the reset input while the clock is running causes the PLB arbiter state machines to initiate a reset sequence. To complete the reset sequence, the reset input must be held at a logic 1 for a minimum of five PLB clock cycles.

**Note:** For the PLB arbiter state machines to reset properly, once started, the reset sequence must be allowed to complete. Otherwise, unexpected behavior can occur following the negation of reset.

Following a reset sequence, the PLB arbiter must sample the reset signal negated for two clock cycles before its state machines can begin processing bus transfer requests.

Refer to **PLB Arbiter Registers**, [page 1493](#) for information on the effect of reset on the PLB arbiter device control registers.

# Supported Configurability

The PLB arbiter supports several configurability options as described below.

## Programmable Arbitration

During the bus arbitration cycle, the bus arbitration control unit uses the **M\_priority[n\*2:n\*2+1]** signals to determine which master will be granted the bus. Specifically, only the priority inputs of masters with their respective **M\_request[n]** signal asserted are used in determining the highest request priority. In addition, the core supports



the fixed priority scheme to handle “tie” situations (that is, situations when two or more masters request the bus simultaneously while presenting the same level of request priority). Selection of the priority mode during tie situations is shown in Table 13-1.

Table 13-1 shows all eight bus masters in the order of priority.

Table 13-1: Priority Order for Bus Masters

| Highest Priority | Decreasing Priority |          |          |          |          |          | Lowest Priority |
|------------------|---------------------|----------|----------|----------|----------|----------|-----------------|
| Master 0         | Master 1            | Master 2 | Master 3 | Master 4 | Master 5 | Master 6 | Master 7        |

## PLB Arbiter Signals

Table 13-2 provides a summary of all PLB arbiter core input/output (I/O) signals in alphabetical order, the interfaces under which they are grouped, followed by a brief description. Note that PLB arbitration support for eight (8) masters is represented using  $n$  (where  $n$  is the number of a master 0, 1, 2, 3, 4, 5, 6, or 7).

Table 13-2: PLB Arbiter Pin Description

| Signal Name            | Interface | I/O    | Description                                  |
|------------------------|-----------|--------|----------------------------------------------|
| CPU_dcrAddr[0:9]       | DCR       | Input  | CPU DCR address bus                          |
| CPU_exeMfDcr           | DCR       | Input  | CPU read from DCR indicator                  |
| CPU_exeMtDcr           | DCR       | Input  | CPU write to DCR indicator                   |
| M_abort[n]             | Master[n] | Input  | Master[n] abort bus request indicator        |
| M_ABus[n*32:n*32+31]   | Master[n] | Input  | Master[n] address bus                        |
| M_BE[n*8:n*8+7]        | Master[n] | Input  | Master[n] byte enables                       |
| M_busLock[n]           | Master[n] | Input  | Master[n] bus lock                           |
| M_compress[n]          | Master[n] | Input  | Master[n] compressed data transfer indicator |
| M_guarded[n]           | Master[n] | Input  | Master[n] guarded transfer indicator         |
| M_lockErr[n]           | Master[n] | Input  | Master[n] lock error indicator               |
| M_msize[n:n+1]         | Master[n] | Input  | Master[n] data bus port width                |
| M_ordered[n]           | Master[n] | Input  | Master[n] synchronize transfer indicator     |
| M_priority[n*2:n*2+1]  | Master[n] | Input  | Master[n] bus request priority               |
| M_rdBurst[n]           | Master[n] | Input  | Master[n] burst read transfer indicator      |
| M_request[n]           | Master[n] | Input  | Master[n] bus request                        |
| M_RNW[n]               | Master[n] | Input  | Master[n] read not write                     |
| M_size[n*4:n*4+3]      | Master[n] | Input  | Master[n] transfer size                      |
| M_type[n*3:n*3+2]      | Master[n] | Input  | Master[n] transfer type                      |
| M_wrBurst[n]           | Master[n] | Input  | Master[n] burst write transfer indicator     |
| M_wrDBus[n*64:n*64+63] | Master[n] | Input  | Master[n] write data bus                     |
| PGM_dcrAddr[0:5]       | DCR       | Input  | Program DCR address space locator            |
| PLB_abort              | Slave     | Output | PLB abort bus request indicator              |

Table 13-2: PLB Arbiter Pin Description (Continued)

| Signal Name               | Interface | I/O    | Description                                         |
|---------------------------|-----------|--------|-----------------------------------------------------|
| PLB_ABus[0:31]            | Slave     | Output | PLB address bus                                     |
| PLB_BE[0:7]               | Slave     | Output | PLB byte enables                                    |
| PLB_busLock               | Slave     | Output | PLB bus lock                                        |
| PLB_compress              | Slave     | Input  | PLB compressed data transfer indicator              |
| PLB_dcrAck                | DCR       | Output | PLB DCR data transfer acknowledge                   |
| PLB_dcrData[0:31]         | DCR       | Output | PLB DCR read data bus                               |
| PLB_guarded               | Slave     | Output | PLB guarded transfer indicator                      |
| PLB_lockErr               | Slave     | Output | PLB lock error indicator                            |
| PLB_masterID[0:2]         | Slave     | Output | PLB current master identifier                       |
| PLB_MAddrAck[n]           | Master[n] | Output | PLB master[n] address acknowledge                   |
| PLB_MBusy[n]              | Master[n] | Output | PLB master[n] slave busy indicator                  |
| PLB_MErr[n]               | Master[n] | Output | PLB master[n] slave error indicator                 |
| PLB_MRdBTerm[n]           | Master[n] | Output | PLB master[n] terminate read burst indicator        |
| PLB_MRdDAck[n]            | Master[n] | Output | PLB master[n] read data acknowledge                 |
| PLB_MRdDBus[n*64:n*64+63] | Master[n] | Output | PLB master[n] read data bus                         |
| PLB_MRdWdAddr[n*4:n*4+3]  | Master[n] | Output | PLB master[n] read word address                     |
| PLB_MRearbitrate[n]       | Master[n] | Output | PLB master[n] bus rearbitrate indicator             |
| PLB_Msize[n*2:n*2+1]      | Master[n] | Output | PLB master[n] slave data bus port width             |
| PLB_MWrBTerm[n]           | Master[n] | Output | PLB master[n] terminate write burst indicator       |
| PLB_MWrDAck[n]            | Master[n] | Output | PLB master[n] write data acknowledge                |
| PLB_msize[0:1]            | Slave     | Output | PLB data bus port width indicator                   |
| PLB_ordered               | Slave     | Output | PLB synchronize transfer indicator                  |
| PLB_PAVValid              | Slave     | Output | PLB primary address valid indicator for upto 66 MHz |
| PLB_pendPri[0:1]          | Slave     | Output | PLB pending request priority                        |
| PLB_pendReq               | Slave     | Output | PLB pending bus request indicator                   |
| PLB_rdBurst               | Slave     | Output | PLB burst read transfer indicator                   |
| PLB_rdPrim                | Slave     | Output | PLB secondary to primary read request indicator     |
| PLB_reqPri[0:1]           | Slave     | Output | PLB current request priority                        |
| PLB_RNW                   | Slave     | Output | PLB read not write                                  |
| PLB_SAVValid              | Slave     | Output | PLB secondary address valid for up to 66 MHz        |
| PLB_size[0:3]             | Slave     | Output | PLB transfer size                                   |
| PLB_type[0:2]             | Slave     | Output | PLB transfer type                                   |
| PLB_wrBurst               | Slave     | Output | PLB burst write transfer indicator                  |
| PLB_wrDBus[0:63]          | Slave     | Output | PLB write data bus                                  |

Table 13-2: PLB Arbiter Pin Description (Continued)

| Signal Name       | Interface | I/O    | Description                                      |
|-------------------|-----------|--------|--------------------------------------------------|
| PLB_wrPrim        | Slave     | Output | PLB secondary to primary write request indicator |
| Reset             | CLK       | Input  | Reset                                            |
| Sl_addrAck        | Slave     | Input  | Slave address acknowledge                        |
| Sl_MErr[0:7]      | Slave     | Input  | Slave error indicator                            |
| Sl_MBusy[0:7]     | Slave     | Input  | Slave busy indicator                             |
| Sl_rdBTerm        | Slave     | Input  | Slave terminate read burst transfer              |
| Sl_rdComp         | Slave     | Input  | Slave read transfer complete indicator           |
| Sl_rdDAck         | Slave     | Input  | Slave read data acknowledge                      |
| Sl_rDBus[0:63]    | Slave     | Input  | Slave read data bus                              |
| Sl_rdWdAddr[0:3]  | Slave     | Input  | Slave read word address                          |
| Sl_rearbitrate    | Slave     | Input  | Slave rearbitrate bus indicator                  |
| Sl_ssize[0:1]     | Slave     | Input  | Slave data bus port size indicator               |
| Sl_wait           | Slave     | Input  | Slave wait indicator                             |
| Sl_wrBTerm        | Slave     | Input  | Slave terminate write burst transfer             |
| Sl_wrComp         | Slave     | Input  | Slave write transfer complete indicator          |
| Sl_wrDAck         | Slave     | Input  | Slave write data acknowledge                     |
| Clk               | CLK       | Input  | System clock                                     |
| XXX_dcrData[0:31] | DCR       | Input  | DCR write data bus                               |

**Note:** When leaving a PLB master port unused, the M\_request[n] and M\_busLock[n] signals on the unused port must be tied to a logic 0 for the PLB arbiter to operate properly. All other signals on this port can be left unconnected.

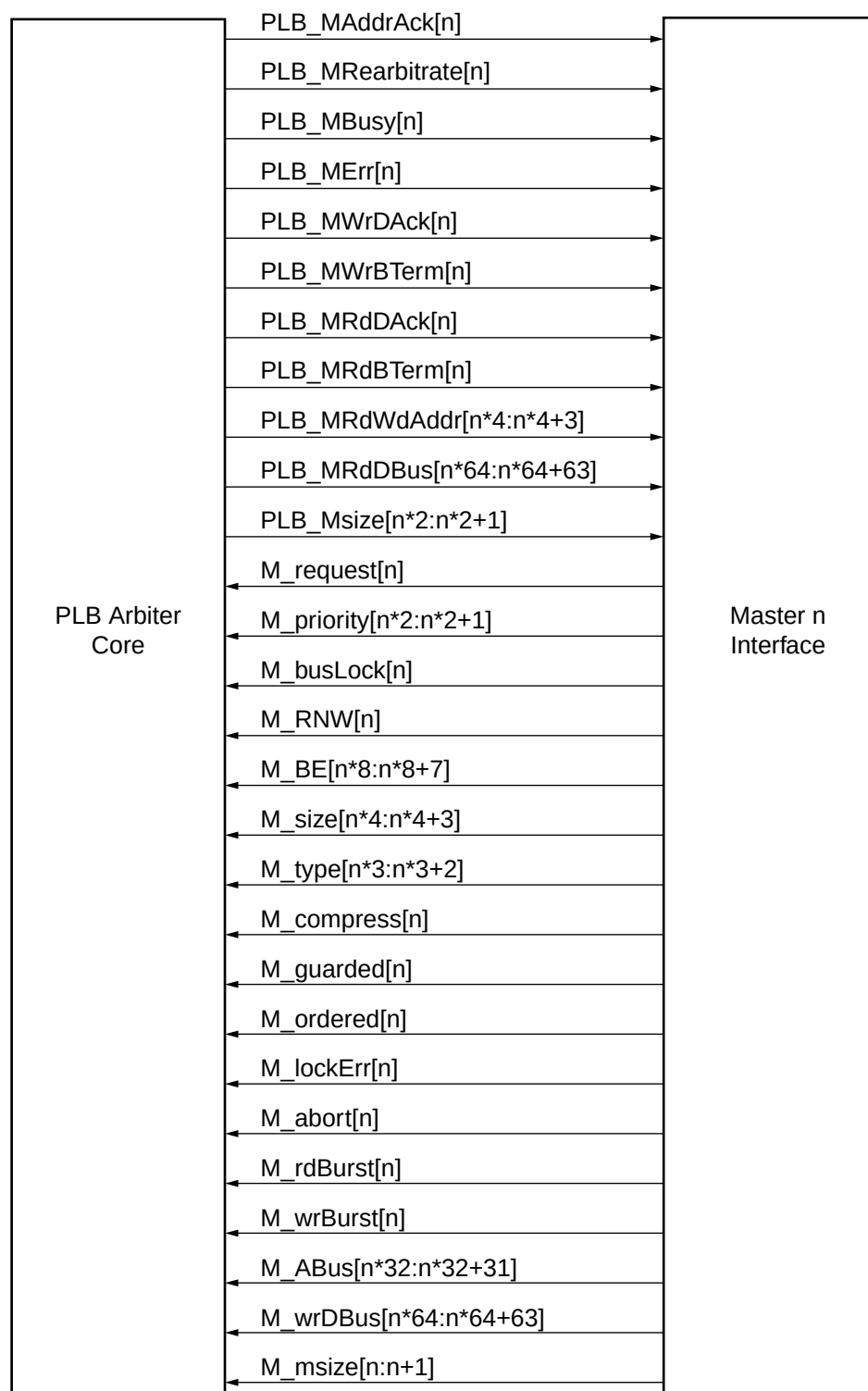
## PLB Arbiter Interfaces

The PLB arbiter I/O signals are grouped under the following interface categories depending on their function. A brief description of each interface and various signals grouped under the particular interface is provided in the following subsections.

- Master[n] Interface
- Slave Interface
- DCR Interface
- Clock Interface

## Master[n] Interface

Figure 13-3 shows all master[n] (where  $n$  is the number of a master 0, 1, 2, 3, 4, 5, 6, or 7) interface I/O signals followed by detailed functional description. Refer to the *Processor Local Bus Architecture Specifications* for additional information.



UG010\_522\_03\_070501

Figure 13-3: Master[n] Interface

## PLB\_MAddrAck[n] (PLB Master Address Acknowledge)

This signal is asserted to indicate that the slave has acknowledged the address and will latch the address and all of the transfer qualifiers at the end of the current clock cycle. This signal is asserted by the slave only while **PLB\_PAVValid** or **PLB\_SAVValid** is asserted and should remain negated at all other times. This signal is the OR of all slaves attached to the PLB bus.

**Note:** It is possible for the slave to acknowledge a valid address in the same clock cycle in which **PLB\_PAVValid** or **PLB\_SAVValid** is first asserted.

## PLB\_MRearbitrate[n] (PLB Master Rearbitrate)

This signal is asserted to indicate that the slave is unable to perform the currently requested transfer and requires the PLB arbiter to rearbitrate the bus. This signal is asserted by the slave only while **PLB\_PAVValid** or **PLB\_SAVValid** is asserted and should remain negated at all other times. (Note, all slave **SI\_rearbitrate** signals are OR'ed together as a single **SI\_rearbitrate** input to the PLB arbiter).

When asserted in response to the assertion of **PLB\_PAVValid**, the PLB arbiter will pass this signal to the master who has control of the address bus, and rearbitrate and grant the bus to the highest priority request in the next clock cycle. Furthermore, to avoid a possible deadlock scenario, the PLB arbiter will ignore the original master request during rearbitration for one clock cycle.

When asserted in response to the assertion of **PLB\_SAVValid**, the PLB arbiter will not pass this signal to the controlling master or rearbitrate the bus. Instead, the PLB arbiter will wait for the current request to become a primary request, and **PLB\_PAVValid** to be asserted, and then sample **SI\_wait**, **SI\_addrAck**, **SI\_rearbitrate**, and **M\_abort[n]**.

### Notes:

1. If **SI\_addrAck** and **SI\_rearbitrate** are sampled asserted in the same clock cycle, the PLB arbiter will ignore the **SI\_rearbitrate** signal and will not rearbitrate the bus. As a result, the slave must perform the requested data transfer (read or write) to avoid a possible deadlock scenario.
2. If the bus had been previously locked, the **SI\_rearbitrate** signal will be ignored by the PLB arbiter to prevent the interruption of an "atomic" operation. Hence, to prevent a deadlock scenario in this case, the locking master must negate its **M\_request[n]**, **M\_RNW[n]**, and **M\_busLock[n]** signals for a minimum of two clock cycles following the sampling of **PLB\_MRearbitrate[n]** asserted.

## PLB\_MBusy[n] (PLB Master Busy)

These signals are driven by the slave and are used to indicate that the slave is either busy performing a read or a write transfer, or has a read or write transfer pending. Each slave is required to drive a separate busy signal for each master attached on the PLB bus (i.e. **SI\_MBusy[0]** corresponds to master ID 0, **SI\_MBusy[1]** corresponds to master ID 1 etc.). The slave should latch the master ID and use this ID to drive the corresponding master busy signal until the data transfer has been completed.

**Note:** The widths of the **SI\_MBusy[0:n]** and **PLB\_MBusy[0:n]** signals are determined by the number of masters supported by the particular PLB based system.

## PLB\_MErr[n] (PLB Master Error)

These signals are driven by the slave and are used to indicate that the slave has encountered an error during a transfer that was initiated by this master. Each slave is required to drive a separate error signal for each master attached on the PLB bus (that is, **SI\_MErr[0]** corresponds to master ID 0, **SI\_MErr[1]** corresponds to master ID 1, etc.). The slave will drive this signal for one clock cycle for each error that is encountered while trying to complete the transfer.

The PLB will OR together all of the slave error inputs for each master.

**Note:** The width of the **SI\_MErr[0:n]** and **PLB\_MErr[0:n]** signals is determined by the number of masters supported by the particular PLB based system.

## PLB\_MWrDAck[n] (PLB Master Write Data Acknowledge)

This signal is driven by the slave, back to the controlling master, for a write transfer to indicate that the data currently on the **PLB\_wrDBus[0:63]** bus is no longer required by the slave (that is, the slave has either already latched the data or will latch the data at the end of the current clock cycle). For a primary write request, the slave can begin to assert the **SI\_wrDAck** signal in the clock cycle **SI\_addrAck** is asserted. For a secondary write request, the slave can begin to assert the **SI\_wrDAck** signal in the clock cycle immediately following the assertion of **PLB\_wrPrim**.

## PLB\_MWrBTerm[n] (PLB Master Write Burst Terminate)

This signal is asserted by the slave to indicate that the current burst write transfer in progress must be terminated by the master. The slave can assert this signal with **SI\_addrAck**, or during any clock cycle thereafter, up to and including the clock cycle in which **SI\_wrComp** is asserted for the current transfer. In response to the assertion of this signal, the master is required to negate its **M\_wrBurst[n]** signal in the following clock cycle. The **M\_wrBurst[n]** signal is then sampled by the slave and when detected negated, the burst write transfer will then complete and the slave will assert the **SI\_wrComp** signal.

## PLB\_MRdDBus[n\*64:n\*64+63] (PLB Master Read Data Bus)

This 64-bit data bus is used to transfer data between a slave and a master during a PLB read transfer.

For a primary read request, the slave can begin to drive data on **SI\_rdbus[0:63]** two clock cycles following the assertion of **SI\_addrAck**. For a secondary read request, the slave can begin to drive data on **SI\_rdbus[0:63]** two cycles following the assertion of **PLB\_rdPrim**. In both cases, the slave can drive data on **SI\_rdbus[0:63]** through one clock cycle following the assertion of **SI\_rdComp**.

**Note:** Since the PLB read data bus is a shared bus, **SI\_rdbus[0:63]** must be driven low (that is, with logic 0s) by the slave any time that the slave is not selected for a read transfer or **Reset** is asserted.

## PLB\_MRdWdAddr[n\*4:n\*4+3] (PLB Master Read Word Address)

These signals are driven by the slave and, during a read line transfer, indicate the word address of the transferred data word within the line of data. Masters will sample these signals in the clock cycle that **SI\_rdDAck** is asserted for a read line transfer.

### Notes:

1. Since the PLB read data bus is a shared bus, **SI\_rdWdAddr[0:3]** must be driven low (that is, with logic 0s) by the slave any time that the slave is not selected for a read transfer or **Reset** is asserted. If selected for a read transfer the slave can begin to drive the **SI\_rdWdAddr[0:3]** signals with non-zero logic values two clock cycles after it has asserted the **SI\_addrAck** signal, through one clock cycle following the assertion of **SI\_rdComp**.
2. For line transfers that time out, the arbiter will complete the handshaking with the master. The arbiter core will respond as a 32-bit slave and provide all read word addresses being requested. The arbiter will not provide target word first response in this case. The order will start with word address 1 and increment to line size-1 and then provide word address 0.

## PLB\_MRdDAck[n] (PLB Master Read Data Acknowledge)

This signal is driven by the slave to indicate that the data on the **SI\_rdbus[0:63]** bus is valid and must be latched at the end of the current clock cycle. For a primary read request, the slave can begin to assert the **SI\_rdDAck** signal two clock cycles following the assertion of **SI\_addrAck**. For a secondary read request, the slave can begin to assert the **SI\_rdDAck** signal two clock cycles following the assertion of **PLB\_rdPrim**.

**Note:** This signal must be driven by the slave to a low value any time that the slave is not selected or the slave is selected but not ready to transfer read data on the read data bus.

## PLB\_MRdBTerm[n] (PLB Master Read Burst Terminate)

This signal is asserted by the slave to indicate to a master that the current burst read transfer in progress must be terminated. This signal can be asserted by the slave starting one clock cycle following the assertion of **SI\_addrAck**, up to and including the clock cycle in which **SI\_rdComp** is asserted. This signal must only be asserted for one clock cycle per termination request. In response to the assertion of this signal, the master is required to negate its **M\_rdBurst[n]** signal in the following clock cycle. The **M\_rdBurst[n]** signal is then sampled by the slave and when detected negated, the slave will supply one, and only one, additional piece of data. The transfer is then completed when the **SI\_rdDAck** signal is asserted while the PLB burst signal is negated.

## PLB\_Msize[n\*2:n\*2+1] (PLB Master Slave Size)

This is the OR'ed **SI\_ssize[0:1]** from all of the slave devices routed through the PLB arbiter and to the requesting master as **PLB\_Msize[n\*2:n\*2+1]**. This signal is an output of all non-32-bit PLB slaves. This signal is activated by the slave with the assertion of **PLB\_PValid** or **PLB\_SValid** and a valid slave address decode and must remain negated at all other times. This signal is only sampled by the master in the clock cycle in which the **PLB\_MAddrAck[n]** signal is asserted. A 64-bit slave device will drive a value of 2'b01 onto **SI\_ssize[0:1]**, 32-bit slave devices will have its **SI\_ssize[0:1]** signal tied to logic 0 (2'b00).

**Note:** A 64-bit slave device can respond as a 32-bit device if it chooses. It might decide to do this based on its resources available or for a particular optimization.

## M\_request[n] (Master Request)

This signal is asserted by the master to request a data transfer across the PLB. Once **M\_request[n]** has been asserted, this signal, the address, and all of the transfer qualifiers must retain their values until one of the following conditions occur:

1. The slave has terminated the address cycle through the assertion of **SI\_addrAck** (**PLB\_MAddrAck[n]**) or **SI\_rearbitrate** (**PLB\_MRearbitrate[n]**)
2. The master has aborted the request through the assertion of **M\_abort[n]**
3. The PLB arbiter has asserted **PLB\_MAddrAck[n]** in the event of a time-out

Once the address cycle has been properly terminated, the master can continue to assert **M\_request[n]** if another transfer is required across the PLB. In this case, the master address and transfer qualifiers will be updated in the clock cycle following the assertion of **PLB\_MAddrAck[n]**, **PLB\_MRearbitrate[n]**, or **M\_abort[n]**, to reflect the new request. If there are no other master requests pending, **M\_request[n]** should be negated in the clock cycle following the assertion of **PLB\_MAddrAck[n]**, **PLB\_MRearbitrate[n]**, or **M\_abort[n]**.

This signal must be negated in response to the assertion of **Reset**.

## M\_abort[n] (Master Abort)

This signal is asserted by the master to indicate that it no longer requires the data transfer it is currently requesting. This signal is only valid while the **M\_request[n]** signal is asserted and can only be used to abort a request that has not been acknowledged or is being acknowledged in the current clock cycle. In the clock cycle following the assertion of **M\_abort[n]**, the master should either negate **M\_request[n]** or make a new request. However, starting in the clock cycle following the assertion of **SI\_addrAck** by the slave, a request can no longer be aborted by the master and the slave is required to perform the necessary handshaking to complete the transfer. This signal will have a minimum amount of set-up time to allow for its assertion late in the clock cycle.

### Notes:

1. A slave can assert **SI\_wrDAck** and **SI\_wrComp** with **SI\_addrAck** for a primary write request, even if the request is being aborted by the master in the same clock cycle. In this case, the master is required to ignore these signals and no data will be stored by the slave.



2. If **SL\_rearbitrate** is asserted in the same clock cycle as **M\_abort[n]**, the PLB arbiter will ignore the **SL\_rearbitrate** signal and the master will not be “backed-off” during rearbitration. In the clock cycle following the assertion of **M\_abort[n]**, the PLB arbiter will rearbitrate and grant the highest priority request.

The **PLB\_abort** signal is sampled by the slaves only while **PLB\_PValid** or **PLB\_SValid** is asserted.

## M\_priority[n\*2:n\*2+1] (Master Priority)

These signals are driven by the master to indicate to the PLB arbiter the priority of the master’s request and are valid any time the **M\_request[n]** signal is asserted.

**Note:** It is permissible for the value of the **M\_priority[n\*2:n\*2+1]** signals to change at any time during the address cycle and prior to the slave asserting **SL\_addrAck** or **SL\_rearbitrate**, or the master aborting the request through **M\_abort[n]**.

The PLB arbiter uses these signals in conjunction with the other master priority signals to determine which request should be granted and then presented to the PLB slaves.

**Table 13-3** shows **M\_priority[n\*2:n\*2+1]** request priority level.

**Table 13-3: M\_priority[n\*2:n\*2+1] Request Priority Level**

| <b>M_priority[n*2:n*2+1]</b> | <b>Priority Level</b> |
|------------------------------|-----------------------|
| 11                           | Highest               |
| 10                           | Next highest          |
| 01                           | Next highest          |
| 00                           | Lowest                |

## M\_busLock[n] (Master Bus Lock)

This signal is asserted by the master with the **M\_request[n]** signal and is sampled by the PLB arbiter in the clock cycle in which the **SL\_addrAck** signal is asserted by the slave. This signal can be used by the current master to lock the arbitration and force the PLB arbiter to continue to grant the bus to that master and ignore all other requests that are pending. The PLB can only be locked by requesting a transfer with the **M\_busLock[n]** signal asserted and being the highest priority request presented to the PLB arbiter.

Once the bus has been successfully locked by the current master, it is not necessary for that master to continuously drive the request signal asserted. If the master negates **M\_request[n]**, but does not negate **M\_busLock[n]**, the bus will continue to be locked to that master and will remain locked until the master negates **M\_busLock[n]**. More specifically, the bus will continue to be locked with the current master until that master has negated its **M\_busLock[n]** signal for one complete clock cycle. On the clock cycle following the negation of the **M\_busLock[n]** signal, if there are no transfers in progress, the PLB arbiter will again rearbitrate and grant the bus to the highest priority request.

**Note:** A master request with the **M\_busLock[n]** signal asserted is a special case in that the PLB arbiter will wait for both the read data bus and the write data bus to be available prior to granting the PLB to a master and presenting that master’s address and transfer qualifiers to the slaves. Please refer to **PLB\_PValid (PLB Primary Address Valid)**, [page 1477](#) and **PLB\_SValid (PLB Secondary Address Valid)**, [page 1477](#) for more detailed information on the PLB arbiter’s handling of a master request with the **M\_busLock[n]** signal asserted.

This signal must be negated in response to the assertion of **Reset**.

## M\_RNW[n] (Master Read Not Write)

This signal is driven by the master and is used to indicate whether the request is for a read or a write transfer. If **M\_RNW[n] = 0b1**, the request is for the slave to supply data to be read into the master. If **M\_RNW[n] = 0b0**, the request is for the master to supply data to be written to the slave.



### M\_BE[n\*8:n\*8+7] (Master Byte Enable)

These signals are driven by the master. For a non-line and non-burst transfer they identify which bytes of the target word being addressed on **M\_ABus[n\*32:n\*32+31]** are to be read from or written to. These signals are routed from the controlling master through the PLB arbiter to the slaves. The **M\_BE** signals can optionally indicate the number of transfers that the master is requesting for fixed-length burst transfers. The burst length refers to the number of transfers of the data type selected by the **M\_size** signals. The **M\_size** = 1000 and **M\_BE** = 1111 will transfer sixteen bytes, **M\_size** = 1001 and **M\_BE** = 1111 will transfer sixteen halfwords, and **M\_BE** = 1111 and **M\_size** = 1010 will transfer sixteen words.

See the *Processor Local Bus Architecture Specifications* for a full description of all possible BE combinations for either partial word transfers or predefined burst length transfers. The slave interface signal description **PLB\_BE[0:7] (PLB Byte Enable)**, page 1479 of this document also gives the BE combinations.

Masters that do not implement a fixed-length transfer should drive all zeros on the BE signals during burst transfers to be compatible with slaves that have implemented this feature. Slaves that do not implement the fixed-length transfer will ignore the **PLB\_BE** signals during a burst transfer and will continue bursting until the **PLB\_rdBurst** or **PLB\_wrBurst** signal is negated by the master.

### M\_size[n\*4:n\*4+3] (Master Transfer Size)

The **M\_size[n\*4:n\*4+3]** signals are driven by the master to indicate the size of the requested transfer. A 0b0000 value indicates that the request is to read/write one to four bytes starting at the target address. The number of bytes to be read will be indicated on the **M\_BE[n\*8:n\*8+7]** signals. Refer to the *Processor Local Bus Architecture Specifications* for a full description of all possible size combinations.

### M\_type[n\*3:n\*3+2] (Master Transfer Type)

These signals are driven by the master and are used to indicate the type of transfer being requested. Most PLB transfers will use type 0b000, which are memory transfers.

This transfer type is used to read or write data from or to a device in the memory address space. Each PLB slave should decode the address on the **PLB\_ABus[0:31]** to determine if the transfer is to/from the memory area that is controlled by the slave.

Please refer to the *Processor Local Bus Architecture Specifications* for a full description of all possible type combinations along with DMA transfer operations for detailed description of DMA transfers using the PLB).

### M\_compress[n] (Master Compress)

This signal is driven by the master to indicate whether or not the requested transfer is for a memory area containing compressed data. If the master is requesting a read data transfer and the **PLB\_compress** signal is asserted, then the master is indicating that the data corresponding to the requested address is compressed and the slave must decompress the data prior to transferring it back to the master. If the master is requesting a write data transfer and the **PLB\_compress** is asserted, then the master is indicating that the data corresponding to the requested address must be compressed and the slave must compress the data prior to writing it out to memory.

### M\_guarded[n] (Master Guarded)

This signal is driven by the master to indicate that the requested transfer can be for a non-well behaved memory region. This signal is a transfer qualifier and must be valid anytime the **M\_request[n]** signal is asserted. This signal is muxed through the PLB arbiter to the slaves. See the *Processor Local Bus Architecture Specifications* for more details.

### M\_ordered[n] (Master Ordered)

This signal is driven by the master for a write request to indicate whether or not the write transfer must be ordered. This signal is a transfer qualifier and must be valid anytime the **M\_request[n]** signal is asserted and the **M\_RNW[n]** signal is low (that is, logic 0) indicating a write transfer. PLB slaves should ignore the **PLB\_ordered** signal during read transfers.

### M\_lockErr[n] (Master Lock Error)

This signal is asserted by the master to indicate whether or not the slave must lock the Slave Error Address Register (SEAR) and the Slave Error Status Register (SESR) when an error is detected during the transfer. This signal is a transfer qualifier and must be valid anytime the **M\_request[n]** signal is asserted.

### M\_ABus[n\*32:n\*32+31] (Master Address Bus)

Each master is required to provide a valid 32-bit address when its request signal is asserted. The PLB will then arbitrate the requests and allow the highest priority master's address to be gated onto the **PLB\_ABus**. For non-line transfers, this 32-bit bus indicates the lowest numbered byte address of the target data to be read/written over the PLB. The **M\_BE[n\*8:n\*8+7]** signals will indicate which bytes of the word are to be read or written for this transfer. (See **M\_BE[n\*8:n\*8+7] (Master Byte Enable)**, [page 1472](#) for a more detailed description of the **M\_BE** signals)

For line read transfers, the address bus can indicate the target byte address within the line of data that is being requested by the master. For line write transfers, the line word address must be zero since line writes transfers are required to be performed in sequential order across the PLB starting with the first word of the line. [Table 13-4](#) indicates the bits of the **M\_ABus** which must be zeroed for line write transfers. The slave must latch the address at the end of the clock cycle in which it asserts **Sl\_addrAck**.

Table 13-4: PLB Address Bus Signal Bits

| Line Size    | Line address         | Word Address                   | Byte Address            |
|--------------|----------------------|--------------------------------|-------------------------|
| 4-word line  | M_ABus[n*32:n*32+27] | M_ABus[n*32+28:n*32+29] = 00   | M_ABus[n*32+30:n*32+31] |
| 8-word line  | M_ABus[n*32:n*32+26] | M_ABus[n*32+27:n*32+29] = 000  | M_ABus[n*32+30:n*32+31] |
| 16-word line | M_ABus[n*32:n*32+25] | M_ABus[n*32+26:n*32+29] = 0000 | M_ABus[n*32+30:n*32+31] |
| Line Size    | Line address         | Word Address                   | Byte Address            |

### M\_wrDBus[n\*64:n\*64+63] (Master Write Data Bus)

This 64-bit data bus is used to transfer data between a master and a slave during a PLB write transfer. For a primary write request, the master must place the first bytes of data to be written on the **M\_wrDBus[n\*64:n\*64+63]** bus in the same clock cycle **M\_request[n]** is first asserted. For a secondary write request, the master must place the first bytes of data to be written on **M\_wrDBus[n\*64:n\*64+63]** in the clock cycle immediately following the last data acknowledge, (**PLB\_wrPrim**), for the primary write request.

### M\_wrBurst[n] (Master Write Burst)

This signal is driven by the master to control the length of a burst write transfer. A burst write of sequential bytes, halfword, or words can be requested by a master on the PLB by indicating a transfer size of 0b1000, 0b1001, or 0b1010, respectively, and a low value on the **M\_RNW[n]** signal.

#### Notes:

1. In the case of a master requesting two back-to-back write burst requests, where the second request is acknowledged prior to all the data being transferred for the first request (that is, the

second request is a secondary write request), the master must guarantee that **M\_wrBurst[n]** is reasserted for the second request in the clock cycle immediately following the last data acknowledge for the first request.

2. In the case of a burst of one the master does not assert **M\_wrBurst[n]**.

This signal must be negated in response to the assertion of **Reset**.

### M\_rdBurst[n] (Master Read Burst)

This signal is driven by the master to control the length of a burst read transfer. APLB master can request a burst read of sequential bytes (up to octalwords) by indicating a transfer size (Table 13-5) and a high value on the **M\_RNW[n]** signal.

Table 13-5: Read Burst Size

| PLB_size[0:3] | Burst Size  |
|---------------|-------------|
| 1000          | Bytes       |
| 1001          | Halfwords   |
| 1010          | Words       |
| 1011          | Doublewords |
| 1100          | Quad words  |
| 1101          | Octal words |

The burst transfer address must be aligned on a boundary of the requested burst size. For example, a halfword burst request must start on an even (halfword) address alignment.

Once a read burst transfer has been acknowledged by a slave, the slave will start sampling the **PLB\_rdBurst** signal in the clock cycle following the assertion of **Sl\_addrAck**, or **PLB\_rdPrim** in the case of a read burst transfer acknowledged as a secondary request, to determine when to terminate the transfer. A high value indicates that the master requires additional sequential transfers of data. A low value indicates the master requires one, and only one, additional transfer of data.

Once the first data transfer has been completed for a burst request, the slave must sequentially increment its address for each of the following data transfers and continue to do so until the **PLB\_rdBurst** signal is sampled negated. In the clock cycle the **PLB\_rdBurst** signal is sampled negated, the slave will also sample its **Sl\_rdDack** signal. If asserted, the slave will terminate the transfer by asserting its **Sl\_rdComp** signal in the following clock cycle or in a later clock cycle. If negated, the slave will supply one, and only one, additional data transfer in the following clock cycle (or in a later clock cycle, depending on the number of wait states) and terminate the transfer by asserting its **Sl\_rdComp** signal with **Sl\_rdDack** or in a later clock cycle.

#### Notes:

1. In the case of the same master requesting two back-to-back read burst requests, where the second request is acknowledged prior to all the data being transferred for the first request (that is, the second request is a secondary read request), the master must guarantee that **M\_rdBurst[n]** is negated during the last data transfer for the first request (that is, the primary read request) before this signal is reasserted for the second request. Furthermore, for the second request, this signal must be asserted in the clock cycle immediately following the last data acknowledge for the first request. See *Processor Local Bus Architecture Specifications* for further details.
2. It is not permissible for the master to assert the **M\_rdBurst[n]** signal until one cycle following the assertion of **Sl\_addrAck** and this signal can only be asserted during read burst transfers and must remain negated for all other transfer types.
3. Slaves should decode all values of Burst Size and provide their full data bus width of data per transfer. See *Processor Local Bus Architecture Specifications* for further details.
4. In the case of a burst of one the master will not assert **M\_rdBurst[n]**.

This signal must be negated in response to the assertion of **Reset**.

## M\_msize[n:n+1] (Master Size)

These signals are inputs to the PLB core for each master port and indicate the data bus width of the associated master. These signals need not be outputs of the master as the inputs to the PLB can be tied if the master will always drive its size to a particular value. The **M\_size** signals are valid with the master's active request for the duration of **PLB\_PValid** or **PLB\_SValid**. The **PLB\_msize[0:1]** signals are inputs to the 64-bit PLB slaves and specify the data bus width of the current valid master request being broadcast to the PLB slaves. Note that it is possible to change the master size with each request.

Table 13-6: M\_msize[n:n+1] Master Size

| M_msize[n:n+1] | Master Data Bus Size |
|----------------|----------------------|
| 00             | 32 bits              |
| 01             | 64 bits              |
| 10             | 128 bits             |
| 11             | 256 bits             |

## Slave Interface

Figure 13-4 demonstrates all slave interface I/O signals followed by detailed functional description. Refer to the *Processor Local Bus Architecture Specifications* for additional information.

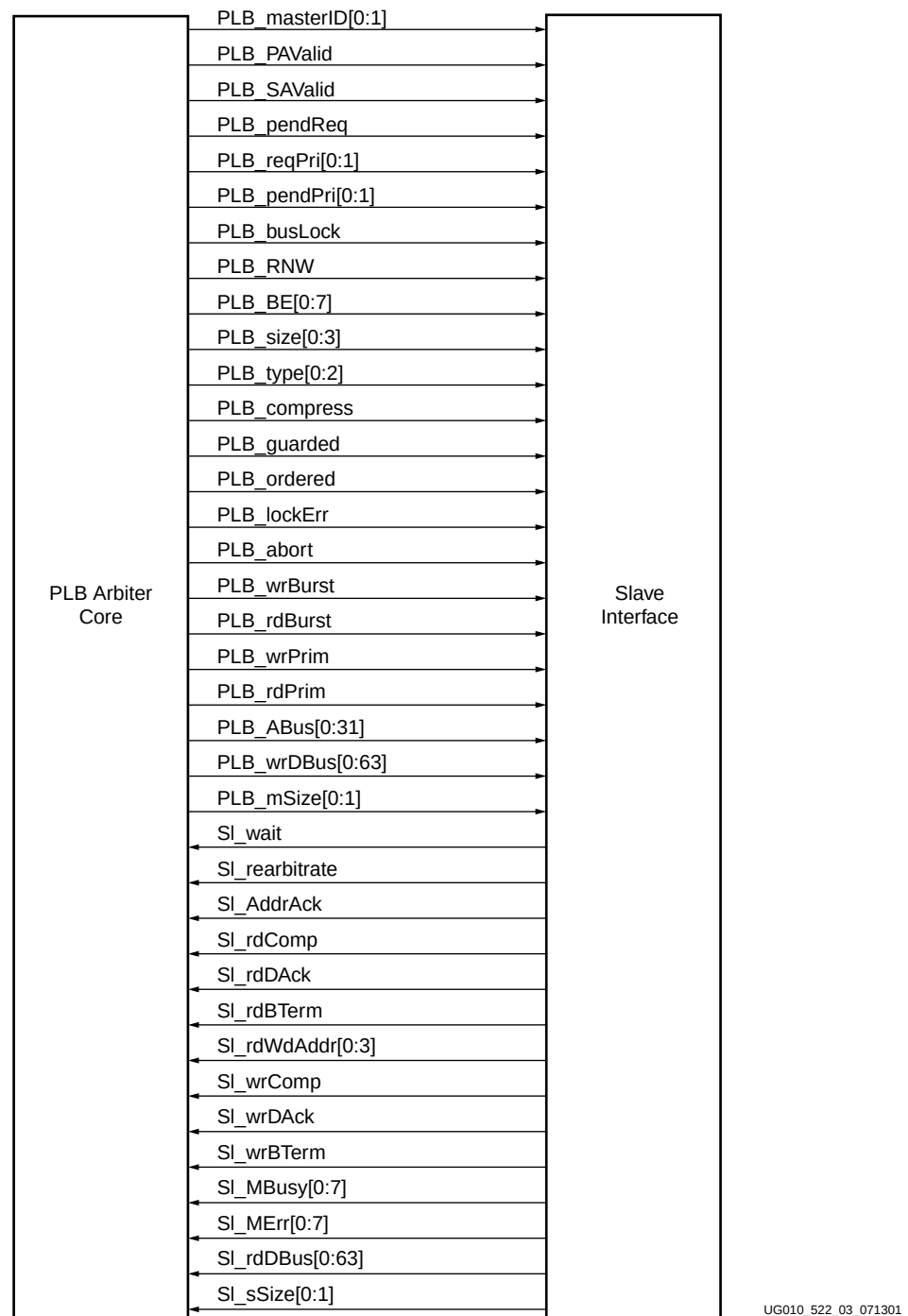


Figure 13-4: Slave Interface

### PLB\_masterID[0:2] (PLB Master ID)

These signals are driven by the PLB arbiter and are valid in any clock cycle in which **PLB\_PValid** or **PLB\_SValid** is asserted. These signals indicate to the slaves the identification of the master of the current transfer. The slave must use these signals to

determine which master busy signal and master signal should be driven on the PLB bus. The master ID can also be latched by the slave in an error syndrome register to indicate which master request caused the error.

### PLB\_PAVValid (PLB Primary Address Valid)

This signal is asserted by the PLB arbiter in response to the assertion of **M\_request[n]** and to indicate that there is a valid primary address and transfer qualifiers on the PLB outputs.

All slaves should sample the **PLB\_PAVValid** signal and if asserted, and the address is within their address range and they are capable of performing the transfer, they should respond by asserting their **SI\_addrAck** signal. If a slave detects a valid primary address on the PLB but is unable to latch the address and transfer qualifiers, or perform the requested transfer, then it should either assert the **SI\_wait** signal to require the PLB arbiter to wait for the request to be properly terminated, or assert the **SI\_rearbitrate** signal to require the PLB arbiter to rearbitrate the bus.

#### Notes:

1. Once **PLB\_PAVValid** has been asserted, the PLB arbiter will wait for sixteen clock cycles for the request to be properly terminated. If no slave responds with **SI\_wait**, **SI\_AddrAck**, or **SI\_rearbitrate**, or the request is not aborted by the master, by the sixteenth clock cycle, the PLB arbiter will time out and complete the necessary handshaking to the master as well as assert the **PLB\_MErr** signal (see the *Processor Local Bus Architecture Specifications* for more detailed information on the handshaking provided by the PLB arbiter).
2. Once the **PLB\_PAVValid** signal is asserted, the PLB arbiter will not rearbitrate the bus until either **SI\_rearbitrate**, or **SI\_addrAck**, or **M\_abort[n]**, is asserted, or the PLB arbiter times out.
3. Once a slave has asserted the **SI\_addrAck** signal the PLB arbiter will wait indefinitely for the slave to assert the read or write complete signal. It is up to the slave design to ensure that a deadlock does not occur on the bus due to an address acknowledge occurring without the corresponding data acknowledge(s).

The **PLB\_PAVValid** signal must be negated in response to the assertion of **Reset**.

### PLB\_SAVValid (PLB Secondary Address Valid)

This signal is asserted by the PLB arbiter to indicate to a PLB slave that there is a valid secondary address and transfer qualifiers on the PLB outputs.

Once **PLB\_SAVValid** has been asserted for a secondary read request, the PLB arbiter will wait indefinitely for any one of the following conditions to occur:

1. **SI\_addrAck** is asserted by a slave.
2. The request is aborted by the requesting master.
3. **SI\_rdComp** is asserted for the primary read request.

In the first and second case, the PLB arbiter will rearbitrate the bus in the following clock cycle. In the third case, the PLB arbiter will not rearbitrate the bus but instead will negate **PLB\_SAVValid** and assert **PLB\_PAVValid** to indicate that there is a new valid primary address and transfer qualifiers on the PLB outputs. **PLB\_SAVValid** will be deasserted and **PLB\_PAVValid** will be asserted in the clock cycle following the assertion of **SI\_rdComp**.

Once **PLB\_SAVValid** has been asserted for a secondary write request, the PLB arbiter will wait indefinitely for any one of the following conditions to occur:

1. **SI\_addrAck** is asserted by a slave.
2. The request is aborted by the requesting master.
3. **SI\_wrComp** is asserted for the primary write request.

In the first and second cases, the PLB arbiter will rearbitrate the bus in the following clock cycle. In the third case, the PLB arbiter will not rearbitrate the bus but instead will negate the **PLB\_SAVValid** signal and assert the **PLB\_PAVValid** signal, in one cycle following the assertion of **SI\_wrComp**, to indicate that there is a new valid primary address and transfer qualifiers on the PLB outputs.

**Note:** It is not possible for a secondary request to time-out on the PLB. Accordingly, if a slave detects a valid secondary address on the PLB but is unable to latch the address and transfer qualifiers, or perform the requested transfer, it is not necessary for the slave to assert **SI\_wait** or **SI\_rearbitrate** signals since these will be ignored by the PLB arbiter until the secondary request has become a primary request, **PLB\_SAVValid** has been negated, and **PLB\_PAVValid** has been asserted.

This signal must be negated in response to the assertion of **Reset**.

### PLB\_pendReq (PLB Request Pending)

This signal is asserted by the PLB arbiter to indicate that a master has a request pending on the PLB. **PLB\_pendReq** is a combined logic OR of all the master request inputs. It can be sampled by any PLB master or slave and can be used by itself, or in conjunction with the **PLB\_pendPri[0:1]** signals, to determine when to negate the **M\_busLock[n]** or **M\_rdBurst[n]** and **M\_wrBurst[n]** signals due to another master requesting the bus.

The **PLB\_pendReq** signal is always valid and will not be negated during a clock cycle in which a request is being aborted by the master.

### PLB\_abort (PLB Abort)

This signal is asserted by the master to indicate that it no longer requires the data transfer it is currently requesting. **PLB\_abort** is only valid while the **M\_request[n]** signal is asserted and can only be used to abort a request which has not been acknowledged or is being acknowledged in the current clock cycle. In the clock cycle following the assertion of **M\_abort[n]**, the master should either negate **M\_request[n]** or make a new request. However, starting in the clock cycle following the assertion of **SI\_addrAck** by the slave, a request can no longer be aborted by the master and the slave is required to perform the necessary handshaking to complete the transfer. This signal has a minimum amount of set-up time to allow for its assertion late in the clock cycle.

The **PLB\_abort** signal is sampled by the slaves only while **PLB\_PAVValid** or **PLB\_SAVValid** is asserted.

### PLB\_reqPri[0:1] (PLB Request Priority)

These signals are driven by the PLB arbiter and are valid any time the **PLB\_pendReq** signal is asserted. The **PLB\_reqPri** signals indicate the priority of the current request that the PLB arbiter has granted and is gating to the slaves. This priority will remain valid from the clock cycle that **PLB\_PAVValid** or **PLB\_SAVValid** is asserted, until the clock cycle in which the request has been acknowledged by the slave. These signals can also be used by slaves to resolve arbitration when requesting access to other buses.

### PLB\_pendPri[0:1] (PLB Pending Priority)

These signals are driven by the PLB arbiter and are valid any time the **PLB\_pendReq** signal is asserted. The **PLB\_pendPri** signals indicate the highest priority of any pending request input from all masters attached to the PLB. Only the priority inputs of masters with their respective **M\_request[n]** signals asserted can be used in determining the **PLB\_pendPri** outputs. These signals can be used by masters to determine when to negate the **M\_busLock[n]** or **M\_rdBurst[n]** and **M\_wrBurst[n]** signals due to another master requesting a higher priority request. Slaves can also use these signals.

### PLB\_busLock (PLB Bus Lock)

This signal is asserted by the master with the **M\_request[n]** signal and is sampled by the PLB arbiter in the clock cycle in which the **SI\_addrAck** signal is asserted by the slave. It can be used by the current master to lock the arbitration and force the PLB arbiter to continue to grant the bus to that master and ignore all other requests that are pending. The PLB can only be locked by requesting a transfer with the **M\_busLock[n]** signal asserted and being the highest priority request presented to the PLB arbiter.



## PLB\_RNW (PLB Read Not Write)

This signal is driven by the master and is used to indicate whether the request is for a read or a write transfer. If **M\_RNW[n]** = 0b1, the request is for the slave to supply data to be read into the master. If **M\_RNW[n]** = 0b0, the request is for the master to supply data to be written to the slave.

## PLB\_BE[0:7] (PLB Byte Enable)

These signals are driven by the master. For a non-line and non-burst transfer they identify which bytes of the target word being addressed on **M\_ABus[n\*32:n\*32+31]** are to be read from or written to. For a read transfer, the slaves should access the indicated bytes and place them on **Sl\_rdDBus[0:63]** in the proper memory alignment. These will then be steered to **PLB\_MRdDBus[n\*64:n\*64+63]**. For a write transfer, the slaves should only write out the indicated bytes from **M\_wrDBus[n\*64:n\*64+63]** to the external devices.

**Note:** The **M\_ABus[n\*32+29:n\*32+31]** must always address the left-most byte that is being transferred across the bus as shown in [Table 13-7](#).

Table 13-7: Byte Enable Signals<sup>(1)</sup>

| <b>M_BE[n*8:n*8+3]_M_BE[n*8+4:n*8+7]</b> | <b>Transfer Request</b>   | <b>M_ABus[n*32+29:n*32+31]</b> |
|------------------------------------------|---------------------------|--------------------------------|
| 1000_0000                                | Byte 0                    | 000                            |
| 0100_0000                                | Byte 1                    | 001                            |
| 0010_0000                                | Byte 2                    | 010                            |
| 0001_0000                                | Byte 3                    | 011                            |
| 0000_1000                                | Byte 4                    | 100                            |
| 0000_0100                                | Byte 5                    | 101                            |
| 0000_0010                                | Byte 6                    | 110                            |
| 0000_0001                                | Byte 7                    | 111                            |
| 1100_0000                                | Halfword 0, 1             | 000                            |
| 0110_0000                                | Unaligned halfword 1, 2   | 001                            |
| 0011_0000                                | Halfword 2, 3             | 010                            |
| 0001_1000                                | Unaligned halfword 3, 4   | 011                            |
| 0000_1100                                | Halfword 4, 5             | 100                            |
| 0000_0110                                | Unaligned halfword 5, 6   | 101                            |
| 0000_0011                                | Halfword 6, 7             | 110                            |
| 1110_0000                                | Bytes 0, 1, 2             | 000                            |
| 0111_0000                                | Bytes 1, 2, 3             | 001                            |
| 0011_1000                                | Bytes 2, 3, 4             | 010                            |
| 0001_1100                                | Bytes 3, 4, 5             | 011                            |
| 0000_1110                                | Bytes 4, 5, 6             | 100                            |
| 0000_0111                                | Bytes 5, 6, 7             | 110                            |
| 1111_0000                                | Word 0, 1, 2, 3           | 000                            |
| 0111_1000                                | Unaligned word 1, 2, 3, 4 | 001                            |



Table 13-7: Byte Enable Signals<sup>(1)</sup> (Continued)

| M_BE[n*8:n*8+3]_M_BE[n*8+4:n*8+7] | Transfer Request          | M_ABus[n*32+29:n*32+31] |
|-----------------------------------|---------------------------|-------------------------|
| 0011_1100                         | Unaligned word 2, 3, 4, 5 | 010                     |
| 0001_1110                         | Unaligned word 3, 4, 5, 6 | 011                     |
| 0000_1111                         | Word 4, 5, 6, 7           | 100                     |
| 1111_1000                         | Bytes 0, 1, 2, 3, 4       | 000                     |
| 0111_1100                         | Bytes 1, 2, 3, 4, 5       | 001                     |
| 0011_1110                         | Bytes 2, 3, 4, 5, 6       | 010                     |
| 0001_1111                         | Bytes 3, 4, 5, 6, 7       | 011                     |
| 1111_1100                         | Bytes 0, 1, 2, 3, 4, 5    | 000                     |
| 0111_1110                         | Bytes 1, 2, 3, 4, 5, 6    | 001                     |
| 0011_1111                         | Bytes 2, 3, 4, 5, 6, 7    | 010                     |
| 1111_1110                         | Bytes 0, 1, 2, 3, 4, 5, 6 | 000                     |
| 0111_1111                         | Bytes 1, 2, 3, 4, 5, 6, 7 | 001                     |
| 1111_1111                         | Doubleword                | 000                     |

**Notes:**

- For line transfers, the M\_BE[n\*8:n\*8+7] signals are ignored by the slave and the M\_size[n\*4:n\*4+3] signals are used to determine the number of bytes that are to be read or written.

For burst transfers, the M\_BE signals can optionally indicate the number of transfers that the master is requesting. The definition of the M\_BE signals during burst transfers is shown in Table 13-6.

Table 13-8: Byte Enable Signals During Burst Transfers

| M_BE[n*8:n*8+7] | Read Burst Length <sup>(1)</sup>                 |
|-----------------|--------------------------------------------------|
| 0000 0000       | Burst length determined by PLB_rd/wrBurst signal |
| 0000 0001       | Burst of 2                                       |
| 0000 0010       | Burst of 3                                       |
| 0000 0011       | Burst of 4                                       |
| 0000 0100       | Burst of 5                                       |
| 0000 0101       | Burst of 6                                       |
| 0000 0110       | Burst of 7                                       |
| 0000 0111       | Burst of 8                                       |
| 0000 1000       | Burst of 9                                       |
| 0000 1001       | Burst of 10                                      |
| 0000 1010       | Burst of 11                                      |
| 0000 1011       | Burst of 12                                      |
| 0000 1100       | Burst of 13                                      |

Table 13-8: Byte Enable Signals During Burst Transfers (Continued)

| <b>M_BE[n*8:n*8+7]</b> | <b>Read Burst Length<sup>(1)</sup></b> |
|------------------------|----------------------------------------|
| 0000 1101              | Burst of 14                            |
| 0000 1110              | Burst of 15                            |
| 0000 1111              | Burst of 16                            |

**Notes:**

1. The burst length refers to the number of transfers of the data type selected by the **M\_size** signals. The **M\_size** = 1000 and **M\_BE** = 1111 will transfer sixteen bytes, **M\_size** = 1001 and **M\_BE** = 1111 will transfer sixteen halfwords, and **M\_BE** = 1111 and **M\_size** = 1010 will transfer sixteen words.

## PLB\_size[0:3] (PLB Transfer Size)

The **M\_size[n\*4:n\*4+3]** signals are driven by the master to indicate the size of the requested transfer. **Table 13-9** defines all PLB transfer size signals.

**Table 13-9: PLB Transfer Size Signals**

| <b>M_size[n*4:n*4+3]</b> | <b>Definition</b>                                                                  |
|--------------------------|------------------------------------------------------------------------------------|
| 0000                     | Transfer one to four bytes of a word starting at the target address <sup>(1)</sup> |
| 0001                     | Transfer the 4-word line containing the target word <sup>(2)</sup>                 |
| 0010                     | Transfer the 8-word line containing the target word <sup>(2)</sup>                 |
| 0011                     | Transfer the 16-word line containing the target word <sup>(2)</sup>                |
| 0100                     | Reserved                                                                           |
| 0101                     | Reserved                                                                           |
| 0110                     | Reserved                                                                           |
| 0111                     | Reserved                                                                           |
| 1000                     | Burst transfer - bytes - length determined by master <sup>(3,4)</sup>              |
| 1001                     | Burst transfer - halfwords - length determined by master <sup>(3,4)</sup>          |
| 1010                     | Burst transfer - words - length determined by master <sup>(3,4)</sup>              |
| 1011                     | Burst transfer - doublewords - length determined by master <sup>(3,5)</sup>        |
| 1100                     | Burst transfer - quadwords - length determined by master <sup>(3,5)</sup>          |
| 1101                     | Burst transfer - octalwords - length determined by master <sup>(3,5)</sup>         |
| 1110                     | Reserved                                                                           |
| 1111                     | Reserved                                                                           |

### Notes:

1. A 0b0000 value indicates that the request is to read/write one to four bytes starting at the target address. The number of bytes to be read will be indicated on the **M\_BE[n\*8:n\*8+7]** signals.
2. For line read transfers, the target word may or may not be the first word transferred, depending on the design of the slave. For line read transfers, the **SI\_rdWdAddr[0:3]** signals will indicate the word that is being transferred. For line write transfers, words must always be transferred sequentially, starting with the first word of the line (that is, **M\_ABus[n\*32+28:n\*32+31] = 4b00XX** for a 4-word line write, **M\_ABus[n\*32+27:n\*32+31] = 5b000XX** an 8-word line write, and **M\_ABus[n\*32+26:n\*32+31] = 6b0000XX** for a 16-word line write). **M\_ABus[n\*32+30:n\*32+31]** should be ignored by the slave for line write transfers.
3. The **M\_BE[n\*8:n\*8+7]** signals are ignored for a burst transfer.
4. If **M\_size[n\*4:n\*4+3]** is 0b1000, 0b1001, or 0b1010, the request is to burst read or write bytes, halfwords, or words, respectively. The slave should start transferring data at the address indicated by the **PLB\_ABus[0:31]** and width as indicated by the size bits. The slave should then continue to read/write bytes, halfwords, or words, until the **M\_rdBurst[n]/M\_wrBurst[n]** signal is negated indicating that the master is no longer in need of additional data.
5. For PLB and devices supporting wider datapaths, doubleword, quadword, and octalword encoding are used to transfer 64 bits, 128 bits, and 256 bits, respectively. Slave should be designed to support all these transfer sizes, even though their data bus widths may be less than the requested transfer size. Slaves with data bus widths less than the requested burst transfer must provide/accept their full data bus widths. For example, when a 64-bit master requests a doubleword (64-bit) burst read transfer from a word (32-bit) slave, the slave should decode the type and must provide word (32-bit) data. The master is responsible for negating the **M\_rdBurst[n]/M\_wrBurst[n]** signal at the appropriate time to fulfill its transfer request.

## PLB\_type[0:2] (PLB Transfer Type)

These signals are driven by the master and are used to indicate the type of transfer being requested. [Table 13-10](#) defines all PLB transfer type signals.

**Table 13-10: PLB Transfer Type Signals**

| <b>M_type[n*3:n*3+2]</b> | <b>Definition</b>                                                    |
|--------------------------|----------------------------------------------------------------------|
| 000                      | Memory transfer                                                      |
| 001                      | DMA flyby transfer <sup>(1)</sup>                                    |
| 010                      | DMA buffered external peripheral transfer <sup>(2)</sup>             |
| 011                      | DMA buffered OPB peripheral transfer <sup>(2)</sup>                  |
| 100                      | PLB slave buffered memory to memory transfer <sup>(2)</sup>          |
| 101                      | PLB slave buffered peripheral to/from memory transfer <sup>(2)</sup> |
| 110                      | DMA buffered memory transfer <sup>(3)</sup>                          |
| 111                      | PLB slave buffered memory to memory transfer with sideband signals   |

**Notes:**

1. Must be used with **M\_size[n\*4:n\*4+3]** values of 0b0000 and 0b1000 - 0b1101 only.
2. Must be used with **M\_size[n\*4:n\*4+3]** values of 0b0000 only.
3. Slave not supporting DMA peripheral transfers must also decode **M\_type[n\*3:n\*3+2] = 0b110** as a memory transfer in order to support DMA buffered memory-to-memory transfers.

### Memory Transfers (M\_type = 0b000):

This transfer type is used to read or write data from or to a device in the memory address space. Each PLB slave should decode the address on the **PLB\_ABus[0:31]** to determine if the transfer is to or from the memory area that is controlled by the slave.

More details in DMA transfers using the PLB is available in the *Processor Local Bus Architecture Specifications*.

## PLB\_compress (PLB Compress)

This signal is driven by the master to indicate whether the requested transfer is for a memory area containing compressed data. If the master is requesting a read data transfer and the **PLB\_compress** signal is asserted, then the master is indicating that the data corresponding to the requested address is compressed and the slave must decompress the data prior to transferring it back to the master. If the master is requesting a write data transfer and the **PLB\_compress** is asserted, then the master is indicating that the data corresponding to the requested address must be compressed and the slave must compress the data prior to writing it out to memory.

## PLB\_guarded (PLB Guarded)

This signal is driven by the master to indicate that the requested transfer can be for a memory that is not well behaved. If a master is requesting a non-burst transfer (that is, **M\_size = 0nnn**), and the **PLB\_guarded** signal is negated, then the master is indicating that the 1K page of memory corresponding to the requested address is well behaved and that the slave can access all of the 1K page, but can stop at the 1K page boundary. If the master is requesting a non-burst transfer with the **PLB\_guarded** signal asserted, then the master is indicating that the 1K page of memory corresponding to the requested address might not be well behaved and the slave should restrict itself to access only exactly what was requested by the master.

## PLB\_ordered (PLB Ordered)

This signal is driven by the master for a write request to indicate whether or not the write transfer must be ordered. This signal is a transfer qualifier and must be valid anytime the **M\_request[n]** signal is asserted and the **M\_RNW[n]** signal is low (that is, logic 0) indicating a write transfer. PLB slaves should ignore the PLB\_ordered signal during read transfers.

When acknowledging a write request with the **M\_ordered[n]** signal asserted, the slave must not allow any subsequent requests (reads or writes) to get in between or ahead of the ordered write request. When acknowledging a write request with the **M\_ordered[n]** signal negated, the slave can decide to hold this request in a buffer and perform subsequent requests (reads or writes) prior to completing the unordered write request.

## PLB\_lockErr (PLB Lock Error)

This signal is asserted by the master to indicate whether or not the slave must lock the Slave Error Address Register (SEAR) and the Slave Error Status Register (SESR) when an error is detected during the transfer. If the value of this signal is low, the slave should not lock the SEAR and SESR when the error occurs. Instead, the error address and syndrome should be latched into the SEAR and SESR and not locked. If a subsequent error is detected by this transfer or any other transfer the values in the SEAR and SESR will be overwritten. If the value of this signal is high (that is, logic 1) the slave should lock the SEAR and SESR on the occurrence of any errors as a result of this transfer. Any errors that occur after the SEAR and SESR are locked will not override the values that were written with the first error. Once the SESR and SEAR registers are locked with an error, they will remain locked until software clears the SESR.

## PLB\_ABus[0:31] (PLB Address Bus)

Each master is required to provide a valid 32-bit address when its request signal is asserted. The PLB will then arbitrate the requests and allow the highest priority master's address to be gated onto the PLB\_ABus. For non-line transfers, this 32-bit bus indicates the lowest numbered byte address of the target data to be read or written over the PLB. The **M\_BE[n\*8:n\*8+7]** signals will indicate which bytes of the word are to be read or written for this transfer. (See **M\_BE[n\*8:n\*8+7] (Master Byte Enable)**, page 1472 for a more detailed description of the M\_BE signals)

For line read transfers, the address bus can indicate the target byte address within the line of data that is being requested by the master. Slaves can read the data in any order and can use the target address to optimize performance by transferring the target word first. For line write transfers, the line word address must be zero since line writes transfers are required to be performed in sequential order across the PLB starting with the first word of the line. Table 13-4 indicates the bits of the **M\_ABus** which must be zeroed for line write transfers. The slave must latch the address at the end of the clock cycle in which it asserts **Sl\_addrAck**.

## PLB\_wrDBus[0:63] (PLB Write Data Bus)

This 64-bit data bus is used to transfer data between a master and a slave during a PLB write transfer. For a primary write request, the master must place the first bytes of data to be written on the **M\_wrDBus[n\*64:n\*64+63]** bus in the same clock cycle **M\_request[n]** is first asserted. For a secondary write request, the master must place the first bytes of data to be written on **M\_wrDBus[n\*64:n\*64+63]** in the clock cycle immediately following the last data acknowledge for the primary write request.

Once a master has requested a write transfer it must begin to sample the **PLB\_MWrDAck[n]** signal continuously. Note that the master must then retain the data on **M\_wrDBus[n\*64:n\*64+63]** until the end of the clock cycle in which **PLB\_MWrDAck[n]** is sampled asserted.

For non-line, non-burst transfers, (that is,  $M\_size[n*4:n*4+3] = 0b0000$ ), the master must retain the data on  $M\_wrDBus[n*64:n*64+63]$  until the end of the clock cycle in which  $PLB\_MWrDAck[n]$  is first sampled asserted, at which time the master will consider the transfer to be complete.

For line write transfers, the master must retain the first word of data (that is, Word-0) on  $M\_wrDBus[n*64:n*64+63]$  until the end of the clock cycle in which  $PLB\_MWrDAck[n]$  is first sampled asserted. The master will then continue to place a new word of data (that is, Word-1, Word-2, etc.) on  $M\_wrDBus[n*64:n*64+63]$  every time  $PLB\_MWrDAck[n]$  is sampled asserted, until this signal is sampled asserted for the last word of the line, at which time the master will consider the transfer to be complete.

For burst write transfers, the master must retain the first byte, halfword, or word of data (that is, Data-0) on  $M\_wrDBus[n*64:n*64+63]$  until the end of the clock cycle in which  $PLB\_MWrDAck[n]$  is first sample asserted. The master will then continue to place a new byte, halfword, or word of data (that is, Data-1, Data-2, etc.) on  $M\_wrDBus[n*64:n*64+63]$  every time  $PLB\_MWrDAck[n]$  is sampled asserted, until the burst transfer is completed.

**Note:** In the case of byte and halfword write burst transfers, the master is required to place the write data on the correct memory-aligned byte or halfword of the PLB. For example, if the master is performing a byte burst starting with address 0, the master must put the first byte on  $M\_wrDBus[0:7]$ , the second byte on  $M\_wrDBus[8:15]$ , the third byte on  $M\_wrDBus[16:23]$ , etc.

### PLB\_wrBurst (PLB Write Burst)

This signal is driven by the master to control the length of a burst write transfer. A burst write of sequential bytes, halfword, or words can be requested by a master on the PLB by indicating a transfer size of 0b1000, 0b1001, or 0b1010, respectively, and a low value on the  $M\_RNW[n]$  signal.

When a write burst transfer is acknowledged by a slave, the slave will sample the  $PLB\_wrBurst$  signal during every clock cycle in which the  $SI\_wrDAck$  signal is asserted to determine when to terminate the burst transfer. A high value indicates that the master requires at least one additional sequential byte, halfword, or word of data. A low value indicates that the current transfer is the last sequential transfer that the master requires to write. Note that this signal can be asserted only during a burst write transfer and must remain negated at all other times.

The slave will complete the burst transfer by asserting the  $SI\_wrComp$  signal. Note that since it is permissible for the slave to assert the  $SI\_wrComp$  signal several clock cycles after the last data transfer clock cycle, the slave must ignore the  $PLB\_wrBurst$  signal once this signal has been negated and until  $SI\_wrComp$  has been asserted for the current burst transfer.

### PLB\_rdBurst (PLB Read Burst)

This signal is driven by the master to control the length of a burst read transfer. A burst read of sequential bytes, halfword, or words can be requested by a master on the PLB by indicating a transfer size of 0b1000, 0b1001, or 0b1010, respectively, and a high value on the  $M\_RNW[n]$  signal. Once a read burst transfer has been acknowledged by a slave, the slave will start sampling the  $PLB\_rdBurst$  signal in the clock cycle following the assertion of  $SI\_addrAck$ , or  $PLB\_rdPrim$  in the case of a read burst transfer acknowledged as a secondary request, to determine when to terminate the transfer. A high value indicates that the master requires additional sequential bytes, halfwords, or words of data. A low value indicates the master requires one, and only one, additional sequential byte, halfword, or word of data.

Once the first data transfer has been completed for a burst request, the slave must sequentially increment its address for each of the following data transfers and continue to do so until the  $PLB\_rdBurst$  signal is sampled negated. In the clock cycle the  $PLB\_rdBurst$  signal is sampled negated, the slave will also sample its  $SI\_rdDAck$  signal. If asserted, the slave will terminate the transfer by asserting its  $SI\_rdComp$  signal in the following clock cycle or in a later clock cycle. If negated, the slave will supply one, and only one, additional

byte, halfword, or word of data in the following clock cycle (or in a later clock cycle, depending on the number of wait states) and terminate the transfer by asserting its **SI\_rdComp** signal the clock cycle before **SI\_rdDack** or with **SI\_rdDack** or in a later clock cycle.

### PLB\_wrPrim (PLB Primary Write Request)

This signal is asserted by the PLB arbiter to indicate that a secondary write request can be considered a primary write request in the following clock cycle. Slaves supporting address pipelining must begin to sample this signal in the clock cycle that **PLB\_SAVValid** is asserted for a secondary write request. In the clock cycle following the assertion of **PLB\_wrPrim**, the PLB arbiter will gate the secondary write data onto the **PLB\_wrDBus**, provided the secondary write request has already been acknowledged, or it is currently being acknowledged and **M\_abort[n]** is not asserted. Accordingly, the slave can begin to assert its **SI\_wrDack** signal for a secondary write request in the clock cycle following the assertion of **PLB\_wrPrim**.

**Note:** If a secondary write request is either aborted or address acknowledged in the same clock cycle that **SI\_wrComp** is asserted for a primary write request, the **PLB\_wrPrim** signal will be asserted by the arbiter and should be ignored by the slaves.

### PLB\_rdPrim (PLB Primary Read Request)

This signal is asserted by the PLB arbiter to indicate that a secondary read request which has already been acknowledged by a slave, can now be considered a primary read request. Slaves supporting address pipelining must begin to sample this signal in the clock cycle following the assertion of **SI\_addrAck** in response to the assertion of **PLB\_SAVValid**. When transferring data for a secondary read request, the slave can begin to drive the **SI\_rdDBus[0:63]** bus two clock cycles after it has sampled the **PLB\_rdPrim** signal asserted.

**Note:** If there is not a secondary read request on the PLB or a secondary read request has not been acknowledged by a slave, then the **PLB\_rdPrim** signal will not be asserted.

### PLB\_MSize[0:1] (PLB Master Size)

These signals are inputs to the PLB core for each master port and indicate the data bus width of the associated master. These signals need not be outputs of the master as the inputs to the PLB can be tied to a particular value if the master will always respond as a particular bus size. The **M\_size** signals are valid with the master's active request for the duration of **PLB\_PAVValid** or **PLB\_SAVValid**. The **PLB\_msize[0:1]** signals are inputs to the 64-bit PLB slaves and specify the data bus width of the current valid master request being broadcast to the PLB slaves. Note that it is possible to change the master size with each request. (See [Table 13-9, page 1482](#).)

### SI\_addrAck (Slave Address Acknowledge)

This signal is asserted to indicate that the slave has acknowledged the address and will latch the address and all of the transfer qualifiers at the end of the current clock cycle. This signal is asserted by the slave only while **PLB\_PAVValid** or **PLB\_SAVValid** is asserted and should remain negated at all other times. Once a **SI\_addrAck** occurs then the PLB arbiter rearbitrates the bus.

**Note:** It is possible for the slave to acknowledge a valid address in the same clock cycle in which **PLB\_PAVValid** or **PLB\_SAVValid** are first asserted.

### SI\_rearbitrate (Slave Rearbitrate)

This signal is asserted to indicate that the slave is unable to perform the currently requested transfer and requires the PLB arbiter to rearbitrate the bus. This signal is asserted by the slave only while **PLB\_PAVValid** or **PLB\_SAVValid** is asserted and should remain negated at all other times.

When asserted in response to the assertion of **PLB\_PAVValid**, the PLB arbiter will pass this signal to the masters and rearbitrate and grant the bus to the highest priority request in the



next clock cycle. Furthermore, to avoid a possible deadlock scenario, the PLB arbiter will ignore the original master request during rearbitration.

When asserted in response to the assertion of **PLB\_SAVValid**, the PLB arbiter will not pass this signal to the masters or rearbitrate the bus. Instead, the PLB arbiter will wait for the current request to become a primary request, and **PLB\_PAVValid** to be asserted, and then sample **SI\_wait**, **SI\_addrAck**, **SI\_rearbitrate**, and **M\_abort[n]**.

**Notes:**

1. If **SI\_addrAck** and **SI\_rearbitrate** are sampled asserted in the same clock cycle, the PLB arbiter will ignore the **SI\_rearbitrate** signal and will not rearbitrate the bus. As a result, the slave must perform the requested data transfer (read or write) to avoid a possible deadlock scenario.
2. If the bus had been previously locked, the **SI\_rearbitrate** signal will be ignored by the PLB arbiter to prevent the interruption of an "atomic" operation. Hence, to prevent a deadlock scenario in this case, the locking master must negate its **M\_request[n]** and **M\_busLock[n]** signals for a minimum of two clock cycles following the sampling of **PLB\_MRearbitrate[n]** asserted.

## SI\_wait (Slave Wait)

This signal is asserted to indicate that the slave has recognized the PLB address as a valid address, but is unable to latch the address and all of the transfer qualifiers at the end of the current clock cycle. The slave can assert this signal anytime it recognizes a valid address and type on the PLB and it is not required to negate it before asserting **SI\_addrAck** or **SI\_rearbitrate**.

**Note:** The PLB arbiter will qualify the **SI\_wait** signal with **PLB\_PAVValid** and thus the slaves are not required to qualify the assertion of **SI\_wait** with **PLB\_PAVValid**.

When asserted in response to the assertion of **PLB\_PAVValid**, the PLB arbiter will use this signal to disable its address cycle time-out mechanism and wait indefinitely for the slave to assert its **SI\_addrAck** or **SI\_rearbitrate** signals. Otherwise, the PLB arbiter will wait a maximum of sixteen clock cycles for **SI\_addrAck** or **SI\_rearbitrate** to be asserted before timing out.

When asserted in response to the assertion of **PLB\_SAVValid**, the PLB arbiter will ignore the **SI\_wait** signal and wait indefinitely for the slave to assert its **SI\_addrAck** signal, or the master to abort the request, or for the secondary request to become a primary request.

The **SI\_wait** signal is an input to the PLB arbiter only, and is not driven back to the PLB masters.

## SI\_rdComp (Slave Read Complete)

This signal is driven by the slave and is used to indicate to the PLB arbiter that the read transfer is either already complete, or will be complete by the end of the next clock cycle. This signal is not driven back to the masters. In order to optimize performance on the PLB, the slave should assert this signal one clock cycle before the data acknowledge phase for the last data transfer cycle and thus allow the next read transfer to be overlapped with data being transferred on the PLB. If this is not possible, then this signal should be asserted in the same clock cycle as the last data transfer (for minimum latency) or in any clock cycle following the last data transfer. The assertion of this signal will cause the PLB arbiter to gate the next read request to the slaves in that clock cycle.

During read burst transfers, the **SI\_rdComp** signal will normally be asserted either in the clock cycle in which the last **SI\_rdDack** is asserted, or in a subsequent clock cycle. The **SI\_rdComp** signal can only be asserted in the cycle prior to the last **SI\_rdDack** during read burst transfers if the **PLB\_rdBurst** signal is negated by the master, or the **PLB\_rdBurst** signal is asserted by the master and the **SI\_rdBTerm** signal is also asserted by the slave.

**Note:** The assertion of the **SI\_rdComp** signal causes arbitration of the next request in the same clock cycle whereas assertion of **SI\_wrComp** causes arbitration of the next request in the following clock cycle.



## SI\_rdDAck (Slave Read Data Acknowledge)

This signal is driven by the slave to indicate that the data on the **SI\_rdDBus[0:63]** bus is valid and must be latched at the end of the current clock cycle. For a primary read request, the slave can begin to assert the **SI\_rdDAck** signal two clock cycles following the assertion of **SI\_addrAck**. For a secondary read request, the slave can begin to assert the **SI\_rdDAck** signal two clock cycles following the assertion of **PLB\_rdPrim**.

For single-beat read transfers, the signal is asserted for one clock cycle only. For line read transfers, the signal will be asserted for 4, 8, or 16 clock cycles. For burst read transfers, this signal will be asserted for as many clock cycles as the length of the burst requires, as indicated by the **M\_rdBurst[n]** signal.

**Note:** This signal must be driven by the slave to a logic 0 any time that the slave is not selected or the slave is selected but not ready to transfer read data on the read data bus.

## SI\_rdBTerm (Slave Read Burst Terminate)

This signal is asserted by the slave to indicate to a master that the current burst read transfer in progress must be terminated. This signal can be asserted by the slave starting one clock cycle following the assertion of **SI\_addrAck**, up to and including the clock cycle in which **SI\_rdComp** is asserted. This signal must only be asserted for one clock cycle per termination request. In response to the assertion of this signal, the master is required to negate its **M\_rdBurst[n]** signal in the following clock cycle. The **M\_rdBurst[n]** signal is then sampled by the slave and when detected negated, the slave will supply one, and only one, additional piece of data. The transfer is then completed when the **SI\_rdDAck** signal is asserted while the PLB burst signal is negated.

## SI\_rdDBus[0:63] (Slave Read Data Bus)

This 64-bit data bus is used to transfer data between a slave and a master during a PLB read transfer.

For a primary read request, the slave can begin to drive data on **SI\_rdDBus[0:63]** two clock cycles following the assertion of **SI\_addrAck**. For a secondary read request, the slave can begin to drive data on **SI\_rdDBus[0:63]** two cycles following the assertion of **PLB\_rdPrim**. In both cases, the slave can drive data on **SI\_rdDBus[0:63]** through one clock cycle following the assertion of **SI\_rdComp**.

Also for a primary read request, the master must begin to sample the **PLB\_MRdDAck[n]** signal two clock cycles following the assertion of **PLB\_MAddrAck[n]**. For a secondary read request, the master must begin to sample the **PLB\_MRdDAck[n]** signal in the clock cycle immediately following the last data acknowledge for the primary. In both cases, the master must latch the data on **PLB\_MRdDBus[n\*64:n\*64+63]** at the end of the clock cycle in which **PLB\_MRdDAck[n]** is sampled asserted.

For non-line read transfers, data must always transferred at the requested width. Byte, halfword, three byte, and fullword transfers are possible. However, in the case of a read transfer involving a slave device whose datapath width is smaller than the width of the requested PLB transfer, the slave must first accumulate all of the requested data internally and then perform the read data transfer on the PLB at the requested width.

**Note:** Since the PLB read data bus is a shared bus, **SI\_rdDBus[0:63]** must be driven low (that is, with logic 0s) by the slave any time that the slave is not selected for a read transfer or **Reset** is asserted.

## SI\_rdWdAddr[0:3] (Slave Read Word Address)

These signals are driven by the slave and, during a read line transfer, indicate the word address of the transferred data word within the line of data. Masters will sample these signals in the clock cycle that **SI\_rdBck** is asserted for a read line transfer.

Table 13-11: PLB Read Word Address Signals

| Line Transfer Size | PLB MRdWdAddr[n*4:n*4+3] |           |       |     |
|--------------------|--------------------------|-----------|-------|-----|
|                    | (0)                      | (1)       | (2)   | (3) |
| 4-word             | Undefined                | Undefined | Valid |     |
| 8-word             | Undefined                | Valid     |       |     |
| 16-word            | Valid                    |           |       |     |

**Note:** Since the PLB read data bus is a shared bus, **SI\_rdWdAddr[0:3]** must be driven low (that is, with logic 0s) by the slave any time that the slave is not selected for a read transfer or **Reset** is asserted. If selected for a read transfer the slave can begin to drive the **SI\_rdWdAddr[0:3]** signals with non-zero logic values two clock cycles after it has asserted the **SI\_addrAck** signal, through one clock cycle following the assertion of **SI\_rdComp**.

## SI\_wrComp (Slave Write Complete)

This signal is asserted by the slave to indicate the end of the current write transfer. It is asserted once per write transfer, either during the last beat of the data transfer or any number of clock cycles following the last beat of data transfer, but not before the last beat of the data transfer. The PLB arbiter uses this signal to allow a new write request to be granted in the following clock cycle.

**Note:** The slave can assert this signal in the same clock cycle that the request was granted if only one data transfer on the PLB is required and the address and data acknowledge signals are also asserted.

## SI\_wrDack (SLave Write Data Acknowledge)

This signal is driven by the slave for a write transfer to indicate that the data currently on the **PLB\_wrDBus[0:63]** bus is no longer required by the slave (that is, the slave has either already latched the data or will latch the data at the end of the current clock cycle). For a primary write request, the slave can begin to assert the **SI\_wrDack** signal in the clock cycle **SI\_addrAck** is asserted. For a secondary write request, the slave can begin to assert the **SI\_wrDack** signal in the clock cycle immediately following the assertion of **PLB\_wrPrim**.

For single-beat write transfers, the signal is asserted for one clock cycle only. For line write transfers, the signal will be asserted for 4, 8, or 16 clock cycles. For burst write transfers, this signal will be asserted for as many clock cycles as the length of the burst requires, as indicated by the **M\_wrBurst[n]** signal.

**Note:** This signal must be driven by the slave to a low value any time that the slave is not selected or the slave is selected but not ready to transfer write data on the write data bus.

## SI\_wrBTerm (Slave Write Burst Terminate)

This signal is asserted by the slave to indicate that the current burst write transfer in progress must be terminated by the master. The slave can assert this signal with **SI\_addrAck**, or during any clock cycle thereafter, up to and including the clock cycle in which **SI\_wrComp** is asserted for the current transfer. In response to the assertion of this signal, the master is required to negate its **M\_wrBurst[n]** signal in the following clock cycle. The **M\_wrBurst[n]** signal is then sampled by the slave and when detected negated, the burst write transfer will then complete and the slave will assert the **SI\_wrComp** signal.

**Note:** If the slave asserts the **SI\_wrBTerm** signal in the same clock cycle the master negates its **M\_wrBurst[n]** signal, no further response is required by the master.

### SI\_MBusy[0:7] (Slave Master Busy)

These signals are driven by the slave and are used to indicate that the slave is either busy performing a read or a write transfer, or has a read or write transfer pending. Each slave is required to drive a separate busy signal for each master attached on the PLB bus (i.e. **SI\_MBusy[0]** corresponds to master ID 0, **SI\_MBusy[1]** corresponds to master ID 1, etc.). The slave should latch the master ID and use this ID to drive the corresponding master busy signal until the data transfer has been completed.

### SI\_MErr[0:7] (Slave Error)

These signals are driven by the slave and are used to indicate that the slave has encountered an error during a transfer that was initiated by this master. Each slave is required to drive a separate error signal for each master attached on the PLB bus (that is, **SI\_MErr[0]** corresponds to master ID 0, **SI\_MErr[1]** corresponds to master ID 1 etc.). The slave will drive this signal for one clock cycle for each error that is encountered while trying to complete the transfer.

### SI\_ssize[0:1] (Slave Size)

The **SI\_ssize[0:1]** signals are outputs of all non-32-bit PLB slaves. 64-bit slave devices will drive a value of 2'b01 to indicate that they are a 64-bit device. A value of 2'b00 indicates the slave device is a 32-bit device or is responding as a 32-bit device.

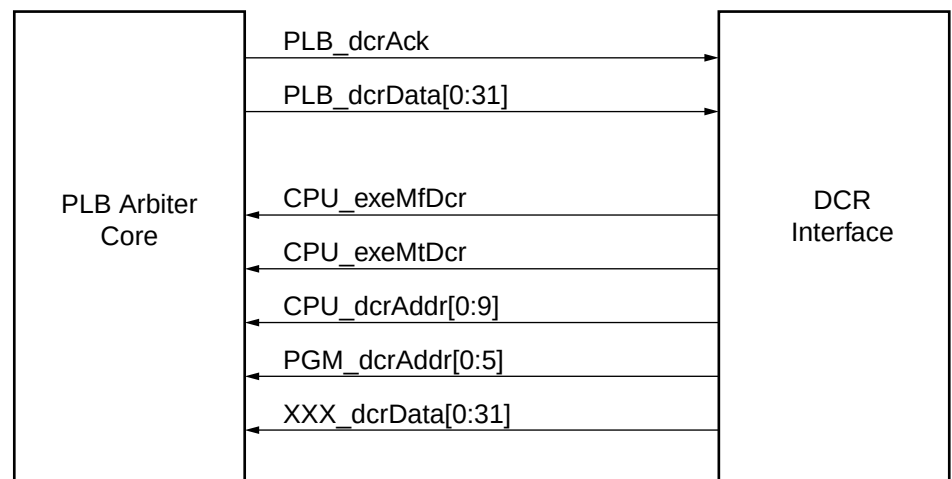
**Note:** Slave devices larger than 32-bits can respond as the size of the slave or as a smaller size. A slave might choose to do this based on the available buffer resources when it acknowledges a master request. (For example, a 64-bit slave might have two 32-bit write buffers that can be used to accept various transfer sizes. It could already have accepted a 32-bit write, then a 64-bit write request comes in and the slave decides to respond as a 32-bit slave forcing a conversion cycle. See the *Processor Local Bus Architecture Specifications* for details on conversion cycles.)

These signals are activated by the slave with the assertion of **PLB\_PValid** or **PLB\_SValid** and a valid slave address decode and must remain negated at all other times. They are routed through the PLB to the requesting master as **PLB\_Msize[n\*2:n\*2+1]**. These signals are only sampled by the master in the clock cycle in which the **PLB\_MAddrAck[n]** signal is asserted.

## DCR Interface

The device control register (DCR) interface allows the CPU core in the system to read and write the DCRs in the PLB arbiter. All DCRs are privileged for both read and write and are accessed by using the *move from device* control register (*mfdcr*) and *move to device* control register (*mtdcr*) instructions. For additional information on the DCR bus, see the *32-Bit Device Control Register Bus Architecture Specifications*.

Figure 13-5 demonstrates all DCR interface input/output signals followed by a detailed functional description.



UG010\_522\_05\_070501

Figure 13-5: DCR Interface

### PLB\_dcrAck (PLB DCR Acknowledge)

This is the DCR transfer acknowledge signal. This signal indicates that the PLB DCR slave logic has recognized **CPU\_exeMfDcr**/**CPU\_exeMtDcr** and the address on **CPU\_dcrAddr[0:9]**. This signal will be asserted once the DCR has been updated during a *mtdcr* operation or data has been placed on **PLB\_dcrData[0:31]** during a *mfdcr* operation. **PLB\_dcrAck** will be kept asserted as long as **CPU\_exeMfDcr** or **CPU\_exeMtDcr** is asserted.

Furthermore, **PLB\_dcrAck** will be negated in the cycle following the negation of **CPU\_exeMfDcr** or **CPU\_exeMtDcr**.

**Note:** The CPU will wait for **PLB\_dcrAck** to be negated for the current *mtdcr*/*mfdcr* operation, prior to starting a new *mtdcr*/*mfdcr* operation.

### PLB\_dcrData[0:31] (PLB DCR Data)

This is the CPU DCR read data bus. The PLB DCR slave logic will drive this bus with the contents of its selected DCR during a *mfdcr* operation as long as the address on **CPU\_dcrAddr[0:9]** matches the DCR's address and **CPU\_exeMfDcr** is asserted. Data must be valid when **PLB\_dcrAck** is asserted. When the PLB slave DCR is not selected and **PLB\_dcrAck** is not active, the DCR slave logic will bypass its DCR and drive this bus with **XXX\_dcrData[0:31]**.

### CPU\_exeMfDcr (Move From DCR)

This is the control signal for a DCR read operation. This signal is asserted by the CPU to indicate that a *move from device* control register (*mfdcr*) operation is in progress and that the address on **CPU\_dcrAddr[0:9]** is valid. Once this signal is asserted, the CPU will wait a maximum of 16 cycles for the **DCR\_cpuAck** signal to be asserted indicating that data is valid on **XXX\_dcrData[0:31]**. If **DCR\_cpuAck** is not asserted within 16 cycles, the CPU will time out and simply execute the next instruction. During reset this signal is held inactive.

### CPU\_exeMtDcr (Move To DCR)

This is the control signal for a DCR write operation. This signal is asserted by the CPU to indicate that a *move to device* control register (*mtdcr*) operation is in progress and that the address on **CPU\_dcrABus[0:9]** is valid. Once this signal is asserted, the CPU will wait a maximum of 16 cycles for the **DCR\_cpuAck** signal to be asserted indicating that the *mtdcr*

operation is complete. If **DCR\_cpuAck** is not asserted within 16 cycles, the CPU will time out and simply execute the next instruction. During reset this signal is held inactive.

### CPU\_dcrAddr[0:9] (DCR Address)

This is the DCR address bus. This bus is valid anytime **CPU\_exeMfDcr**, or **CPU\_exeMtDcr** is asserted, and its invalid at all other times. Hence, the DCR slave logic must always qualify this bus with **CPU\_exeMfDcr**/**CPU\_exeMtDcr**. The DCR address will remain stable during the *mtdcr* / *mfocr* operation.

### PGM\_dcrAddr[0:5] (Program DCR Address)

DCR address space locator. These signals are tied to either GND or VDD during the physical design phase. These signals are used by the PLB to determine if one of its DCRs is being accessed during a *mtdcr* or *mfocr* operation.

### XXX\_dcrData[0:31] (DCR Data In)

This is the DCR write data bus. This bus is driven by the CPU with valid DCR data during a *mtdcr* operation and is valid anytime the **CPU\_exeMtDcr** signal is asserted. During reset, this bus is driven with zeros by the CPU and is used by the DMA to initialize certain DCRs.

**Note:** From the DCR slave logic's perspective, this bus can be driven directly by the CPU or another's unit bypass.

## Clock Interface

Figure 13-6 demonstrates all clock interface input/output signals followed by a detailed functional description.

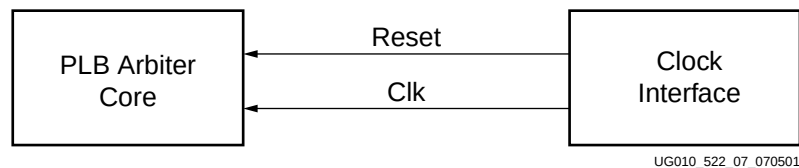


Figure 13-6: Clock Interface

### Reset (Reset)

A logic 1 placed on this input causes the PLB core to begin a system reset. This input must be maintained at a logic 1 for a total of five system clock cycles minimum for all state machines and user-programmable registers to be reset properly.

This signal is the PLB arbiter's *power on reset* signal. This signal can also be used to bring the PLB to an idle (quiescent state). The PLB idle state is defined as the bus state in which:

- No bus requests (read or write) are pending (that is, all **M\_request[n]** signals are negated)
- The bus is not locked (that is, all **M\_busLock[n]** signals, and **PLB\_busLock**, are negated)
- The bus is not granted or being granted to any master (that is, **PLB\_PAVailid** is negated)
- The read and write data buses are not being used (that is, all **Sl\_rdDAck** and **Sl\_wrDAck** signals are negated and all **Sln\_rdDbus[0:31]** and **Sln\_rdWdAddr[0:3]** signals are driven to a logic 0)

This signal must only be asserted, or negated, relative to the rising edge of **Clk**. How long this signal must be kept asserted when forcing the PLB to an idle state in a system will

depend on the actual implementation of that system's PLB arbiter, master, and slave devices.

**Note:** In addition to the reset input, a PLB master can have other means by which it can force itself into a reset state but without affecting the state of other masters and slaves attached to the PLB, or the PLB arbiter. However, if currently involved in a PLB transfer, the master must allow for the transfer to be completed, or properly terminate it by using **M\_abort[n]**. Otherwise, if a master's request is acknowledged by a slave, and the master wishes to enter its reset state before all the data associated with that request is transferred, then the master must be tolerable of receiving the data acknowledges while entering, during, and after the reset state. Furthermore, the master must negate the **M\_busLock[n]** and **M\_rdBurst[n]** signals if currently asserted.

### Clk (System Clock)

This is the PLB system clock. This signal provides the timing for the PLB and is an input to all PLB masters and slaves, as well as the PLB arbiter. All PLB master, slave, and arbiter output signals are asserted/negated relative to the rising edge of **Clk** and all PLB master, slave, and arbiter input signals are sampled relative to this edge.

**Note:** The master and slave attached to the PLB are expected to operate at the frequency of the PLB. Thus, any speed matching that is required due to I/O constraints or units that run at different frequencies will be handled in the PLB interfaces of masters and slaves. Processor cores which run at speeds significantly greater than that of the PLB will require synchronization logic to be inserted either within the processor core or between the core and the PLB.

## PLB Arbiter Registers

All PLB arbiter registers are device control registers that are accessed by using the *move from device control register (mf dcr)* and *move to device control register (mt dcr)* instructions. PLB arbiter registers are architected as 32-bits and are privileged for both read and write.

**Table 13-12** provides a summary of all PLB arbiter registers that are discussed in detail in this section. The column labeled "Default DCRN Number" lists register numbers that are used in the instruction mnemonics.

The DCR address of the PLB registers is determined by the values on the **PGM\_dcrAddr[0:5]** signals combined with the decode listed in the table below. To access the PACR for example, **PGM\_dcrAddr[0:5]** must equal **CPU\_dcrAddr[0:5]** and **CPU\_dcrAddr[6:9]** must equal 0x7. The default setting of **PGM\_dcrAddr[0:5]** is equal to 0b001001.

**Table 13-12: PLB Arbiter Registers**

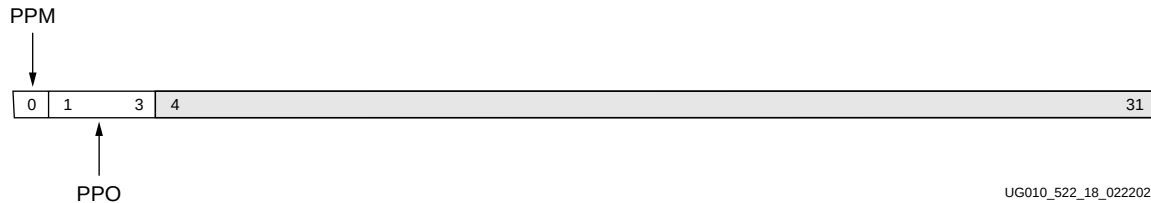
| Mnemonic | Register Name                | Decode [6:9] | Default DCRN Number | Access     |
|----------|------------------------------|--------------|---------------------|------------|
| PACR     | PLB Arbiter Control Register | 0x7          | 0x097               | Read       |
| PEAR     | PLB Error Address Register   | 0x6          | 0x096               | Read/Write |
| PESR     | PLB Error Status Register    | 0x4          | 0x094               | Read/Clear |



## PLB Arbiter Control Register (PACR)

The PLB Arbiter Control Register (PACR) is a 32-bit register that can be accessed with **CPU\_dcrAddr[6:9] = 0x7**. At reset, all bits in the PACR are loaded with zeros and Fixed Priority Order '000' is selected. Currently, this register is not writeable.

Figure 13-7 and Table 13-13 show the PACR bit definitions.



UG010\_522\_18\_022202

Figure 13-7: PLB Arbiter Control Register (PACR)

Table 13-13: PLB Arbiter Control Register (PACR)

| Bit    | Name | Description                                                                                                                                                                                                                                                                                                                                         |
|--------|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [0]    | PPM  | PLB priority mode:<br>0 → Fixed<br>1 → Fair                                                                                                                                                                                                                                                                                                         |
| [1:3]  | PPO  | PLB priority order:<br>000 → Masters 0, 1, 2, 3, 4, 5, 6, 7<br>001 → Masters 1, 2, 3, 4, 5, 6, 7, 0<br>010 → Masters 2, 3, 4, 5, 6, 7, 0, 1<br>011 → Masters 3, 4, 5, 6, 7, 0, 1, 2<br>100 → Masters 4, 5, 6, 7, 0, 1, 2, 3<br>101 → Masters 5, 6, 7, 0, 1, 2, 3, 4<br>110 → Masters 6, 7, 0, 1, 2, 3, 4, 5<br>111 → Masters 7, 0, 1, 2, 3, 4, 5, 6 |
| [4:31] |      | Reserved.                                                                                                                                                                                                                                                                                                                                           |

## PLB Error Address Register (PEAR)

The PLB Error Address Register (PEAR) is a 32-bit read-only register that contains the address of the access where the bus time-out error occurred. As in the case of the PESR master error status fields (see **PLB Error Status Register (PESR)**, page 1494), the PEAR can be locked by the master having its **M\_lockErr[n]** signal asserted for the transfer. Once locked, the PEAR cannot be updated if any subsequent error occurs until all **FLK[n]** bits in the PESR are cleared. The contents of the PEAR can be accessed by using the *move from device* control register (*mfdcr*) and the *move to device* control register (*mtdcr*) instruction.

The PEAR can be accessed with **CPU\_dcrAddr[6:9] = 0x6**. The PEAR is not affected by Reset.

## PLB Error Status Register (PESR)

The PLB Error Status Register (PESR) is a 32-bit register whose bits identify time-out errors on PLB bus transfers, the master initiating the transfer, and the type of transfer. The contents of the PESR can be loaded into a general purpose register by way of the *move from device* control register (*mfdcr*) instruction.

Each master error status field can be locked by the master having its **M\_lockErr[n]** signal asserted for the transfer in which the time-out is encountered. Once locked, the master

error status field cannot be updated if any subsequent error occurs until the **FLK[n]** bit is cleared by way of the *move to device* control register (*mtdcr*) instruction. To clear a bit in the PESR, a 1 must be written to the bit. Writing a 0 to any bit in the PESR will not affect the status of the bit.

The PESR can be accessed with **CPU\_dcrAddr[6:9] = 0x4** (read/clear) and **CPU\_dcrAddr[6:9] = 0x5** (set). At reset, all bits in the PESR are loaded with zeros.

Figure 13-8 and Table 13-14 show the PESR bit definitions.

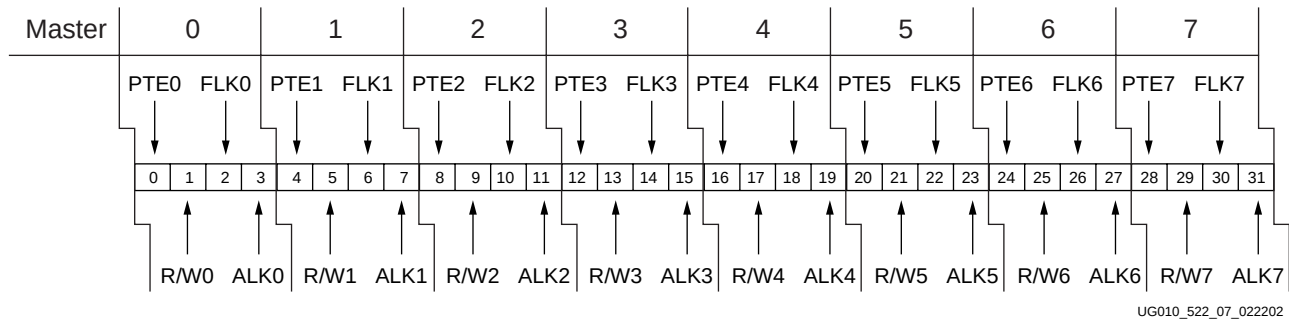


Figure 13-8: PLB Error Status Register (PESR)

Table 13-14: PLB Error Status Register (PESR)

| Bit | Name | Description                                                                                                         |
|-----|------|---------------------------------------------------------------------------------------------------------------------|
| [0] | PTE0 | Master 0 PLB time-out error status<br>0 → No master 0 time-out error.<br>1 → Master 0 time-out error.               |
| [1] | R/W0 | Master 0 read/write status<br>0 → Master 0 error operation was a write.<br>1 → Master 0 error operation was a read. |
| [2] | FLK0 | Master 0 PESR field lock<br>0 → Master 0 PESR field is unlocked.<br>1 → Master 0 field is locked.                   |
| [3] | ALK0 | Master 0 PEAR address lock<br>0 → Master 0 PEAR is unlocked.<br>1 → Master 0 PEAR is locked.                        |
| [4] | PTE1 | Master 1 PLB time-out error status<br>0 → No master 1 time-out error.<br>1 → Master 1 time-out error                |
| [5] | R/W1 | Master 1 read/write status<br>0 → Master 1 error operation was a write.<br>1 → Master 1 error operation was a read. |
| [6] | FLK1 | Master 1 PESR field lock<br>0 → Master 1 PESR field is unlocked.<br>1 → Master 1 PESR field is locked.              |
| [7] | ALK1 | Master 1 PEAR address lock<br>0 → Master 1 PEAR is unlocked.<br>1 → Master 1 PEAR is locked.                        |



Table 13-14: PLB Error Status Register (PESR) (Continued)

| Bit  | Name | Description                                                                                                         |
|------|------|---------------------------------------------------------------------------------------------------------------------|
| [8]  | PTE2 | Master 2 PLB time-out error status<br>0 → No master 2 time-out error.<br>1 → Master 2 time-out error                |
| [9]  | R/W2 | Master 2 read/write status<br>0 → Master 2 error operation was a write.<br>1 → Master 2 error operation was a read. |
| [10] | FLK2 | Master 2 PESR field lock<br>0 → Master 2 PESR field is unlocked.<br>1 → Master 2 PESR field is locked.              |
| [11] | AL2  | Master 2 PEAR address lock<br>0 → Master 2 PEAR is unlocked.<br>1 → Master 2 PEAR is locked.                        |
| [12] | PTE3 | Master 3 PLB time-out error status<br>0 → No Master 3 time-out error.<br>1 → Master 3 time-out error                |
| [13] | R/W3 | Master 3 read/write status<br>0 → Master 3 error operation was a write.<br>1 → Master 3 error operation was a read. |
| [14] | FLK3 | Master 3 PESR field lock<br>0 → Master 3 PESR field is unlocked.<br>1 → Master 3 PESR field is locked.              |
| [15] | ALK3 | Master 3 PEAR address lock<br>0 → Master 3 PEAR is unlocked.<br>1 → Master 3 PEAR is locked.                        |
| [16] | PTE4 | Master 4 PLB time-out error status<br>0 → No master 4 time-out error.<br>1 → Master 4 time-out error.               |
| [17] | R/W4 | Master 4 read/write status<br>0 → Master 4 error operation was a write.<br>1 → Master 4 error operation was a read. |
| [18] | FLK4 | Master 4 PESR field lock<br>0 → Master 4 PESR field is unlocked.<br>1 → Master 4 field is locked.                   |
| [19] | ALK4 | Master 4 PEAR address lock<br>0 → Master 4 PEAR is unlocked.<br>1 → Master 4 PEAR is locked.                        |
| [20] | PTE5 | Master 5 PLB time-out error status<br>0 → No master 1 time-out error.<br>1 → Master 1 time-out error                |

Table 13-14: PLB Error Status Register (PESR) (Continued)

| Bit  | Name | Description                                                                                                         |
|------|------|---------------------------------------------------------------------------------------------------------------------|
| [21] | R/W5 | Master 5 read/write status<br>0 → Master 5 error operation was a write.<br>1 → Master 5 error operation was a read. |
| [22] | FLK5 | Master 5 PESR field lock<br>0 → Master 5 PESR field is unlocked.<br>1 → Master 5 PESR field is locked.              |
| [23] | ALK5 | Master 5 PEAR address lock<br>0 → Master 5 PEAR is unlocked.<br>1 → Master 5 PEAR is locked.                        |
| [24] | PTE6 | Master 6 PLB time-out error status<br>0 → No master 6 time-out error.<br>1 → Master 6 time-out error                |
| [25] | R/W6 | Master 6 read/write status<br>0 → Master 6 error operation was a write.<br>1 → Master 6 error operation was a read. |
| [26] | FLK6 | Master 6 PESR field lock<br>0 → Master 6 PESR field is unlocked.<br>1 → Master 6 PESR field is locked.              |
| [27] | AL6  | Master 6 PEAR address lock<br>0 → Master 6 PEAR is unlocked.<br>1 → Master 6 PEAR is locked.                        |
| [28] | PTE7 | Master 7 PLB time-out error status<br>0 → No Master 7 time-out error.<br>1 → Master 7 time-out error                |
| [29] | R/W7 | Master 7 read/write status<br>0 → Master 7 error operation was a write.<br>1 → Master 7 error operation was a read. |
| [30] | FLK7 | Master 7 PESR field lock<br>0 → Master 7 PESR field is unlocked.<br>1 → Master 7 PESR field is locked.              |
| [31] | ALK7 | Master 7 PEAR address lock<br>0 → Master 7 PEAR is unlocked.<br>1 → Master 7 PEAR is locked.                        |

## PLB Arbiter Operations

The *Processor Local Bus Architecture Specifications* document provides a comprehensive discussion on the various PLB arbiter operations and transfers that are similar to the PLB. This section on PLB arbiter operations provides detailed information on operations that are specific to the PLB arbiter, namely:

- **Single Read Transfer Bus Time-Out**
- **Single Write Transfer Bus Time-Out**
- **Line Read Transfer Bus Time-Out**
- **Line Write Transfer Bus Time-Out**
- **Burst Read Transfer Bus Time-Out**
- **Burst Write Transfer Bus Time-Out**
- **Read Transfer**
- **Pipelined Read Transfer**
- **Pipelined Write Transfer**

## Single Read Transfer Bus Time-Out

Figure 13-9 shows a bus time-out for a single read transfer on the PLB. The PLB arbiter samples the **SI\_wait** and **SI\_rearbitrate** signals 15 cycles after the initial assertion of the **PLB\_PValid** signal, and if both are negated, it asserts the **PLB\_MAddrAck[n]** signal. During the cycle in which **PLB\_MAddrAck[n]** is asserted, the PLB watchdog timer samples the **SI\_addrAck** signal, and if negated, it completes the handshaking to the master by asserting the **PLB\_MRdDack[n]** and **PLB\_MErr[n]** signals two cycles later.

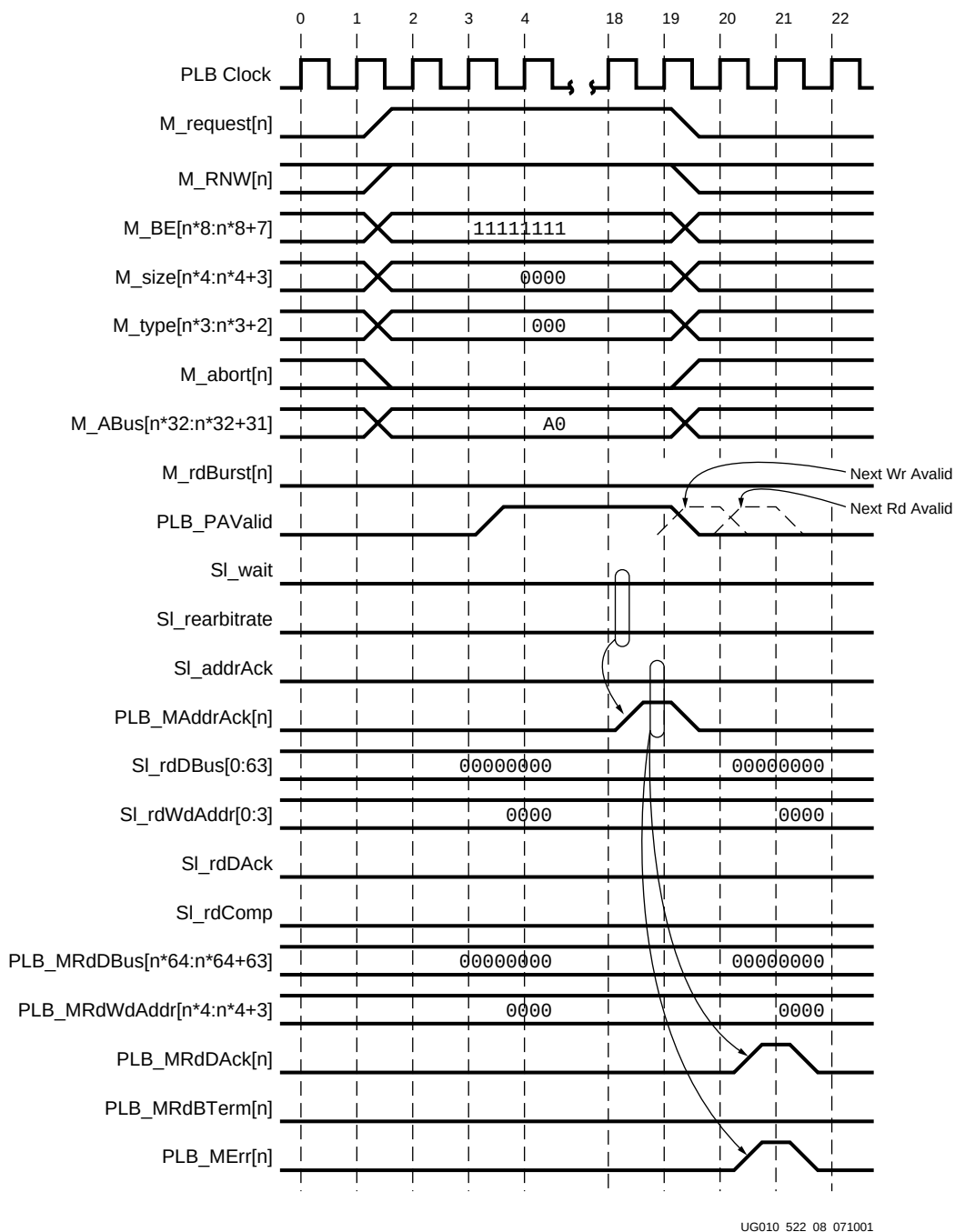
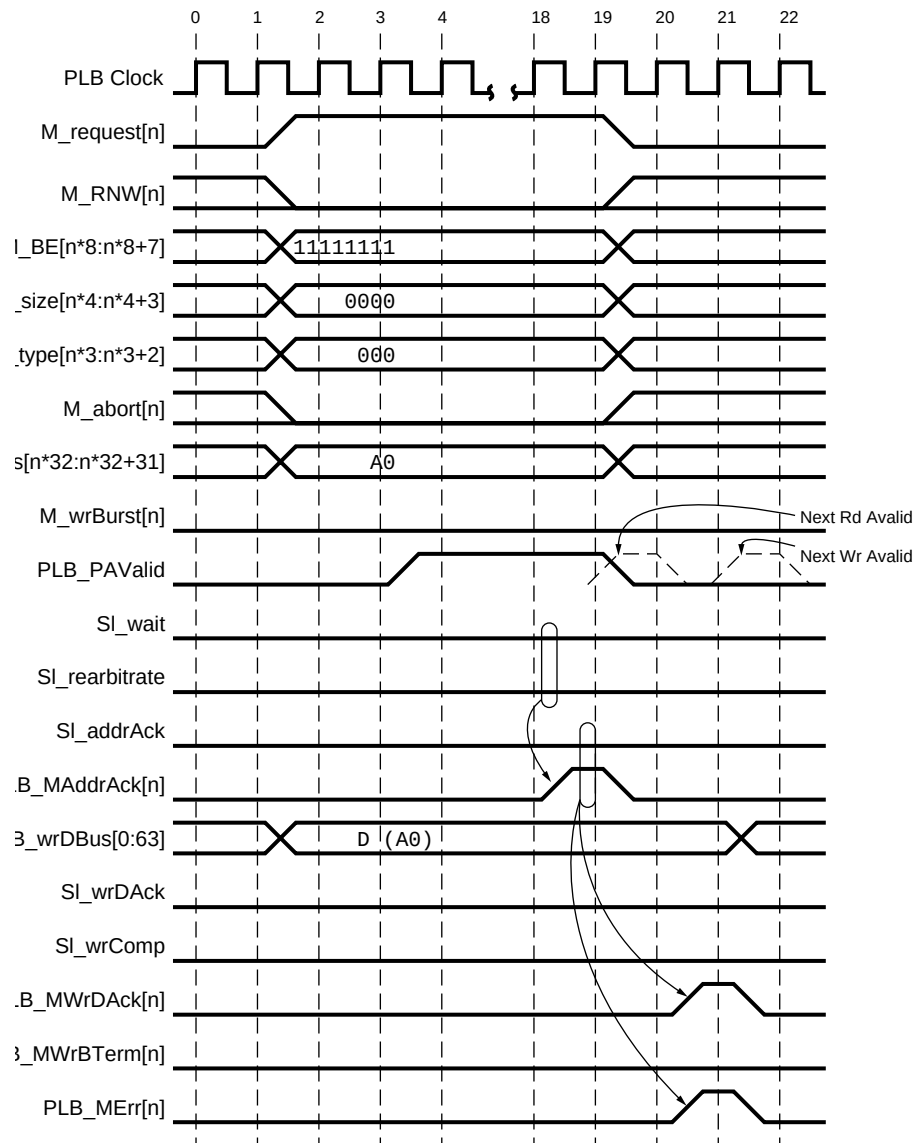


Figure 13-9: Single Read Transfer Bus Time-Out

## Single Write Transfer Bus Time-Out

Figure 13-10 shows a bus time-out for a single write transfer on the PLB. The PLB arbiter samples the **SI\_wait** and **SI\_rearbitrate** signals 15 cycles after the initial assertion of the **PLB\_PValid** signal, and if both are negated, it asserts the **PLB\_MAddrAck[n]** signal. During the cycle in which **PLB\_MAddrAck[n]** is asserted, the PLB watchdog timer samples the **SI\_addrAck** signal, and if negated, it completes the handshaking to the master by asserting the **PLB\_MWrDack[n]** and **PLB\_MErr[n]** signals two cycles later.

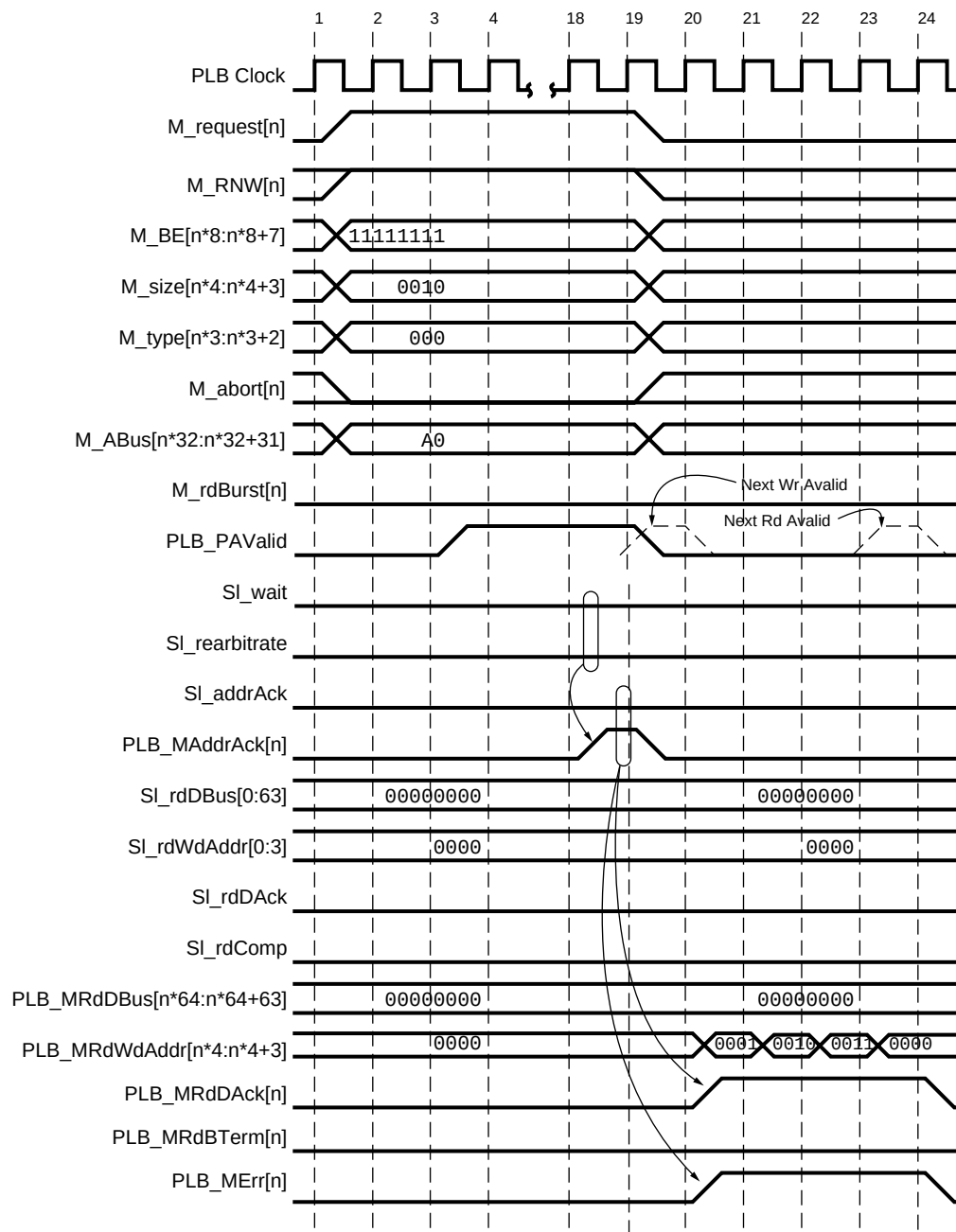


UG010\_522\_09\_071101

Figure 13-10: Single Write Transfer Bus Time-Out

## Line Read Transfer Bus Time-Out

Figure 13-11 shows a bus time-out for a 4-word line read transfer on the PLB. The PLB arbiter samples the **SI\_wait** and **SI\_rearbitrate** signals 15 cycles after the initial assertion of the **PLB\_PValid** signal, and if both are negated, it asserts the **PLB\_MAddrAck[n]** signal. During the cycle in which **PLB\_MAddrAck[n]** is asserted, the PLB watchdog timer samples the **SI\_addrAck** signal, and if negated, it completes the handshaking to the master by asserting the **PLB\_MRdDack[n]** and **PLB\_MErr[n]** signals two cycles later.

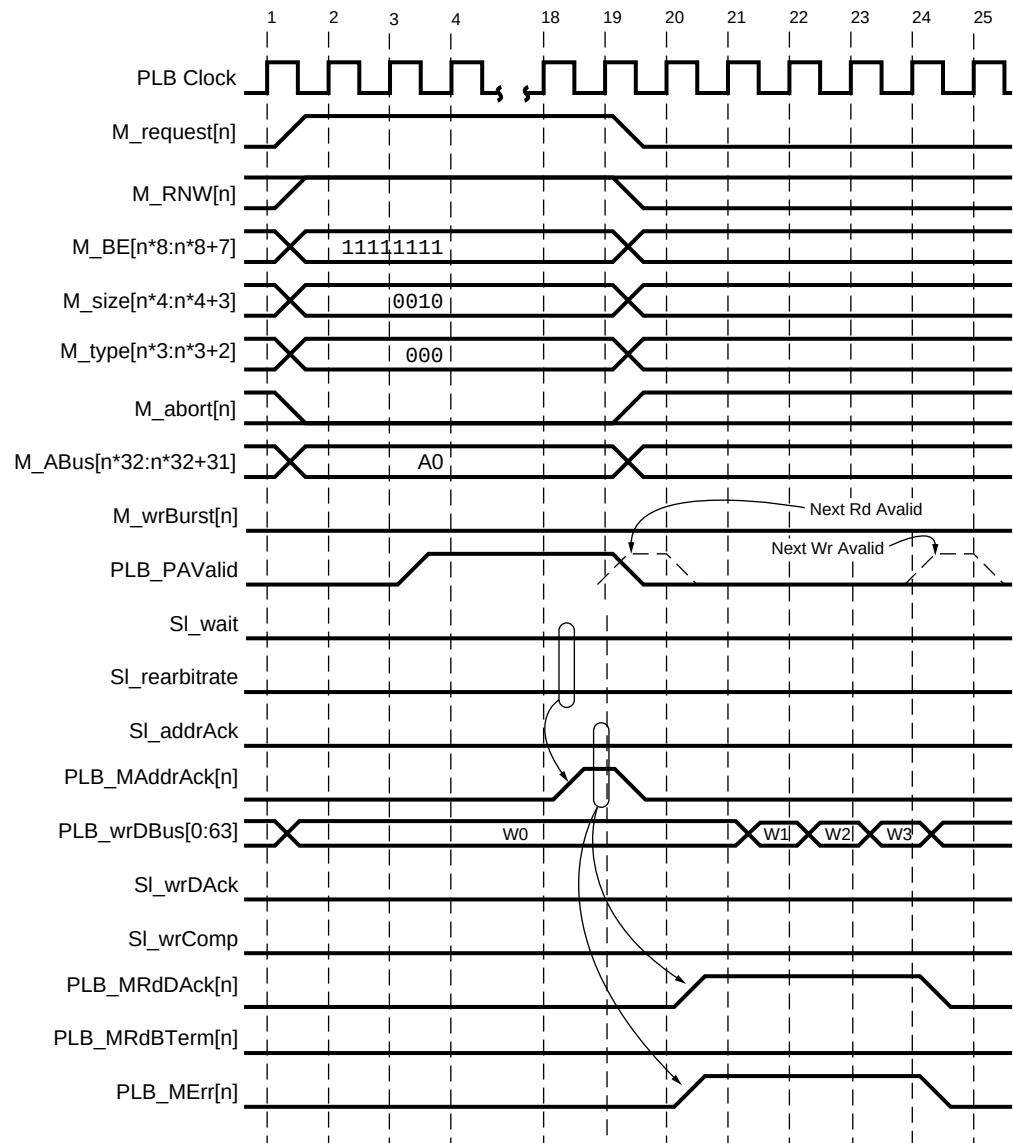


UG010\_522\_10\_071101

Figure 13-11: Line Read Transfer Bus Time-Out

## Line Write Transfer Bus Time-Out

Figure 13-12 shows a bus time-out for a line write transfer on the PLB. The PLB arbiter samples the **SI\_wait** and **SI\_rearbitrate** signals 15 cycles after the initial assertion of the **PLB\_PValid** signal, and if both are negated, it asserts the **PLB\_MAddrAck[n]** signal. During the cycle in which **PLB\_MAddrAck[n]** is asserted, the PLB watchdog timer samples the **SI\_addrAck** signal, and if negated, it completes the handshaking to the master by asserting the **PLB\_MWrDack[n]** and **PLB\_MErr[n]** signals two cycles later.



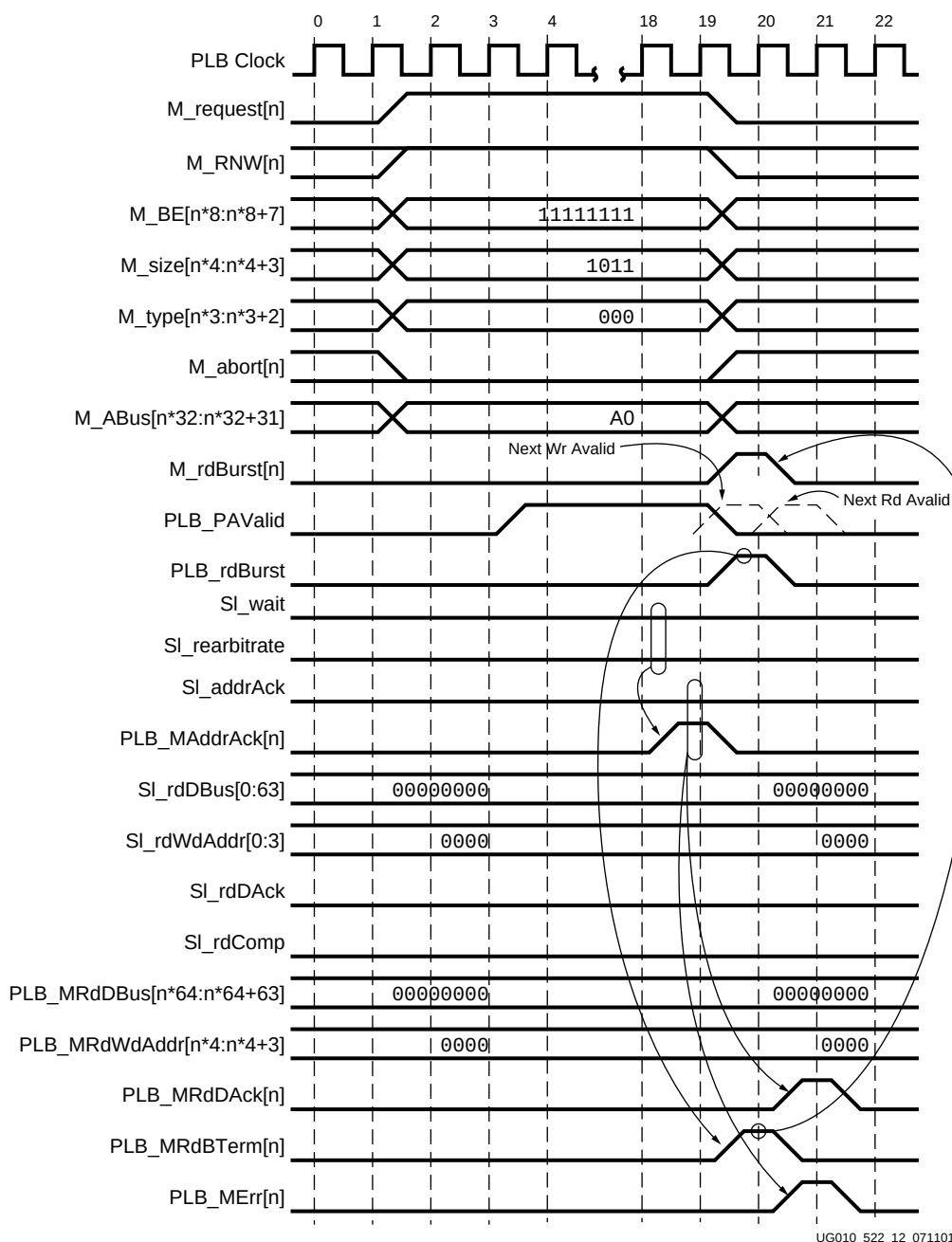
UG010\_522\_11\_071101

Figure 13-12: Line Write Transfer Bus Time-Out

## Burst Read Transfer Bus Time-Out

**Figure 13-13** shows a bus time-out for a burst read transfer on the PLB. The PLB arbiter samples the **SI\_wait** and **SI\_rearbitrate** signals 15 cycles after the initial assertion of the **PLB\_PValid** signal, and if both are negated, it asserts the **PLB\_MAddrAck[n]** signal. During the cycle in which **PLB\_MAddrAck[n]** is asserted, the PLB watchdog timer samples the **SI\_addrAck** signal, and if negated, it completes the handshaking to the master by asserting the **PLB\_MRdDack[n]** and **PLB\_MErr[n]** signals two cycles later. Also, in the cycle following the assertion of **PLB\_MAddrAck[n]**, the PLB watchdog timer samples the **PLB\_rdBurst** signal, and if asserted, it asserts the **PLB\_MRdBTerm[n]** signal in the same cycle.

**Note:** The master is expected to negate its **M\_rdBurst[n]** signal in response to the assertion of **PLB\_MRdBTerm[n]**.



UG010\_522\_12\_071101

**Figure 13-13: Burst Read Transfer Bus Time-Out**



## Burst Write Transfer Bus Time-Out

Figure 13-14 shows a bus time-out for a burst write transfer on the PLB. The PLB arbiter samples the **SI\_wait** and **SI\_rearbitrate** signals 15 cycles after the initial assertion of the **PLB\_PValid** signal, and if both are negated, it asserts the **PLB\_MAddrAck[n]** signal. During the cycle in which **PLB\_MAddrAck[n]** is asserted, the PLB watchdog timer samples the **SI\_addrAck** signal, and if negated, it completes the handshaking to the master by asserting the **PLB\_MWrDack[n]** and **PLB\_MErr[n]** signals two cycles later. Also, in the cycle following the assertion of **PLB\_MAddrAck[n]**, the PLB watchdog timer samples the **PLB\_wrBurst** signal, and if asserted, it asserts the **PLB\_MWrBTerm[n]** signal in the same cycle.

**Note:** The master is expected to negate its **M\_wrBurst[n]** signal in response to the assertion of **PLB\_MWrBTerm[n]**.

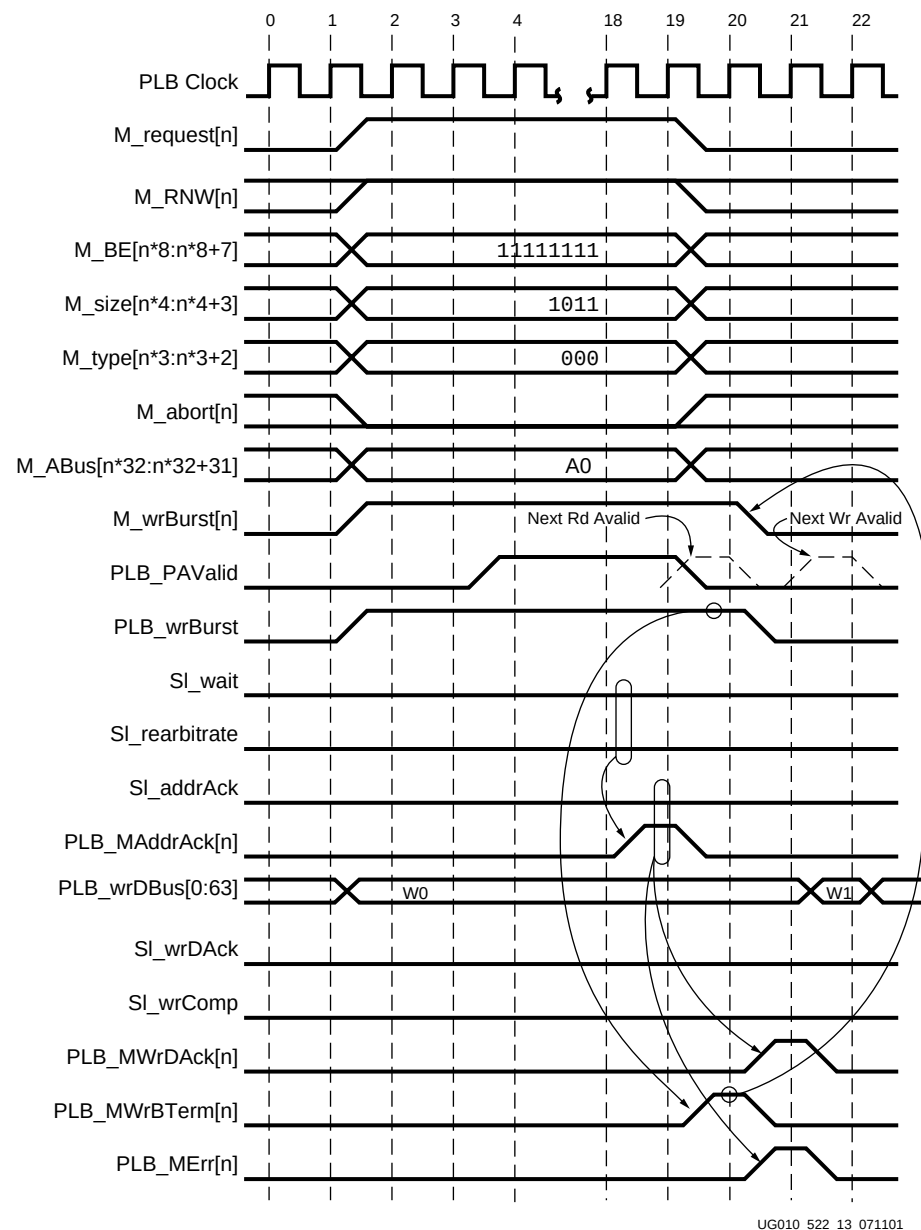
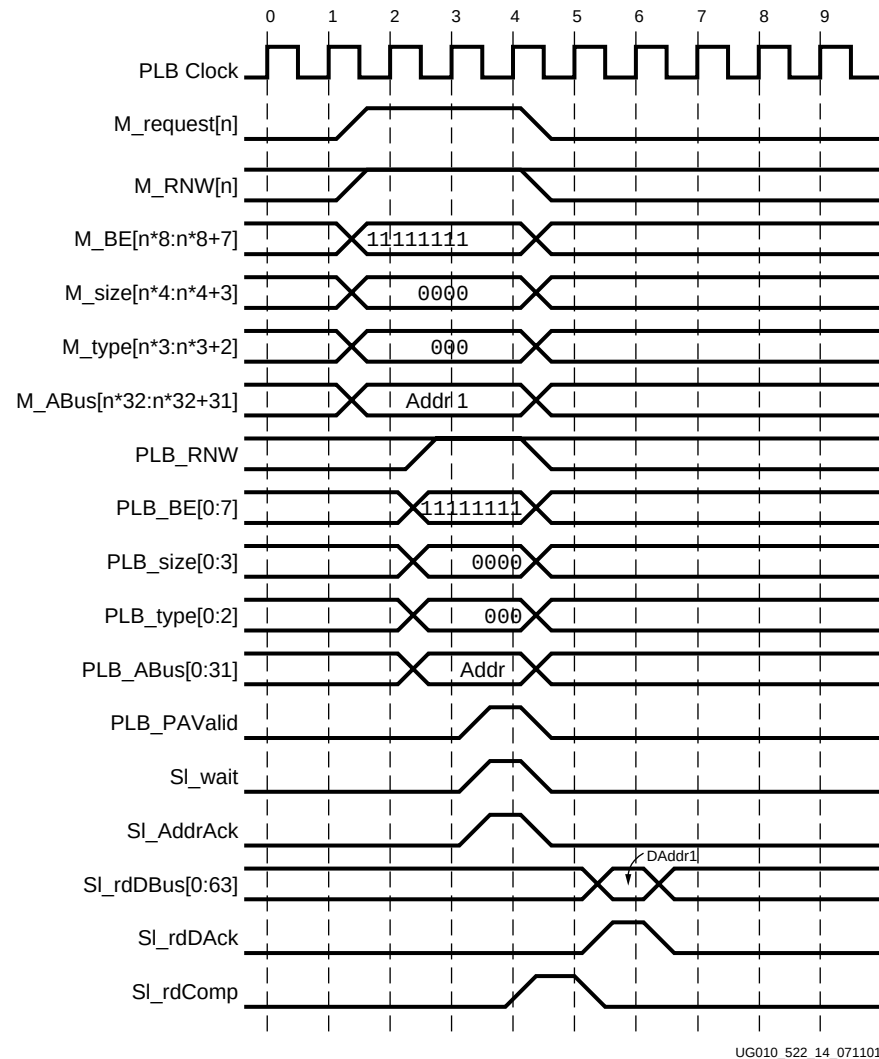


Figure 13-14: Burst Write Transfer Bus Time-Out

## Read Transfer

Figure 13-15 shows the operation of a single read data transfer on the PLB bus with one cycle delay before **PLB\_PValid** is driven out. Note that the **Master [n]** transfer qualifiers are driven to the PLB arbiter in cycle 1, and then they are driven out onto the PLB bus in cycle number 2 after this master wins the grant from the PLB arbiter. Thus these qualifiers are available at the beginning of the address valid cycle (cycle #3) for the slave to use.

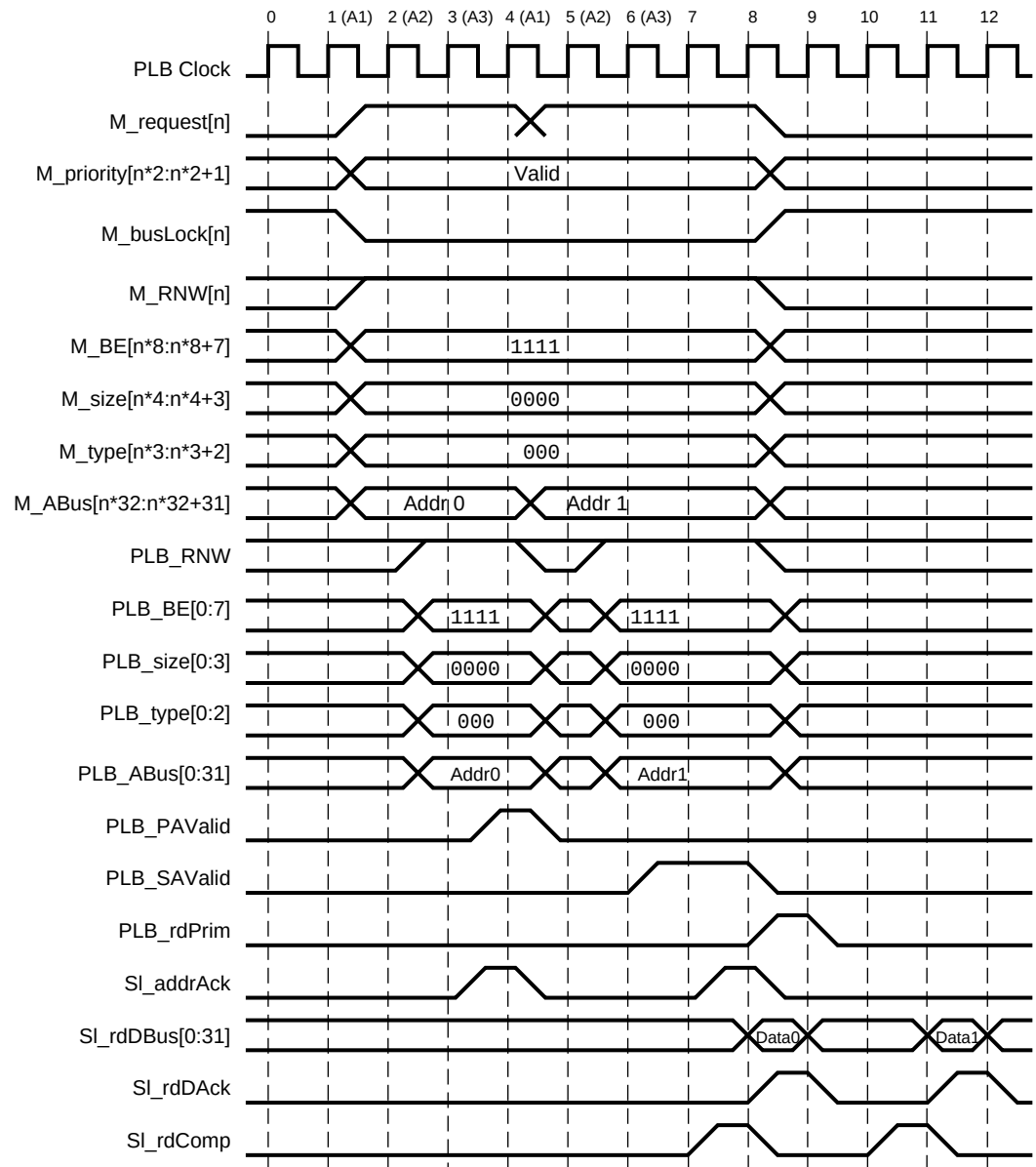


UG010\_522\_14\_071101

Figure 13-15: Read Transfer

## Pipelined Read Transfer

In **Figure 13-16** a master is requesting multiple reads. The first read is a primary read that is acknowledged by the slave in cycle 3. The master has another request pending, keeping its request on, which becomes a secondary read request on the PLB Bus in clock 6. This is acknowledged by the slave device in the next clock cycle as **SI\_rdComp** is given. Note that **PLB\_rdPrim** is given with **SI\_rdComp**.



UG010\_522\_15\_071201

Figure 13-16: Back-to-Back Reads

## Pipelined Write Transfer

In **Figure 13-17**, a master is requesting multiple writes. The first write is a primary write and is the next request becomes a secondary write request which is acknowledged by the slave device. **PLB\_wrPrim** is driven in the cycle that the first write completes (slave asserts **SI\_wrComp**).

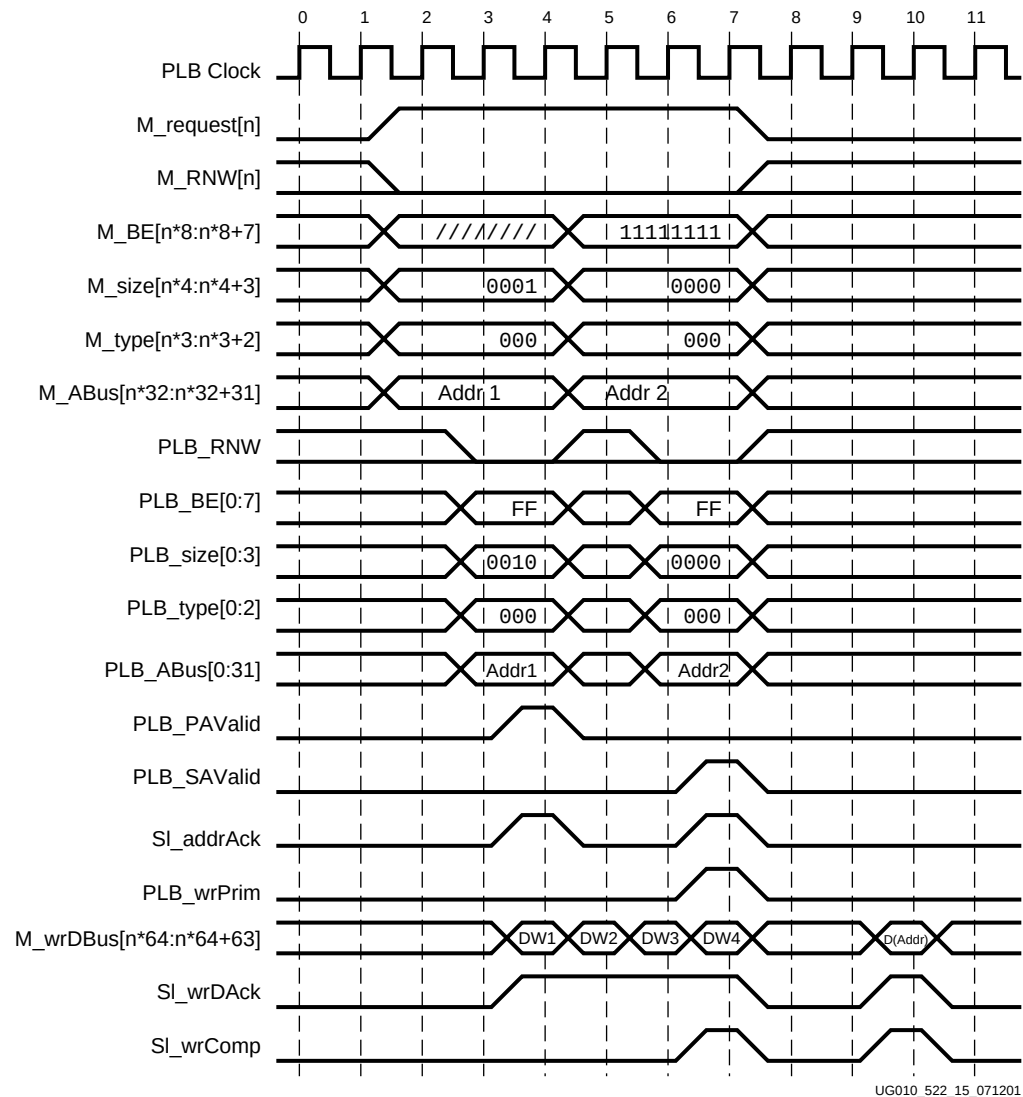


Figure 13-17: Pipelined Write Transfer



# PLB BRAM Controller

---

## Overview

The 64-bit block RAM (BRAM) controller module is a PLB slave device that acts as a high-speed memory controller. It allows a wide degree of flexibility for interfacing with BRAMs. The slave interface supports a subset of the PLB specification that minimizes logic complexity while providing high performance and full compatibility with the CPU.

## Related Documents

The following documents provide additional information

- *IBM CoreConnect™ PLB Specification*
- *Virtex-II Pro™ Platform FPGAs (Advance Product Specification)*
- *Using Data2BRAM*

## Features

- 64-bit PLB interface
- Implements a 64-bit BRAM data bus with byte enables for writes
  - Directly connects with 16 KB to 128 KB of BRAM (other sizes are possible with some external logic)
- Supports the following PLB transfer sizes:
  - 1-8 byte(s) non-burst
  - 4-word cacheline (target word first during reads)
  - 8-word cacheline (target word first during reads)
  - 16-word cacheline (target word first during reads)
- Supports up to 16 PLB masters
- Contains no control or status registers

## Module Port Interface

### PLB Slave Interface Signals

Table 14-1: Global Signals

| Name         | Direction | Description      |
|--------------|-----------|------------------|
| SYS_plbClk   | Input     | PLB system clock |
| SYS_plbReset | Input     | PLB system reset |

Table 14-2: Transfer Qualifier Signals

| Name              | Direction | Description                            |
|-------------------|-----------|----------------------------------------|
| PLB_PAVValid      | Input     | PLB primary address valid indicator    |
| PLB_busLock       | Input     | PLB bus lock                           |
| PLB_masterID[0:3] | Input     | PLB current master identifier          |
| PLB_RNW           | Input     | PLB read not write                     |
| PLB_BE[0:7]       | Input     | PLB byte enables                       |
| PLB_size[0:3]     | Input     | PLB transfer size                      |
| PLB_type[0:2]     | Input     | PLB transfer type                      |
| PLB_MSize[0:1]    | Input     | PLB master data bus size               |
| PLB_compress      | Input     | PLB compressed data transfer indicator |
| PLB_guarded       | Input     | PLB guarded transfer indicator         |
| PLB_ordered       | Input     | PLB synchronize transfer indicator     |
| PLB_lockErr       | Input     | PLB lock error indicator               |
| PLB_abort         | Input     | PLB abort bus request indicator        |
| PLB_ABus[0:31]    | Input     | PLB address bus                        |
| Sl_addrAck        | Output    | Slave address acknowledge              |
| Sl_wait           | Output    | Slave wait indicator                   |
| Sl_SSize[0:1]     | Output    | Slave data bus size                    |
| Sl_rearbitrate    | Output    | Slave rearbitrate bus indicator        |
| Sl_MBusy[0:15]    | Output    | Slave busy indicator                   |
| Sl_MErr[0:15]     | Output    | Slave error indicator                  |

Table 14-3: Address Pipelining Signals

| Name         | Direction | Description                                      |
|--------------|-----------|--------------------------------------------------|
| PLB_SAVValid | Input     | PLB secondary address valid indicator            |
| PLB_rdPrim   | Input     | PLB secondary to primary read request indicator  |
| PLB_wrPrim   | Input     | PLB secondary to primary write request indicator |

Table 14-4: Write Data Bus Signals

| Name             | Direction | Description                             |
|------------------|-----------|-----------------------------------------|
| PLB_wrDBus[0:63] | Input     | PLB write data bus                      |
| PLB_wrBurst      | Input     | PLB burst write transfer indicator      |
| Sl_wrDAck        | Output    | Slave read data acknowledge             |
| Sl_wrComp        | Output    | Slave write transfer complete indicator |
| Sl_wrBTerm       | Output    | Slave terminate write burst transfer    |

Table 14-5: Read Data Bus Signals

| Name             | Direction | Description                            |
|------------------|-----------|----------------------------------------|
| PLB_rdBurst      | Input     | PLB burst read transfer indicator      |
| Sl_rdBUS[0:63]   | Output    | Slave read data bus                    |
| Sl_rdWdAddr[0:3] | Output    | Slave read word address                |
| Sl_rDAck         | Output    | Slave read data acknowledge            |
| Sl_rdComp        | Output    | Slave read transfer complete indicator |
| Sl_rdBTerm       | Output    | Slave terminate read burst transfer    |

## BRAM Interface Signals

Table 14-6: BRAM Interface Signals

| Name            | Direction | Description       |
|-----------------|-----------|-------------------|
| BRAM_dout[0:63] | Input     | BRAM Data Out     |
| BRAM_addr[0:28] | Output    | BRAM Address      |
| BRAM_din[0:63]  | Output    | BRAM Data In      |
| BRAM_en         | Output    | BRAM Enable       |
| BRAM_wen[0:7]   | Output    | BRAM Write Enable |



## Parameters

Table 14-7: Parameters

| Name        | Direction | Description                                                                                                                                                        |
|-------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| C_BASE_ADDR | N/A       | 32-bit base address of the PLB BRAM Controller                                                                                                                     |
| C_LSB_ADDR  | N/A       | Least Significant Address Bit Used By Address Comparator (i.e. setting C_LSB_ADDR=15 means PLB Address bits [0:15] are decoded which implies a 64 KB memory space) |

## High-Level Description

The PLB BRAM controller is designed using a pipelined architecture to enable it to operate at high clock frequencies. The basic design of the BRAM Controller (see [Figure 14-1](#)) contains logic to decode transfer requests and address counters to handle burst operations. Additional logic controls the timing of signals such as write strobes, data acknowledges, and various other PLB handshaking signals. The address counters are capable of being sequenced in target-word-first order to reduce CPU stalls during cacheline read requests. For maximum system performance and reduced area, the BRAM controller's logic is designed to support a subset of PLB transactions used by the CPU.

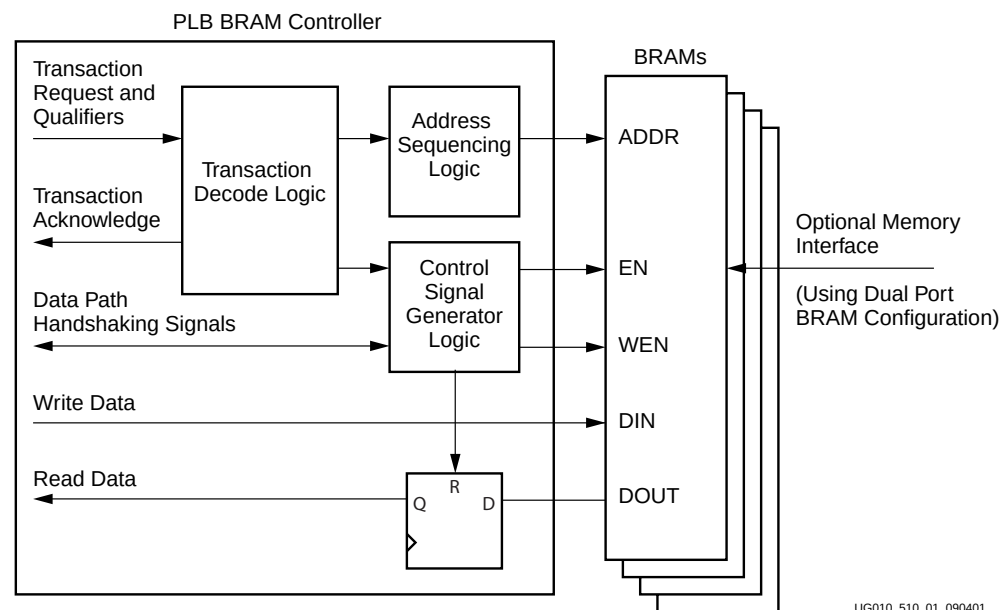


Figure 14-1: High-Level View of PLB BRAM Controller

## Implementation

The Transaction Decode Logic uses the PLB address and transfer qualifiers to look for valid transaction requests. A valid transaction requires that the request address fit within the address space of the BRAM controller and that the transfer be a non-burst or cacheline sized operation from a 64-bit master. The transfer type must also be encoded as a memory type operation (as opposed to DMA transfer types). Transactions that do not fit these

qualifications are ignored, while valid ones cause a PLB address acknowledge to be returned. The address decode logic requires a full clock cycle to operate, which adds one cycle of latency to transactions. This address decode cycle can be overlapped, and therefore hidden, if the transaction is pipelined behind another ongoing operation. Once a valid transaction is accepted, a command is issued to the Address Sequencing Logic and the Control Signal Generator Logic to execute the transaction.

The Address Sequencing Logic gets loaded with the start address and then cycles through subsequent addresses during cacheline transfers. The addresses for read transactions are sequenced in target-word-first order. These address signals directly feed the BRAM's address inputs.

The Control Signal Generator Logic handles all the remaining PLB and BRAM signals to ensure that their timing and behavior are correct. The control logic essentially runs off a down counter that is loaded with the transaction burst length and stops when it hits zero. As long as the counter is non-zero, a data transfer is in progress, meaning that read/write acknowledges and strobes can be generated. A non-zero counter value also asserts a busy flag to prevent another transaction from being initiated by the Transaction Decode Logic.

## Design Extensions and Improvements

The BRAM controller design provides a reference example of a high-performance PLB slave memory controller. The design can be modified to better suit the system it is used in, or to interface with other synchronous SRAM-like devices. The notes below provide some ideas for how one might further customize the design.

### User Port Configuration

Virtex-II Pro BRAM blocks are true dual port memories. The second BRAM port can therefore be configured to operate at a different clock rate or even a different bus width than the memory port used by the PLB interface. One can use creativity in utilizing the second BRAM port to facilitate bus matching functionality or even allow multiple devices to access different sections of the memory independently. The true dual port design of the BRAMs provides a high degree of flexibility to design with.

### Number of BRAMs

The design can directly support 8 BRAM modules (8-bit wide configuration) up to 64 BRAM modules (1-bit wide configuration) since it utilizes a 64-bit data path with 8 byte enables. The number of BRAMs can be changed by redistributing the BRAM control signals. If byte enables need not be supported, the number of BRAMs can be decreased further to reduce BRAM resource utilization or to improve timing. Additional BRAMs can be added to build bigger memories by including the necessary multiplexing logic.

### Code / Data Space Initialization

BRAMs can be pre-initialized with data or instructions that would be available to the CPU at powerup or after partial reconfiguration. A tool called "data2bram" helps designers generate the necessary BRAM initialization information to be used during simulation, place and route, or bitstream generation.



## PLB UART Controller

---

### Overview

The PLB Universal Asynchronous Receiver/Transmitter (UART) Controller module is a PLB slave device that implements a standard serial interface. The UART module is an example design that utilizes a PLB-based Intellectual Property Interface (IPIF) to help simplify the development of PLB devices. The PLB IPIF module provides a generic SRAM protocol interface to connect a synthesizable 16450 or 16550 UART core to the PLB infrastructure.

### Related Documents

The following documents provide additional useful information:

- *IBM CoreConnect™ PLB Specification*
- *Virtex-II Pro™ Platform FPGAs (Advance Product Specification )*
- *IPIF Specification*
- *OPB 16450 UART Controller*
- *OPB 16550 UART Controller*

### Features

- 64-bit PLB interface
- IPIF module with SRAM protocol interface
  - Serves as a design example for IPIF modules
- 16450/16550 UART functionality

### Module Port Interface

#### PLB Slave Interface Signals

Table 15-1: Global Signals

| Name         | Direction | Description      |
|--------------|-----------|------------------|
| SYS_plbClk   | Input     | PLB system clock |
| SYS_plbReset | Input     | PLB system reset |

Table 15-2: Transfer Qualifier Signals

| Name              | Direction | Description                            |
|-------------------|-----------|----------------------------------------|
| PLB_PAVValid      | Input     | PLB primary address valid indicator    |
| PLB_busLock       | Input     | PLB bus lock                           |
| PLB_masterID[0:3] | Input     | PLB current master identifier          |
| PLB_RNW           | Input     | PLB read not write                     |
| PLB_BE[0:7]       | Input     | PLB byte enables                       |
| PLB_size[0:3]     | Input     | PLB transfer size                      |
| PLB_type[0:2]     | Input     | PLB transfer type                      |
| PLB_MSize[0:1]    | Input     | PLB master data bus size               |
| PLB_compress      | Input     | PLB compressed data transfer indicator |
| PLB_guarded       | Input     | PLB guarded transfer indicator         |
| PLB_ordered       | Input     | PLB synchronize transfer indicator     |
| PLB_lockErr       | Input     | PLB lock error indicator               |
| PLB_abort         | Input     | PLB abort bus request indicator        |
| PLB_ABus[0:31]    | Input     | PLB address bus                        |
| Sl_addrAck        | Output    | Slave address acknowledge              |
| Sl_wait           | Output    | Slave wait indicator                   |
| Sl_SSize[0:1]     | Output    | Slave data bus size                    |
| Sl_rearbitrate    | Output    | Slave rearbitrate bus indicator        |
| Sl_MBusy[0:15]    | Output    | Slave busy indicator                   |
| Sl_MErr[0:15]     | Output    | Slave error indicator                  |

Table 15-3: Address Pipelining Signals

| Name         | Direction | Description                                      |
|--------------|-----------|--------------------------------------------------|
| PLB_SAVValid | Input     | PLB secondary address valid indicator            |
| PLB_rdPrim   | Input     | PLB secondary to primary read request indicator  |
| PLB_wrPrim   | Input     | PLB secondary to primary write request indicator |

Table 15-4: Write Data Bus Signals

| Name             | Direction | Description                             |
|------------------|-----------|-----------------------------------------|
| PLB_wrDBus[0:63] | Input     | PLB write data bus                      |
| PLB_wrBurst      | Input     | PLB burst write transfer indicator      |
| Sl_wrDAck        | Output    | Slave read data acknowledge             |
| Sl_wrComp        | Output    | Slave write transfer complete indicator |
| Sl_wrBTerm       | Output    | Slave terminate write burst transfer    |

Table 15-5: Read Data Bus Signals

| Name             | Direction | Description                            |
|------------------|-----------|----------------------------------------|
| PLB_rdBurst      | Input     | PLB burst read transfer indicator      |
| Sl_rDBus[0:63]   | Output    | Slave read data bus                    |
| Sl_rdWdAddr[0:3] | Output    | Slave read word address                |
| Sl_rDAck         | Output    | Slave read data acknowledge            |
| Sl_rdComp        | Output    | Slave read transfer complete indicator |
| Sl_rdBTerm       | Output    | Slave terminate read burst transfer    |

## UART Interface Signals

Table 15-6: UART Interface Pin Description

| Name      | Direction | Description         |
|-----------|-----------|---------------------|
| uart_ctsN | Input     | Clear to send       |
| uart_dsrN | Input     | Data set ready      |
| uart_dcdN | Input     | Data carrier detect |
| uart_rIN  | Input     | Ring indicator      |
| uart_sin  | Input     | Serial input        |
| uart_dtrN | Output    | Data terminal ready |
| uart_intr | Output    | Interrupt           |
| uart_rtsN | Output    | Request to send     |
| uart_sout | Output    | Serial output       |

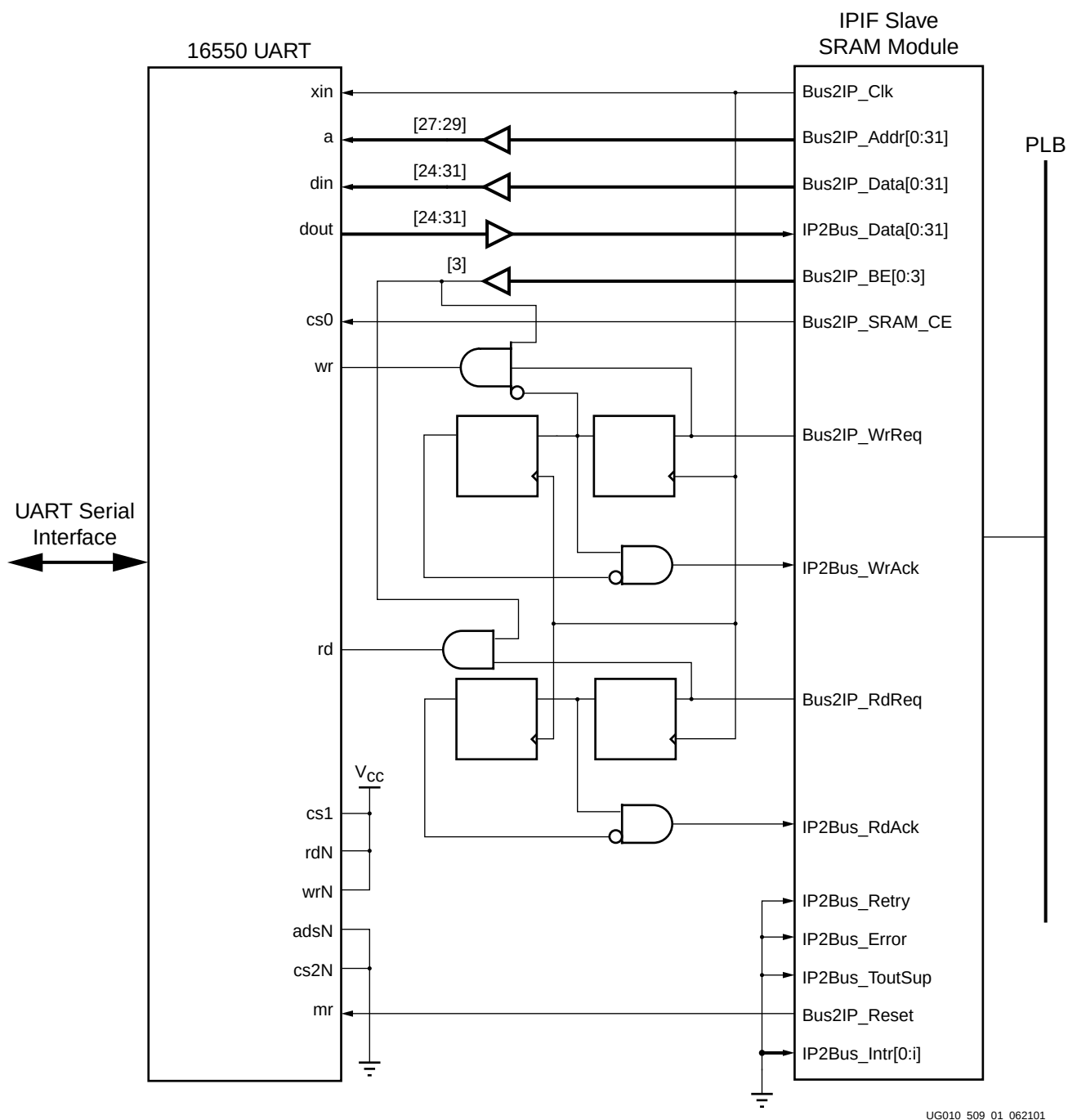
Table 15-7: Parameters

| Name        | Direction | Description                                                                |
|-------------|-----------|----------------------------------------------------------------------------|
| C_BASE_ADDR | N/A       | 32-bit base address of PLB UART<br>(must be aligned to a 32-byte boundary) |

## Implementation

The PLB UART is designed by connecting a synthesizable 16450/16550 UART core to a PLB IPIF slave module, as shown in [Figure 15-1](#). The PLB IPIF slave makes it extremely easy to attach the 16450/16550 UART core to the PLB CoreConnect Bus and uses very little glue logic. The UART's control and status register functionality is taken from an industry standard 16450/16550 UART such as the PC16550D from National Semiconductor (See <http://www.national.com/pf/PC/PC16550D.html>.) Note that some differences exist

between the PLB UART and a standard 16450/16550 UART due to FPGA implementation issues.



UG010\_509\_01\_062101

Figure 15-1: Connection Between PLB IPIF and Synthesizable UART Core

## PLB IPIF

The PLB IPIF provides a slave SRAM protocol as described in the IPIF Specification. The version of the PLB IPIF used by the PLB UART only supports PLB non-burst memory transactions (PLB\_size[0:3] = 0000, PLB\_type[0:2] = 000) of 1-4 bytes. It ignores all other PLB transaction sizes and types.



## Differences From 16550 UART Discrete Chips

The UART design differs from chips such as the National Semiconductor PC16550D in order to make the design better suited for FPGA implementation. In some cases, the functionality is modified or some pins may have been internalized/removed. Most of these changes have little effect on the functionality of the UART from a software perspective.

- The baudoutN (Baud Out) and rclk (Receiver Clock) pins have been internalized. These pins are tied together internally and the frequency of baudoutN is determined by setting the divisor latch register inside the UART. In order to save FPGA clock resources, the user cannot externally supply the receiver clock to the UART.
- The output signals DDIS (Driver Disable), out1N (Output 1), out2N (Output 2), rxrdyN (Receiver DMA Signalling), and txrdyN (Transmitter DMA Signalling) are available internally, but are not brought out to the PLB UART's ports.
- Refer to associated Xilinx 16450/16550 UART Controller documentation for additional information about differences from discrete UART chips.

## PLB SRAM/Flash Controller

### Overview

This document describes the SRAM/Flash Controller module that interfaces to the IBM CoreConnect™ Processor Local Bus (PLB). This document assumes that the user is familiar with PowerPC® and PLB-based designs as well as the basics of SRAM and Flash. All examples and diagrams in this document were made with the 4 Mb IDT SRAM and 64 Mb Intel® StrataFlash® memory devices as a reference. **Figure 16-1** is a block diagram of the interface of the controller between the PowerPC PLB and the external memories. The SRAM/Flash Controller module connects to the PLB as a slave device. A typical configuration is shown in **Figure 16-1**.

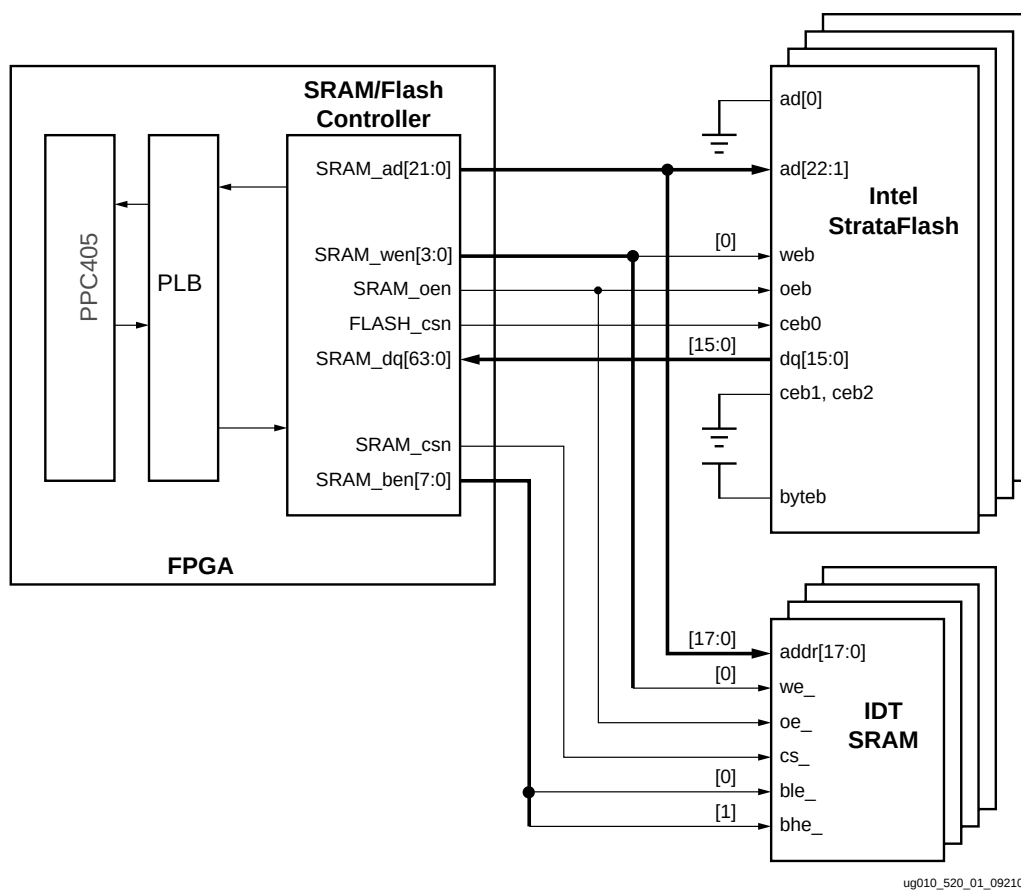


Figure 16-1: SRAM/Flash Controller Module Block Diagram

## Related Documents

- *PLB Architecture Specification*
- IDT, *SRAM Data Sheet*,  
[http://www.idt.com/docs/71V416\\_DS\\_74666.pdf](http://www.idt.com/docs/71V416_DS_74666.pdf)
- Intel, *StrataFlash Data Sheet*,  
<ftp://download.intel.com/design/flcomp/datashts/29066709.pdf>

## Features

- Tuned for the PLB interface
- 64-bit data read and write
- Supports up to 128 MB SDRAMs and 128 MB Flash
- Supports single data beat or cacheline data transfers (bursts of 2, 4, 8, or 16 words)
- Supports *target word first* read burst for optimized CPU operation

The current implementation is focused on simulation. Future versions can be tuned for synthesis within the Virtex-II fabric to operate at 133 MHz.

## Module Port Interface

Table 16-1 lists the pins in the SRAM interface.

Table 16-1: SRAM Controller Signals

| Name              | Direction | Description                                                 |
|-------------------|-----------|-------------------------------------------------------------|
| SYS_plbClk        | Input     | PLB system clock                                            |
| SYS_plbReset      | Input     | PLB system reset                                            |
| PLB_abort         | Input     | Abort transaction prior to address acknowledge              |
| PLB_ABus[0:31]    | Input     | PLB address bus                                             |
| PLB_BE[0:7]       | Input     | Bytes enables                                               |
| PLB_busLock       | Input     | Used to lock the bus                                        |
| PLB_compress      | Input     | Indicates compressed data transfer                          |
| PLB_guarded       | Input     | Indicates a guarded transaction                             |
| PLB_lockErr       | Input     | PLB lock error indicator                                    |
| PLB_masterID[0:3] | Input     | PLB current master identifier                               |
| PLB_Msize[0:1]    | Input     | Size of master (bit-width)                                  |
| PLB_ordered       | Input     | Prevents other writes interrupting                          |
| PLB_PAVValid      | Input     | Indicates a primary address valid, initiating a transaction |
| PLB_RNW           | Input     | Indicates read (1) or write (0)                             |
| PLB_size[0:1]     | Input     | Encoded number of words to transfer                         |
| PLB_type          | Input     | Different transfer types (e.g. DMA)                         |
| Sl_addrAck        | Output    | Slave acknowledges the address                              |

Table 16-1: SRAM Controller Signals (Continued)

| Name             | Direction | Description                                                       |
|------------------|-----------|-------------------------------------------------------------------|
| Sl_Mbusy[0:15]   | Output    | Slave decoding of Master ID, indicates master is busy - one hot   |
| Sl_Merr[0:15]    | Output    | Slave error with this master - one hot                            |
| Sl_rearbitrate   | Output    | Restart the transaction                                           |
| Sl_Ssize[0:1]    | Output    | Slave size - indicates width                                      |
| Sl_wait          | Output    | The slave will respond, but not soon                              |
| PLB_rdPrim       | Input     | Secondary address now primary read                                |
| PLB_SAVValid     | Input     | Secondary address valid, allows for pipelining and pre-processing |
| PLB_wrPrim       | Input     | Secondary address now primary write                               |
| PLB_wrBurst      | Input     | Write burst unit size (e.g. byte)                                 |
| PLB_wrDBus[0:63] | Input     | Write data bus                                                    |
| Sl_wrBTerm       | Output    | Master must terminate current burst                               |
| Sl_wrComp        | Output    | Write transfer complete                                           |
| Sl_wrDAck        | Output    | Write data acknowledge                                            |
| PLB_rdBurst      | Input     | Read burst unit size (e.g. byte)                                  |
| Sl_rdBTerm       | Output    | Master must terminate current burst                               |
| Sl_rdComp        | Output    | Read transfer complete                                            |
| Sl_rdDAck        | Output    | Read data acknowledge                                             |
| Sl_rdbus[0:63]   | Output    | Read data bus                                                     |
| Sl_rdWdAddr[0:3] | Output    | Current address being read for <i>target word first</i> transfer  |
| SRAM_ad[23:0]    | Output    | SRAM address                                                      |
| SRAM_dq_i[63:0]  | Input     | SRAM data input                                                   |
| SRAM_dq_o[63:0]  | Output    | SRAM data output                                                  |
| SRAM_dq_t[63:0]  | Output    | SRAM data 3-state enable                                          |
| SRAM_ben[7:0]    | Output    | SRAM byte enable (active low)                                     |
| SRAM_wen[3:0]    | Output    | SRAM write enable (active low)                                    |
| SRAM_oen         | Output    | SRAM output enable (active low)                                   |
| SRAM_csn         | Output    | SRAM chip select (active low)                                     |
| FLASH_csn        | Output    | FLASH chip select (active low)                                    |

The SRAM Controller module receives PLB requests to read and write to external SRAM/Flash. With 64-bit data and 24-bit address, the controller can interface up to 128 MB of SRAMS and 128 MB of Flash.

## Address Mapping

The PLB SRAM/Flash Controller has two address decoders: one for SRAM and one for Flash. See [Table 16-2](#).

Table 16-2: SRAM/Flash Controller Parameters

| Address Decoder | Parameters        | Default Setting | Default PLB Space                        | Definition                                    |
|-----------------|-------------------|-----------------|------------------------------------------|-----------------------------------------------|
| SRAM            | C_SRAM_LSB_ADDR   | 10              | From 0x1000_0000 to 0x1000_0000 (2 MB)   | Defines the size of the SRAM address decoder  |
|                 | C_SRAM_BASE_ADDR  | 0x1000_0000     |                                          | Defines the starting address for SRAM         |
| Flash           | C_FLASH_LSB_ADDR  | 4               | From 0x1800_0000 to 0x1FFF_FFFF (128 MB) | Defines the size of the Flash address decoder |
|                 | C_FLASH_BASE_ADDR | 0x1800_0000     |                                          | Defines the starting address for Flash        |

By default, C\_SRAM\_LSB\_ADDR is set to 10, and C\_SRAM\_BASE\_ADDR is set to 0x1000\_0000. This defines the SRAM space of 2 MB starting from 0x1000\_0000 to 0x101F\_FFFF.

C\_FLASH\_LSB\_ADDR is set to 4, and C\_FLASH\_BASE\_ADDR is set to 0x1800\_0000. This defines the Flash space of 128 MB starting from 0x1800\_0000 to 0x1FFF\_FFFF.

The mapping between the PLB address and the SRAM address is shown in [Figure 16-2](#).

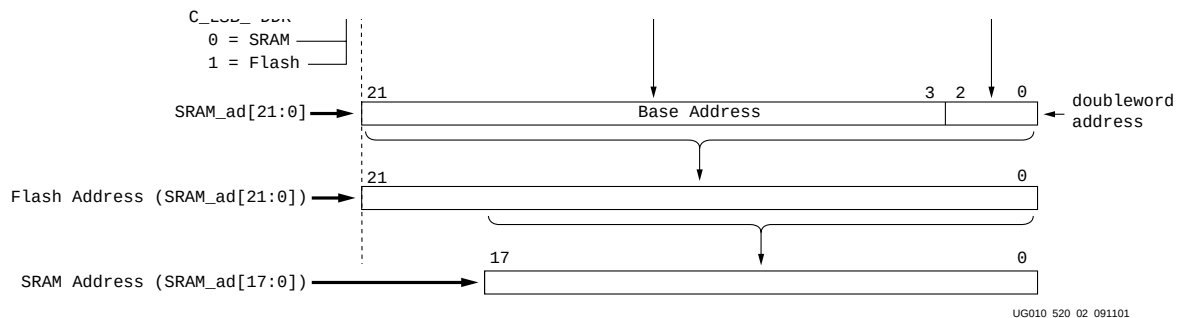


Figure 16-2: Mapping between the PLB and External SRAM Address

The SRAM/Flash address is comprised of a base address and a word address. Since the SRAM/Flash Controller handles data in 64-bit chunks (8 bytes), the last three bits of the PLB address are dropped. The SRAM/Flash base address always maps to PLB\_ABus[7:25]. The lower three bits of the SRAM/Flash address comprise the doubleword address.

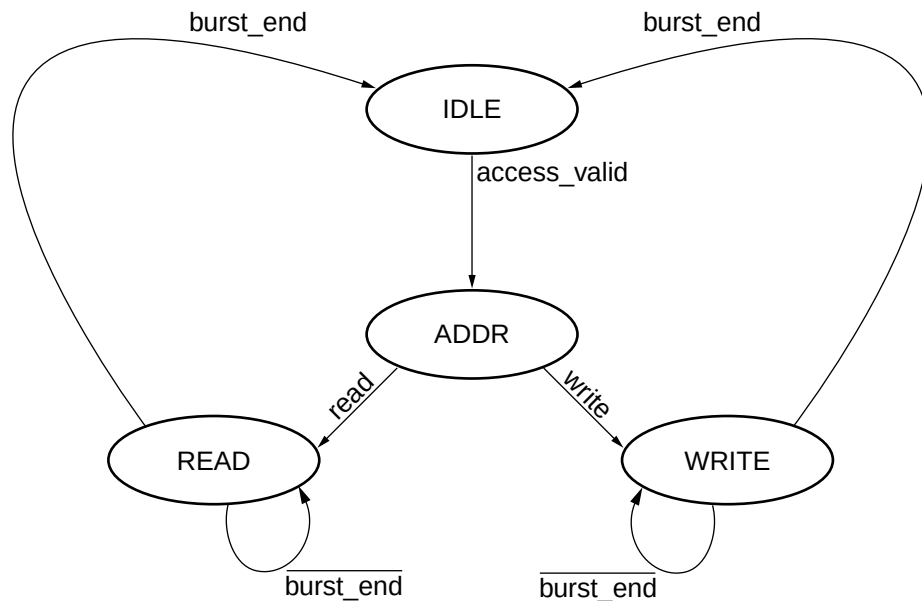
For read transfers, the controller supports target word first, so the SRAM/Flash word address maps to PLB\_ABus[26:28] and will wrap around for the different burst types.

For write transfers, the address is required to align on the word boundary. For example, for a burst of two, PLB\_ABus[28] must be "0", for bursts of 4, PLB\_ABus[27:28] must be "00", and for bursts of 8, PLB\_ABus[26:28] must be "000".

## Controller State Machine

The state machine for the SRAM/Flash Controller is small and simple. It has only four states: IDLE, ADDR, READ, and WRITE.

The controller initializes in the IDLE state. When it detects a valid access (correct address, transaction type, and burst sizes), it advances to the ADDR state and latches the current address, burst size, and master ID. The controller then goes to either the READ or WRITE state depending on the transaction type. It stays in either the READ or WRITE state until the burst is completed. When the burst is completed, the controller returns to the IDLE state and waits for the next valid access.



UG010\_520\_03\_090601

Figure 16-3: Controller State Machine

## SRAM Timing Diagrams

### Write Cycle

A timing diagram for a cacheline write to the SRAM of two doublewords is shown in Figure 16-4.

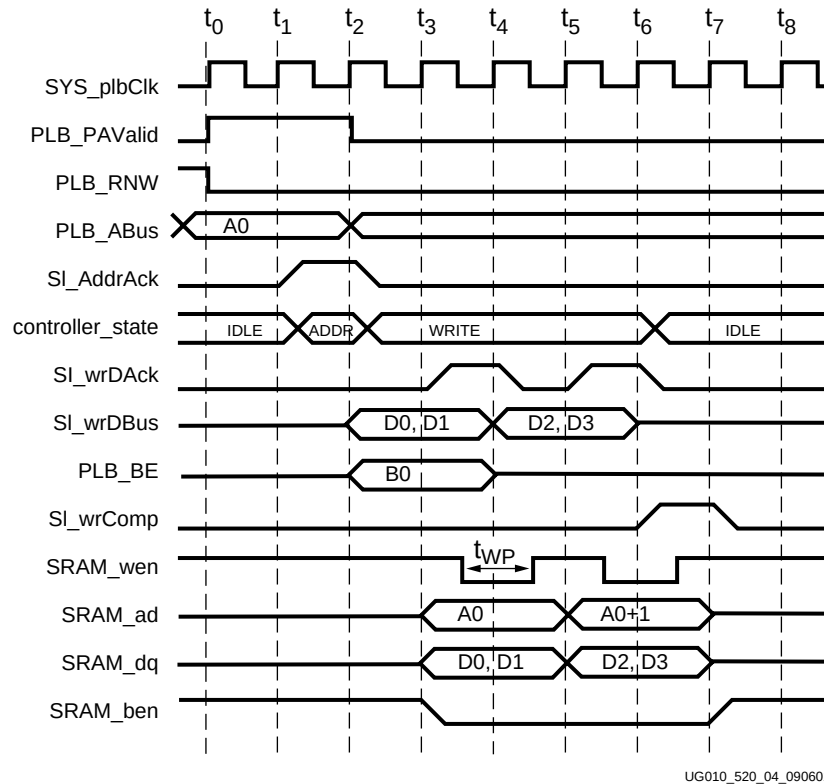


Figure 16-4: Timing Diagram for Cacheline Write of Two Doublewords

1. At  $t_0$ , the controller sees PLB\_PValid, valid address, and transfer type.
2. At  $t_1$ , it goes from the IDLE to the ADDR state. The controller latches the write address, burst type, and master ID, generating SI\_AddrAck.
3. At  $t_2$ , the controller goes to the WRITE state. During this time, the controller accepts data from the PLB and generates SRAM\_wen pulses, enabling address and data for the SRAM.

Note: To achieve 100 MHz operation in -10 SRAM parts, the write pulse width ( $t_{WP}$ ) is designed using a 1 clock period. It takes two clock cycles to write each doubleword. To signal the PLB to give one doubleword every two clock cycles, the controller generates one pulse of SI\_wrDack every two clock cycles.

The PLB\_BE is transferred to SRAM\_ben only during single beat write transactions.

The SRAM address is automatically incremented for burst transactions. To guarantee SRAM setup times, the SRAM address, data, and byte enable are generated on the positive edge of clk, while the write enable goes low on the negative edge of clk.

When the controller completes the current burst, it generates SI\_wrComp to signal the completion of the current transaction and returns to the IDLE state.

## Read Cycle

A timing diagram for a cacheline read from the SRAM of two doublewords is shown in Figure 16-5.

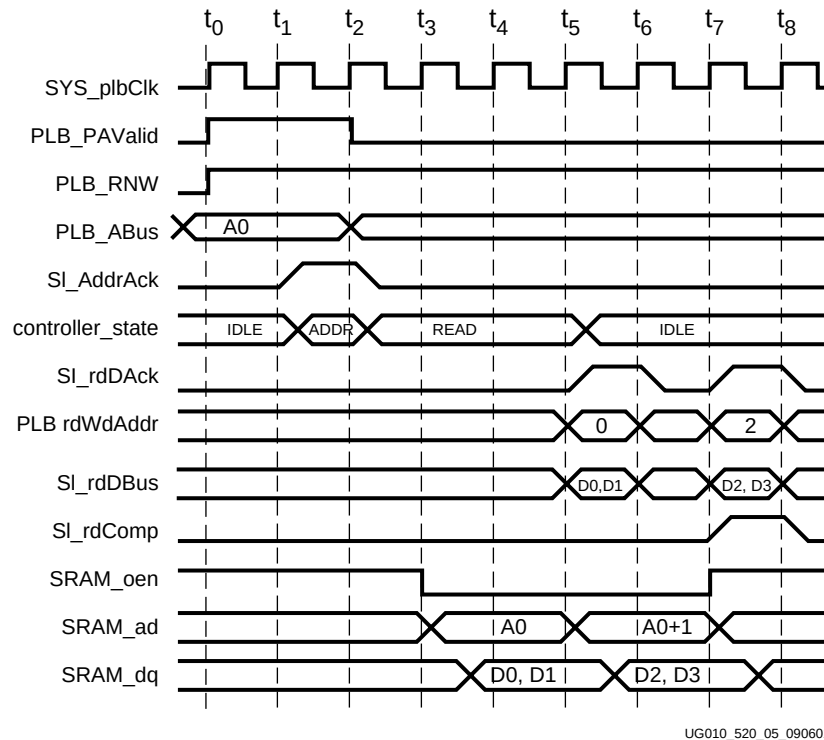


Figure 16-5: Timing Diagram for Cacheline Read of Two Doublewords

1. At  $t_0$ , the controller sees PLB\_PAVValid, valid address, and transfer type.
2. At  $t_1$ , it goes from IDLE to ADDR state. The controller latches the write address, burst type, master ID, generating SI\_AddrAck.
3. At  $t_2$ , the controller goes to the READ state. The controller generates SRAM\_oen and the address for the SRAM at  $t_3$ .

Note: For a -10 SRAM part, the address access time ( $t_{AA}$ ) is 10 ns. To guarantee that valid data can be captured correctly in the FPGA, the address is held stable for two clock cycles. Data is captured 1.5 clock cycles after the address is presented to the SRAM. The first read data is available at SI\_rdDBus four clock cycles after SI\_AddrAck. The PLB expects to see SI\_rdDack asserted for one clock cycle for each doubleword. Since the SRAM controller delivers one doubleword per two clock cycles, SI\_rdDack is pulsed throughout a burst transfer.

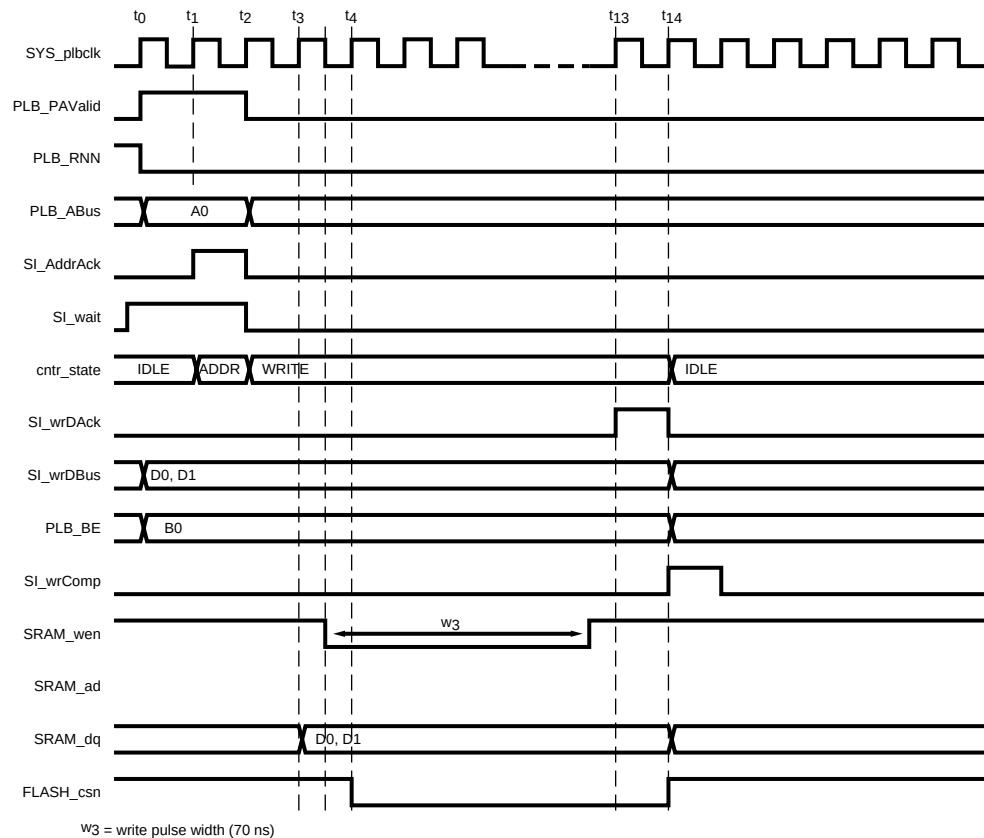
When the controller completes the current read transfer, it generates SI\_wrComp and returns to the IDLE state.



## Flash Timing Diagrams

The Flash READ and WRITE timing diagrams are similar to the SRAMs, except that Flash's pulse width and access time are much longer than that of SRAMs.

In the reference design, the BYTEB input of the StrataFlash is tied high, enabling 16-bit accesses. Because of this, the controller does not support byte write to the Flash. The SRAM\_be (byte enable) outputs are not used. The PLB\_BE is decoded into 4-bit SRAM\_wen, selecting the different Flash devices for halfword, singleword, and doubleword accesses.



ug010\_520\_06\_092101

Figure 16-6: Flash Write Cycle

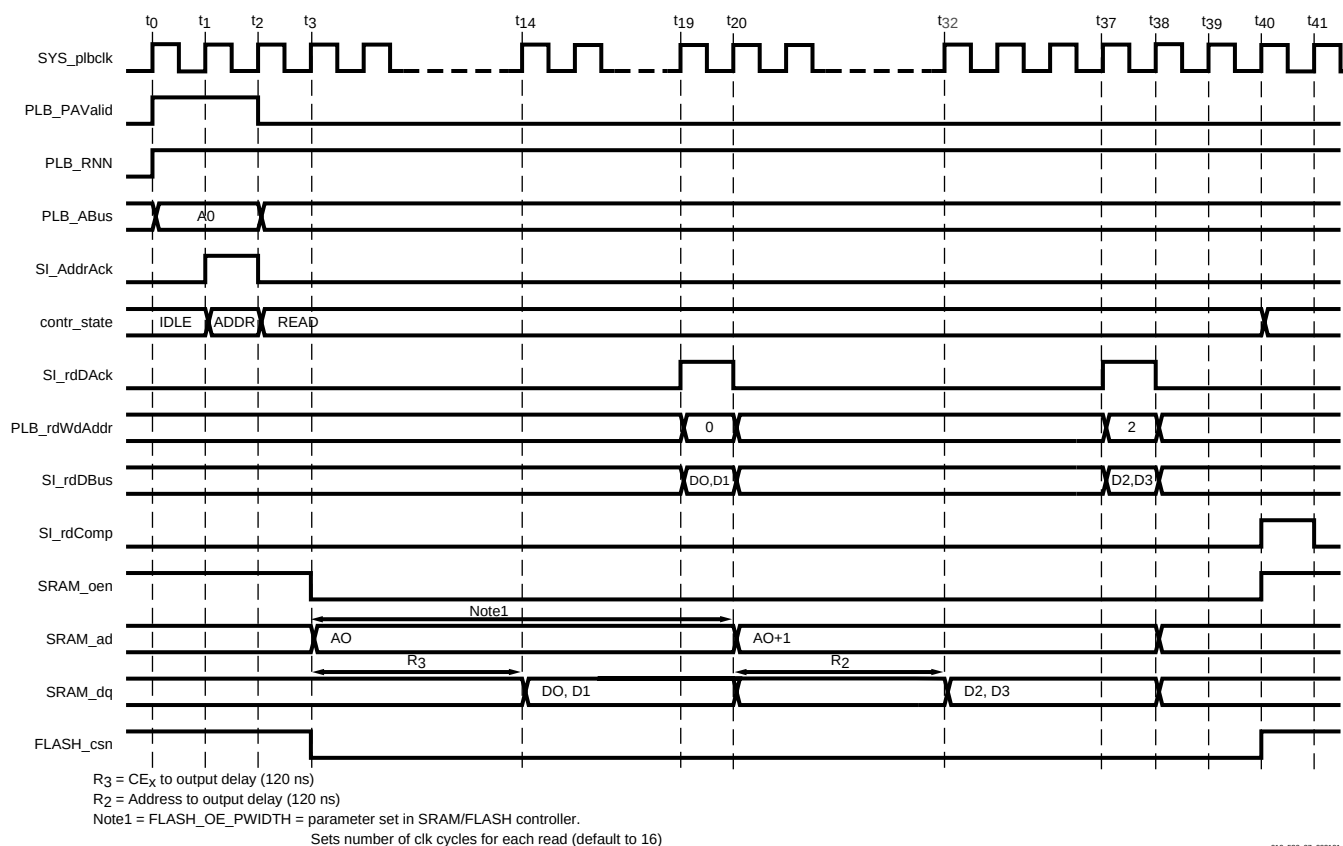


Figure 16-7: Flash Read Cycle

## Conclusion

The SRAM/Flash Controller module demonstrates the interfaces to the Processor Local Bus (PLB). It provides access to external SRAMs for systems implemented with a PowerPC core. The source Verilog code is provided, and can be easily modified to meet the requirements of a user's design.

## Further Reading

The Virtex-II Pro™ Developer's Kit contains a wealth of valuable information that will assist you in your design efforts. The documentation contained within the eight volumes is organized to assist you in quickly finding relevant materials. To obtain the most recent revision of this documentation, please see

[http://support.xilinx.com/xlnx/xil\\_tt\\_product.jsp?sProduct=Virtex-II+Pro](http://support.xilinx.com/xlnx/xil_tt_product.jsp?sProduct=Virtex-II+Pro).

## Documentation Provided by Xilinx

*Virtex-II Pro™ Platform FPGA Handbook*

*Virtex-II Pro™ Platform FPGA Developer's Kit Documentation:*

*Volume 1, Advance Product Specification*

*Volume 2, PowerPC® 405 Processor*

*Volume 3, Rocket I/O™ Transceiver User Guide*

*Volume 4, Design Flow*  
*Volume 5, Software Development Tools*  
*Volume 6, Reference Systems*  
*Volume 7, Hardware IP Specifications*  
*Volume 8, Software IP and Applications*

## IBM® CoreConnect™ Documentation

The Virtex-II Pro Developer's Kit integrates seamlessly with the IBM CoreConnect Toolkit. This toolkit is not included with the Developer's Kit, but is required if bus functional simulation is desired. The toolkit provides a number of features which enhance design productivity and allow you to get the most from the Developer's Kit. To obtain the toolkit, you must be a licensee of the IBM CoreConnect Bus Architecture. Licensing CoreConnect provides access to a wealth of documentation, Bus Functional Models, Hardware IP, and the toolkit.

Xilinx provides a Web-based licensing mechanism that allows you to obtain the CoreConnect toolkit from our website. To license CoreConnect, use an Internet browser to access [http://www.xilinx.com/ipcenter/processor\\_central/register\\_coreconnect.htm](http://www.xilinx.com/ipcenter/processor_central/register_coreconnect.htm). Once your request has been approved (typically within 24 hours), you will receive an e-mail granting access to a protected website. You may then download the toolkit.

If you prefer, you can also license CoreConnect directly from IBM.

If you would like further information on CoreConnect Bus Architecture, please see IBM's CoreConnect website at <http://www.ibm.com/chips/products/coreconnect>.

Once you have licensed the CoreConnect toolkit, and installed it with the Developer's Kit, the following documents will be available to you in the following locations:

### IBM CoreConnect Bus Architecture Specifications

*IBM CoreConnect Processor Local Bus (PLB) Architecture Specification*  
see \$CORECONNECT/published/corecon/64bitPlbBus.pdf

*IBM CoreConnect On-chip Peripheral Bus (OPB) Architecture Specification*  
see \$CORECONNECT/published/corecon/OpbBus.pdf

*IBM CoreConnect Device Control Register (DCR) Bus Architecture Specification*  
see \$CORECONNECT/published/corecon/DcrBus.pdf

### IBM CoreConnect Toolkit Documentation

*PLB Bus Functional Model Toolkit - User's Manual*  
see \$CORECONNECT/published/corecon/PlbToolkit.pdf

*OPB Bus Functional Model Toolkit - User's Manual*  
see \$CORECONNECT/published/corecon/OpbToolkit.pdf

*DCR Bus Functional Model Toolkit - User's Manual*  
see \$CORECONNECT/published/corecon/DcrToolkit.pdf

*CoreConnect Test Generator - User's Manual*  
see \$CORECONNECT/published/corecon/ctg.pdf

**Note:** \$CORECONNECT is an environment variable that is created when installing the Developer's Kit or CoreConnect Toolkit.

## Software Development Documentation

There are many sources of documentation available for those who wish to learn more about Software Development. It is recommended that a web search be conducted using a favorite search engine for keywords such as "PowerPC+Software+Development".

Alternatively, a technical bookstore should be able to provide many valuable resources. The books listed below are a very small fraction of those available.

## Books About Programming in C

There are many good books about the C and C++ programming languages. A few of these are listed below:

### Books about C

Kernigham Brian, Ritchie Dennis. 1988. *The C Programming Language*. Prentice Hall

Wang Paul. 1992. *An Introduction to ANSI C on Unix*. Wadsworth.

Kumar Ram, Agrawal Rakesh. 1992. *Programming in ANSI C*. West Publishing Company.

### Books about C++

Stroustrup Bjarne. 1991. *The C++ Programming Language*. Addison-Wesley.

Lafore Robert. 1995. *Object-Oriented Programming in C++*. Waite Group Press.

## Online Documents About Programming in C

The Internet offers plenty of documentation about how to program in C and C++. One of the best approaches to finding documentation online is to use a search engine (such as Google, <http://www.google.com>) and search on "introduction to programming in C."



## PLB ZBT SRAM Controller

### Overview

This document describes the ZBT SRAM Controller module that interfaces to the IBM Core Connect™ Processor Local Bus (PLB). This document assumes that the user is familiar with PowerPC® and PLB based designs as well as the basics of ZBT SRAM.

The ZBT SRAM Controller module connects to the PLB as a slave device. A typical configuration is shown in **Figure 17-1**.

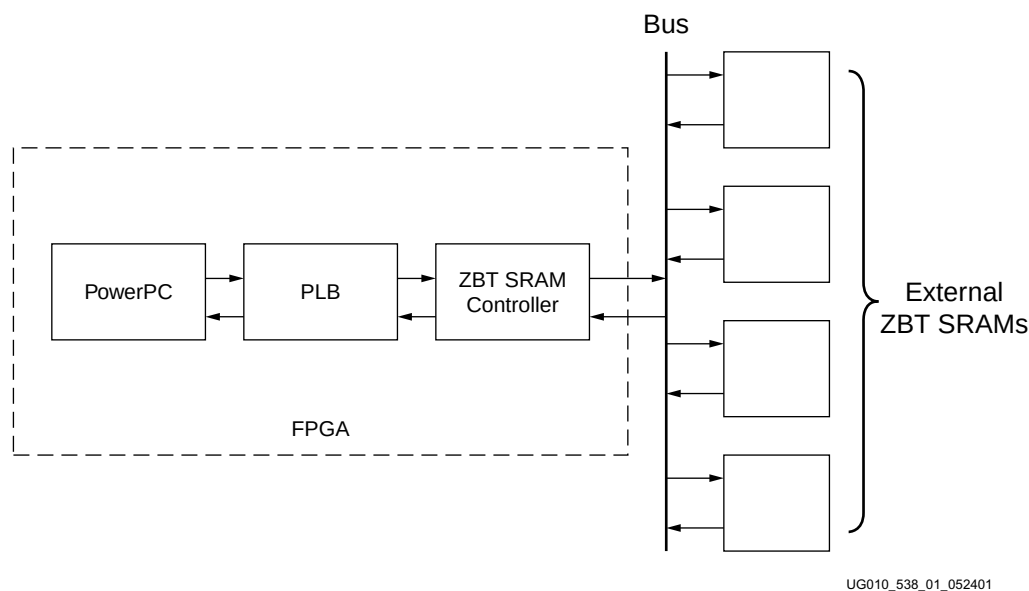


Figure 17-1: ZBT SRAM Controller Module Block Diagram

### Related Documents

- PLB Architecture Specification
- Micron *ZBT SRAM Data Sheet*, [MT55L512L18P 2.pdf](#)

### Features

The ZBT SRAM Controller module includes the following features:

- Tuned for the PLB interface
- 64-bit read/write data

- User-configurable size (defaults to 1 MB of ZBT SRAM)
- Supports single data beat or cacheline data transfers (burst of 2, 4, or 8 words)
- Supports *target word first* read burst for optimized CPU operation
- Partial support for secondary address processing

The current implementation can be tuned for synthesis within the Virtex-II fabric to operate at a 100Mhz. Additional functionality can also be added to more fully exploit the PLB architecture and the ZBT capabilities.

## Module Port Interface

The ZBT SRAM Controller module receives instructions from the PLB to read and write to external ZBT SRAMs. With 64-bit data and a default 17-bit address, the controller can interface to 1 MB of ZBT SRAM.

**Table 17-1** lists the pins in the ZBT SRAM interface. For a more detailed description of PLB pins, see the IBM document "Processor Local Bus Architecture Specifications, 64-Bit Implementation V3.3." For more detailed information regarding the pins of the ZBT SRAM, see the Micron ZBT SRAM Data Sheet at

[http://download.micron.com/pdf/datasheets/sram/MT55L512L18P\\_2.pdf](http://download.micron.com/pdf/datasheets/sram/MT55L512L18P_2.pdf).

## PLB Interface Signals

**Table 17-1** shows the signals for the PLB interface.

**Table 17-1: PLB Interface Pin Description**

| Name           | Direction | Description                                                     |
|----------------|-----------|-----------------------------------------------------------------|
| SYS_plbClk     | Input     | System clock                                                    |
| SYS_plbReset   | Input     | System reset                                                    |
| PLB_abort      | Input     | Abort transaction prior to address acknowledge                  |
| PLB_ABus[0:31] | Input     | PLB address bus                                                 |
| PLB_BE[0:7]    | Input     | Byte enables                                                    |
| PLB_busLock    | Input     | Used to lock the bus                                            |
| PLB_compress   | Input     | Indicates compressed data transfer                              |
| LB_guarded     | Input     | Indicates a guarded transaction                                 |
| PLB_lockErr    | Input     | The SEAR or SESR should be locked PLB_masterID                  |
| PLB_Msize[0:1] | Input     | Size of master (bit-width)                                      |
| PLB_ordered    | Input     | Prevents other writes interrupting                              |
| PLB_PAVValid   | Input     | Indicates a primary address valid (Initiates a transaction)     |
| PLB_RNW        | Input     | Indicates read (1) or write (0)                                 |
| PLB_size[0:3]  | Input     | Encoded number of words to transfer                             |
| PLB_type[0:2]  | Input     | Different transfers types (e.g. DMA)                            |
| Sl_addrAck     | Output    | Slave acknowledges the address                                  |
| Sl_MBusy[0:15] | Output    | Slave decoding of Master ID, indicates master is busy - one hot |
| Sl_MErr[0:15]  | Output    | Slave error with this master - one hot                          |
| Sl_rearbitrate | Output    | Restart the transaction                                         |

Table 17-1: PLB Interface Pin Description (Continued)

| Name             | Direction | Description                                                       |
|------------------|-----------|-------------------------------------------------------------------|
| Sl_SSize[0:1]    | Output    | Slave size, indicates width                                       |
| Sl_wait          | Output    | The slave will respond, but not soon                              |
| PLB_rdPrim       | Input     | Secondary address now primary read                                |
| PLB_SAVValid     | Input     | Secondary address valid, allows for pipelining and pre-processing |
| PLB_wrPrim       | Input     | Secondary address now primary write                               |
| PLB_wrBurst      | Input     | Write burst unit size (e.g. byte)                                 |
| PLB_wrDBus[0:63] | Input     | Write data bus                                                    |
| Sl_wrBTerm       | Output    | Master must terminate current burst                               |
| Sl_wrComp        | Output    | Write transfer complete                                           |
| Sl_wrDAck        | Output    | Write data acknowledge                                            |
| PLB_rdBurst      | Input     | Read burst unit size (e.g. byte)                                  |
| Sl_rdBTerm       | Output    | Master must terminate current burst                               |
| Sl_rdComp        | Output    | Read transfer complete                                            |
| Sl_rDAck         | Output    | Read data acknowledge                                             |
| Sl_rDBus[0:63]   | Output    | Read data bus                                                     |
| Sl_rdWdAddr[0:3] | Output    | Current address being read for Target word first transfer         |

## ZBT SRAM Interface Signals

Table 17-2 shows the signals for the ZBT SRAM interface.

Table 17-2: ZBT SRAM Interface Pin Description

| Name              | Direction | Description                                       |
|-------------------|-----------|---------------------------------------------------|
| ZBT_ad[17:0]      | Output    | Address                                           |
| ZBT_adv_ld_n      | Output    | Advance (1) or load(0) control signals            |
| ZBT_be_n[7:0]     | Output    | Byte enables (active low)                         |
| ZBT_ce_n          | Output    | Clock enable (active low)                         |
| ZBT_ce1_n         | Output    | Chip enable 1 (active low)                        |
| ZBT_ce2           | Output    | Chip enable 2 (active high)                       |
| ZBT_ce2_n         | Output    | Chip enable 2 (active low)                        |
| ZBT_lbo_n         | Output    | Linear (0) or interleaved (1) address burst order |
| ZBT_oe_n          | Output    | Output enables                                    |
| ZBT_rw_n          | Output    | Read (1) or write (0)                             |
| ZBT_dq_i[63:0]    | Input     | ZBT data input                                    |
| ZBT_dq_o[63:0]    | Output    | ZBT data output                                   |
| ZBT_dq_oe_n[63:0] | Output    | ZBT data 3-state enable                           |



## Read and Write Transactions

Figure 17-3 and Figure 17-2 show the read and write transactions for the ZBT SRAM Controller module.

The **PLB\_SAVAlid** command indicates that a secondary address is available. This is used to bypass the IDLE state and go directly to the ADDR state after a read or write transaction is complete.

The transaction control signals are read into the ZBT SRAM if the **ZBT\_adv\_ld\_n** and **ZBT\_ce2** signals are active during the same clock cycle. If the **ZBT\_adv\_ld\_n** signal is active (0) and the **ZBT\_ce2** signal is inactive (0), the ZBT will be inactive (other than finishing transactions that are already in the pipeline).

Some intricacies are required to provide for *target word first* reads. See the **Address Mapping** section for how this is handled using the **ZBT\_lbo\_n** signal.

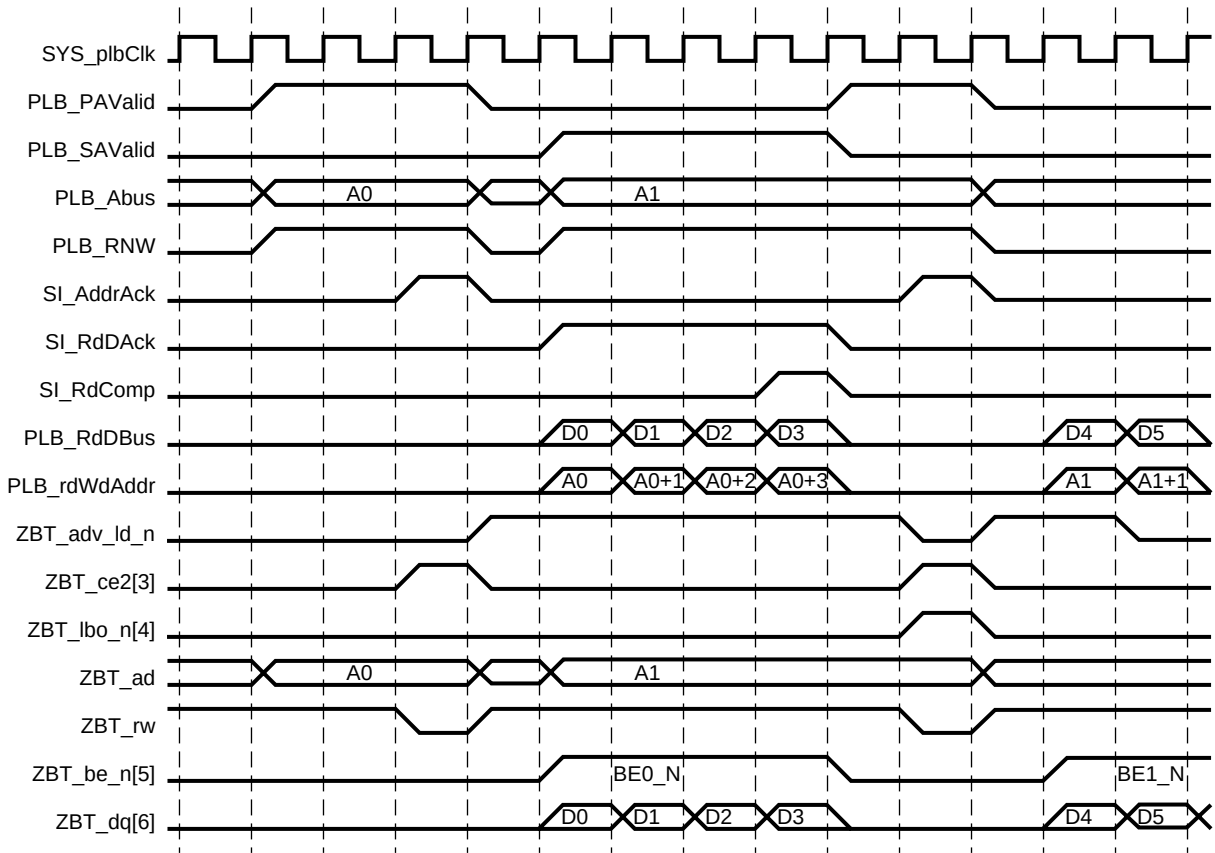
Byte enables, **ZBT\_be\_n** (not shown), are handled concurrently with the data, not with the control signals. Therefore, byte enables must be active during every cycle that data is presented to the ZBT SRAM.

The ZBT SRAM requires two clock cycles to retrieve requested data from storage, therefore, the data lines follow the control signals by two clock cycles.

Although the data is presented to the ZBT SRAM on the cycle it is available from the PLB, it will not be through the ZBT pipeline for two clock cycles. However, this does not impact the user other than to ensure that the clock enable stays high.

# Read Transactions

Figure 17-2 shows the timing diagram for a read transaction. The basic methodology for read transactions depends on the size of the transaction, so two transactions are shown: a four doubleword transfer followed by a two doubleword transfer.

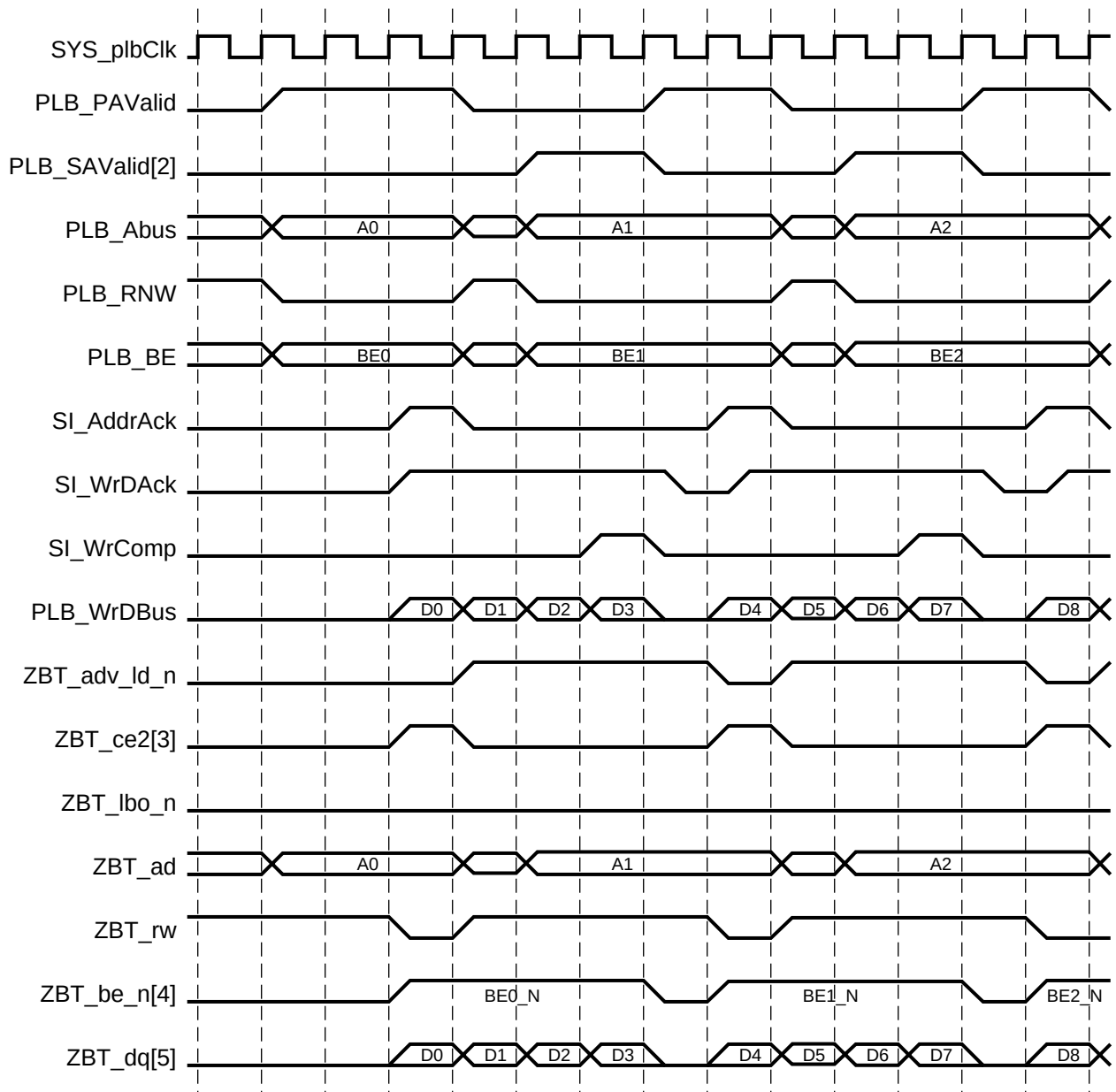


Note: Generally, there is a  $\leq 2.0$  ns setup time, and a  $\leq 0.5$  ns hold time on signals to the ZBT SRAM. UG010\_538\_03\_060401

Figure 17-2: Read Transaction Timing

## Write Transactions

Figure 17-3 shows the timing diagram for a write transaction. Since the basic methodology for write transactions does not significantly change with the size of the transaction, a consecutive pair of four doubleword transfers is shown.



Note: Generally, there is a  $\leq 2.0$  ns setup time, and a  $\leq 0.5$  ns hold time on signals to the ZBT SRAM.

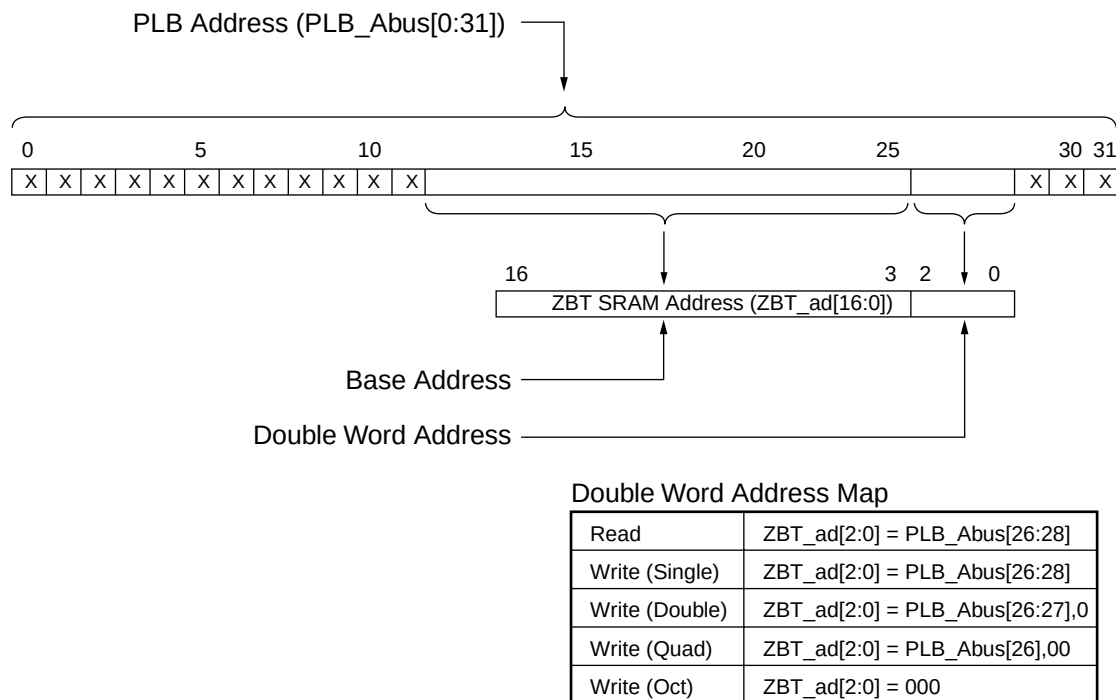
UG010\_538\_02\_060401

Figure 17-3: Write Transaction Timing

## Address Mapping

The ZBT SRAM Controller module interfaces to the external ZBT SRAMs with a default 17-bit address that is configurable using the **C\_LSB\_ADDR** parameter. This 17-bit ZBT address corresponds to a 20-bit PLB address, as the ZBT is 64-bit (8-byte) addressed, while the PLB is byte addressed. The upper 12 bits, also configurable based on the **C\_LSB\_ADDR** parameter, are used to determine if the ZBT SRAM is being addressed by a transaction.

The mapping between the PLB address and the ZBT SRAM address is shown in Figure 17-4.



UG010\_538\_04\_052401

Figure 17-4: Mapping Between the ZBT SRAM Controller and External ZBT SRAMs

The ZBT SRAM address is comprised of a base address and a doubleword address. Since the ZBT SRAM Controller module handles data in 64-bit chunks (8 bytes), and the PLB is based on byte addresses, the ZBT SRAM ignores the last three bits. The ZBT SRAM base address maps to **PLB\_ABus[12:28]**. The least significant three bits of the ZBT SRAM address comprises the double word address.

The ZBT SRAM word address value is dependent on the type and size of the transfer, as indicated in the Double Word Address Map in Figure 17-4. For a Write transfer, the address is supposed to align on the transfer size boundary, therefore some or all of the word address is ignored (therefore maps to 0), depending on the burst size. For a burst of 8 (which is not currently supported), all of the ZBT SRAM word address maps to 0. For bursts of 4, **ZBT SRAM\_ad[1:0]** maps to 0, and for bursts of 2, **ZBT SRAM\_ad[0]** maps to 0. For a single 64-bit transfer, **ZBT\_ad** is fully mapped by the **PLB\_Abus**.

For a Read transfer, the controller supports target word first, so the ZBT SRAM word address maps to **PLB\_ABus[26:28]** for the first transfer of a burst, and will wrap around for the subsequent transaction of a burst. There can be some difficulties when starting on a non-zero, doubleword address. For ZBT SRAM, this is not a problem for a four doubleword burst. The linear burst built into ZBT SRAM will increment and wrap the address for a four transfer burst. However, if the address in **PLB\_ABus[26:28]** is 001 during a doubleword transfer, it will automatically advance to 010, when proper wrapping would produce 000. If it is 011, it will advance to 000, when 010 is required. To correct this, use the interleaved burst order by toggling the **ZBT\_lbo\_n** signal high. For more details on the functioning of the linear burst order versus interleaved burst order, see the Micron ZBT SRAM Data Sheet at

[http://download.micron.com/pdf/datasheets/sram/MT55L512L18P\\_2.pdf](http://download.micron.com/pdf/datasheets/sram/MT55L512L18P_2.pdf).

## Controller State Machine

An overview of the state machine is shown in Figure 17-5. The dashed lines indicate an automatic sequence.

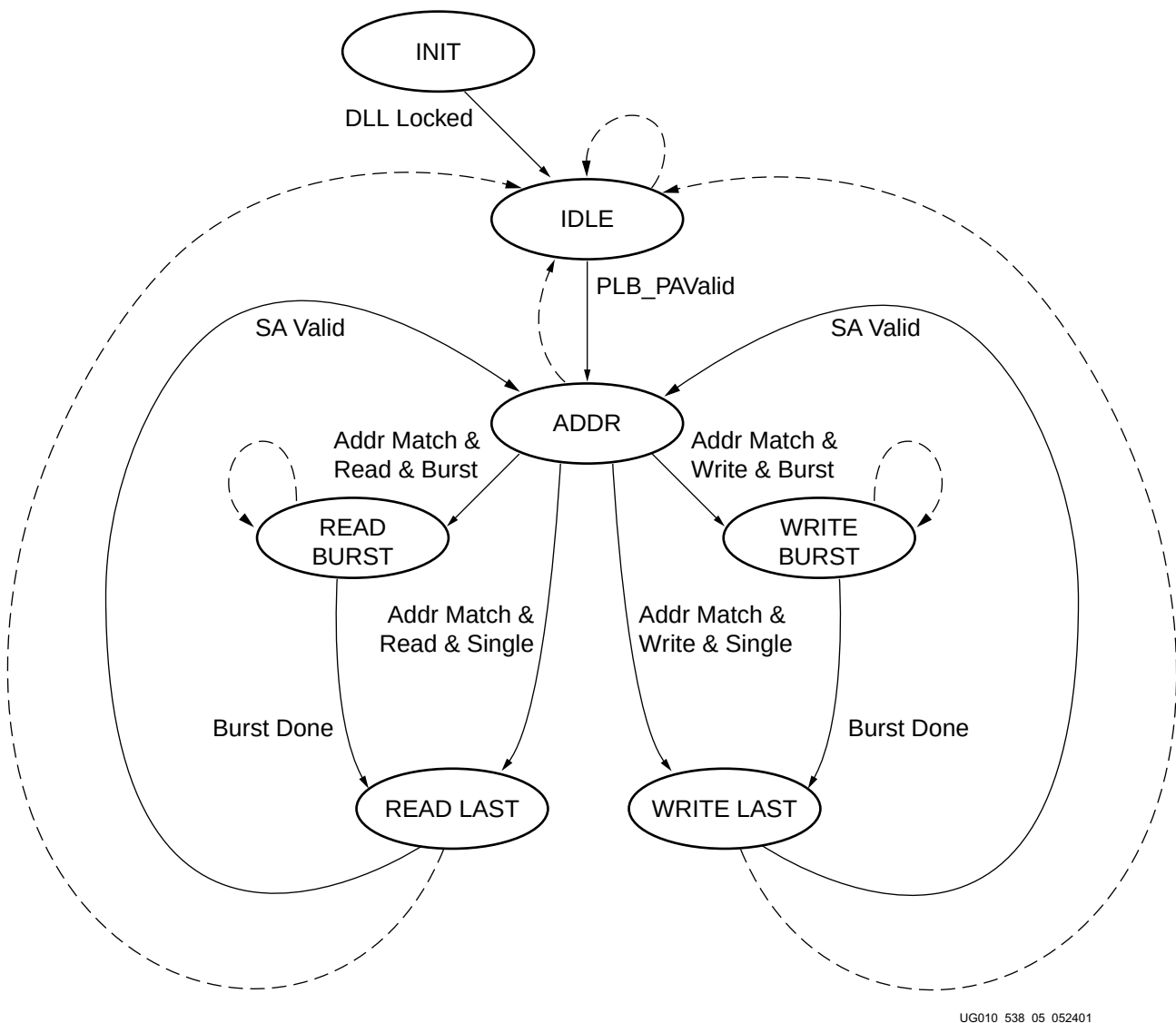


Figure 17-5: Controller State Machine Overview

### INIT

The controller starts here on reset, and waits to go to the IDLE state until the ZBT DLL locks, synchronizing the external ZBT SRAM with the internal clocks. All of the ZBT SRAM signals are held inactive.

### IDLE

While waiting for a transaction to begin, the controller state machine stays in the IDLE state. When a **PValid** is received, it latches the address and control signals, and goes to the ADDR state. All of the ZBT SRAM signals are held inactive, except for **ZBT\_CE**, which must be enabled to let the pipelined transaction complete.

## ADDR

In the ADDR state, the control signals and address are decoded to determine if the transaction is for the ZBT SRAM, and to determine the type of transaction. There are four different transaction types: Read Burst, Read Single, Write Burst and Write Single. These each correspond to a state after the ADDR state. When leaving the ADDR state for one of these transaction states, the controller will send a **SL\_AddrAck**. If the address or control signals do not match, the state machine returns to the IDLE state.

## READ BURST

The READ BURST state is where a burst read is initiated, and the controller will stay there until the second to last transaction. During this state, the **SL\_rdAck** is initiated (it is pipelined, so is active two cycles later). For the final transaction, it goes to the READ LAST state (also known as READ SINGLE). In the READ BURST state, there is some added complexity due to support of the Read Target Word first, as discussed in the **Address Mapping** section.

## READ LAST

The READ LAST state is the last read of a read burst. In the case of a single beat transfer, for which only one read state is needed, the READ LAST state acts as a READ SINGLE state. During this state, the last read transaction is done, and both a **SL\_rdAck** and a **SL\_RdComp** are initiated (both are pipelined, so will be active two cycles later). If there is a **PLB\_SAVa lid** (Secondary Address Valid) signal during this state, it enters the ADDR state. Otherwise, it defaults to the IDLE state.

## WRITE BURST

The WRITE BURST state is where a burst write is initiated, and the controller will remain until the second-to-last transaction. During this state, the **SL\_wrAck** is set high to indicate that the data on the **PLB\_wrDBus** has been read. For the final transaction, it goes to the WRITE LAST state (also known as WRITE SINGLE).

## WRITE LAST

The WRITE LAST state is the last write of a write burst. In the case of a single beat transfer, for which only one write state is needed, the WRITE LAST state acts as a WRITE SINGLE state. During this state, the last write transaction is done, and both a **SL\_wrAck** and a **SL\_wrComp** are set high to indicated that the data on the **PLB\_wrDBus** has been read, and this is the last write of the burst. If there is a **PLB\_SAVa lid** (Secondary Address Valid) signal during this state, the next state will be ADDR. Otherwise, it defaults to the IDLE state.

## Conclusion

The ZBT SRAM controller module reference design demonstrates the interfaces to the Processor Local Bus. It provides access to external ZBT SRAMs for systems implemented with a PowerPC core in the Virtex-II fabric. The source Verilog code is provided, and can be easily modified to meet the requirements of a user's design.



## PLB DDR SDRAM Controller

### Overview

This document describes the DDR SDRAM Controller module that interfaces to the IBM Core Connect™ Processor Local Bus (PLB). This document assumes that the user is familiar with PowerPC® and PLB based designs as well as the basics of DDR SDRAM. The DDR SDRAM Controller module connects to the PLB as a slave device. A typical configuration is shown in **Figure 18-1**.

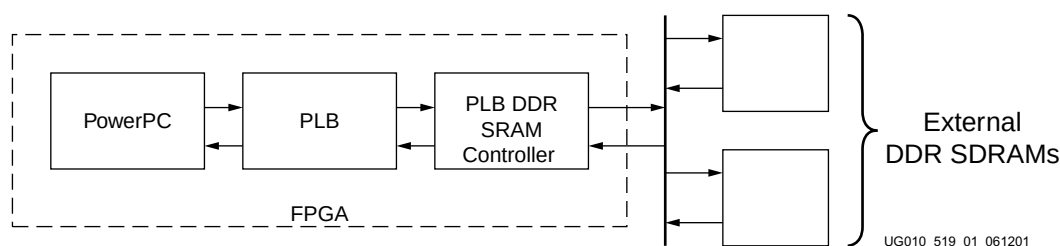


Figure 18-1: PLB DDR SDRAM Controller Module Block Diagram

### Related Documents

The following documents provide additional useful information:

- *PLB Architecture Specification*
- *DDR SDRAM Data Sheet* (<http://www.micron.com>)

### Features

The DDR SDRAM Controller includes the following features:

- Tuned for the PLB interface
- Interfaces to PLB with 64-bit data, single data rate
- Interfaces to DDR SDRAM with 32-bit data, double data rate
- Supports CAS latency of 2
- Supports 32 MB SDRAMs
- Provides auto-refresh cycle to DDR SDRAM
- Supports single data beat or cacheline data transfers (bursts of 2, 4, or 8 words)
- Supports *Target Word First* read bursts for optimized CPU operation



The current implementation can be tuned for synthesis within the Virtex-II fabric to operate at a 100Mhz. Additional functionality can also be added to more fully exploit the PLB architecture and the ZBT capabilities.

## Module Port Interface

The PLB interface signals are listed in [Table 18-1](#).

**Table 18-1: PLB Interface Pin Description**

| Name              | Direction | Description                                                        |
|-------------------|-----------|--------------------------------------------------------------------|
| SYS_plbClk        | Input     | System Clock                                                       |
| SYS_plbReset      | Input     | System Reset                                                       |
| PLB_abort         | Input     | Abort transaction prior to Address Ack                             |
| PLB_ABus[0:31]    | Input     | PLB Address Bus                                                    |
| PLB_BE[0:7]       | Input     | Byte Enable                                                        |
| PLB_busLock       | Input     | Used to lock the bus                                               |
| PLB_compress      | Input     | Indicates compressed data transfer                                 |
| PLB_guarded       | Input     | Indicates a guarded transaction                                    |
| PLB_lockErr       | Input     | PLB lock error indicator                                           |
| PLB_masterID[0:3] | Input     | PLB current master identifier                                      |
| PLB_Msize[0:1]    | Input     | Size of master (bit-width)                                         |
| PLB_ordered       | Input     | Prevents other writes interrupting                                 |
| PLB_PAVali        | Input     | Indicates a primary address valid. This initiates a transaction    |
| PLB_RNW           | Input     | Indicates read (1) or write (0)                                    |
| PLB_size[0:3]     | Input     | Encoded number of words to transfer                                |
| PLB_type[0:2]     | Input     | Different transfers types (e.g. DMA)                               |
| Sl_addrAck        | Output    | Slave acknowledges the address                                     |
| Sl_MBusy[0:15]    | Output    | Slave decoding of Master ID, indicates master is Busy - one hot    |
| Sl_MErr[0:15]     | Output    | Slave error with this master - one hot                             |
| Sl_rearbitrate    | Output    | Restart the transaction                                            |
| Sl_SSize[0:1]     | Output    | Slave Size - indicates width                                       |
| Sl_wait           | Output    | The slave will respond, but not soon                               |
| PLB_rdPrim        | Input     | Secondary Address now primary read                                 |
| PLB_SAVali        | Input     | Secondary Address Valid, allows for Pipe-lining and pre-processing |
| PLB_wrPrim        | Input     | Secondary Address now primary write                                |
| PLB_wrBurst       | Input     | Write burst unit size (e.g. byte)                                  |
| PLB_wrDBus[0:63]  | Input     | Write Data Bus                                                     |

Table 18-1: PLB Interface Pin Description (Continued)

| Name             | Direction | Description                                               |
|------------------|-----------|-----------------------------------------------------------|
| Sl_wrBTerm       | Output    | Master must terminate current burst                       |
| Sl_wrComp        | Output    | Write transfer Complete                                   |
| Sl_wrDAck        | Output    | Write Data Acknowledge                                    |
| PLB_rdBurst      | Input     | Read burst unit size (e.g. byte)                          |
| Sl_rdBTerm       | Output    | Master must terminate current burst                       |
| Sl_rdComp        | Output    | Read transfer Complete                                    |
| Sl_rxDAck        | Output    | Read Data Acknowledge                                     |
| Sl_rxDBus[0:63]  | Output    | Read Data Bus                                             |
| Sl_rdWdAddr[0:3] | Output    | Current address being read for Target Word First transfer |

Table 18-2: DDR SDRAM Interface Pin Description

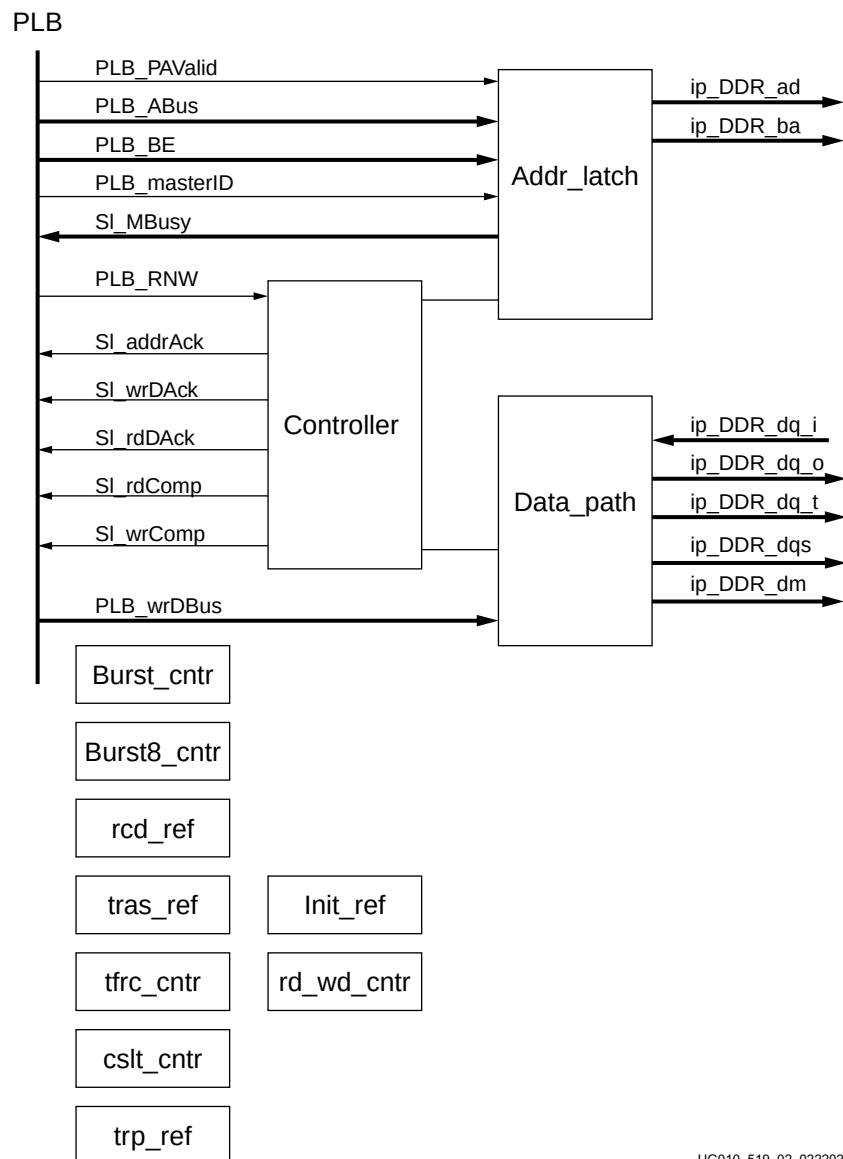
| Name              | Direction | Description                          |
|-------------------|-----------|--------------------------------------|
| ip_DDR_ad[11:0]   | Output    | Address                              |
| ip_DDR_dq_i[31:0] | Input     | Double data rate data input          |
| ip_DDR_dq_o[31:0] | Output    | Double data rate data output         |
| ip_DDR_dq_t[31:0] | Output    | Double data rate data 3-state enable |
| ip_DDR_dqs[3:0]   | Output    | Data strobe                          |
| ip_DDR_rasb       | Output    | Command                              |
| ip_DDR_casb       | Output    | Command                              |
| ip_DDR_web        | Output    | Command                              |
| ip_DDR_ba[1:0]    | Output    | Bank address                         |
| ip_DDR_csb        | Output    | Chip select (active low)             |
| ip_DDR_cke        | Output    | Clock enable                         |
| ip_DDR_dm[3:0]    | Output    | Data mask                            |

Table 18-3: Other System Signals Pin Description

| Name      | Direction | Description |
|-----------|-----------|-------------|
| SYS_Clk2x | Input     | 2x clock    |

## DDR Controller

The DDR SDRAM Controller module receives instructions from the PLB to read and write to external DDR SDRAMs. Figure 18-2 shows the block diagram for the DDR controller.



UG010\_519\_02\_022202

Figure 18-2: DDR Controller Block Diagram

The DDR SDRAM Controller module receives PLB transfer requests and generates the control signals for the DDR SDRAM as well as for other modules. The Addr\_latch module latches the PLB address to generate row, column, and bank addresses for the DDR SDRAM. The Data\_path module latches the 64-bit data from the PLB and multiplexes it into 32-bit double data rate data for the DDR SDRAM. The Init\_ref module handles the DDR SDRAM initialization sequence and periodic refresh.

## Initialization and Refresh

Upon power up, the controller initializes the DDR SDRAMs with the following sequence:

**precharge-all, EMRS, MRS, precharge\_all, refresh, refresh**

The DDR SDRAMs are programmed during MRS with these values:

**Burst length = 8**  
**Burst type = interleaved**  
**CAS latency = 2**  
**Operating mode = normal**

### Burst Length

The controller supports the following PLB transfer sizes: 1-8 bytes non-burst, 4-word cacheline, and 8-word cacheline. The DDR SDRAMs are programmed with burst length of 8. For read transactions, the controller always does a full read of 8 words and **SL\_rdBck** selects the appropriate words to transfer back to the PLB.

For write transfers, the controller always generates **ip\_DDR\_dqs** strobes for a full burst of 8, and uses **ip\_DDR\_dm** signals to mask out all unnecessary words. **ip\_DDR\_dm** signals are also used to further select different bytes within the words to be written.

### Burst Type

In order to support Target Word First during read transactions, the DDR SDRAMs are programmed with Interleaved Burst type. The **rd\_wd\_cntr** module generates the appropriate **SL\_rdWdAddr** for the PLB.

### CAS Latency

The CAS latency is set to 2, making data available from the DDR SDRAMs two clock cycles after a read command.

### Refresh

The DDR SDRAMs requires AUTO REFRESH cycles at an average interval of 15.625  $\mu$ s (maximum). The controller has a 9-bit counter to generate the AUTO REFRESH command every 512 clock cycles.

## Address Mux

The **addr\_latch** module gets its control signals from the controller and generates row, column, and bank addresses for the DDR SDRAMs. The DDR addresses (along with control signals) are generated on the negative edge of **clk** to guarantee the hold time on the DDR SDRAMs.

The mapping between the PLB address and the DDR SDRAM address is shown in **Figure 18-3**.

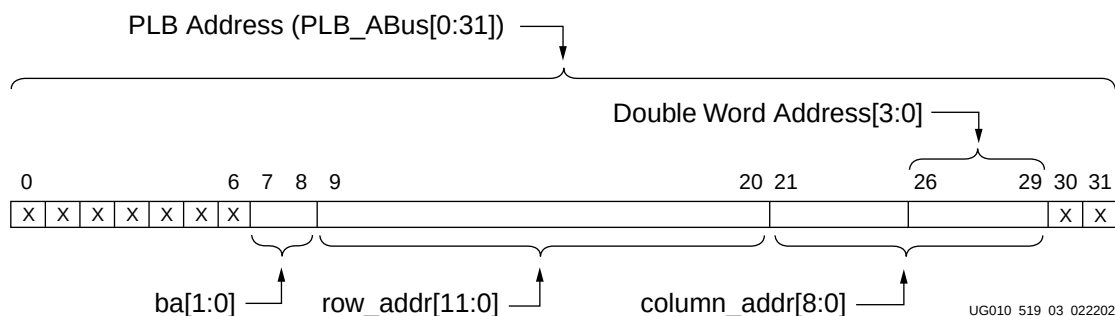


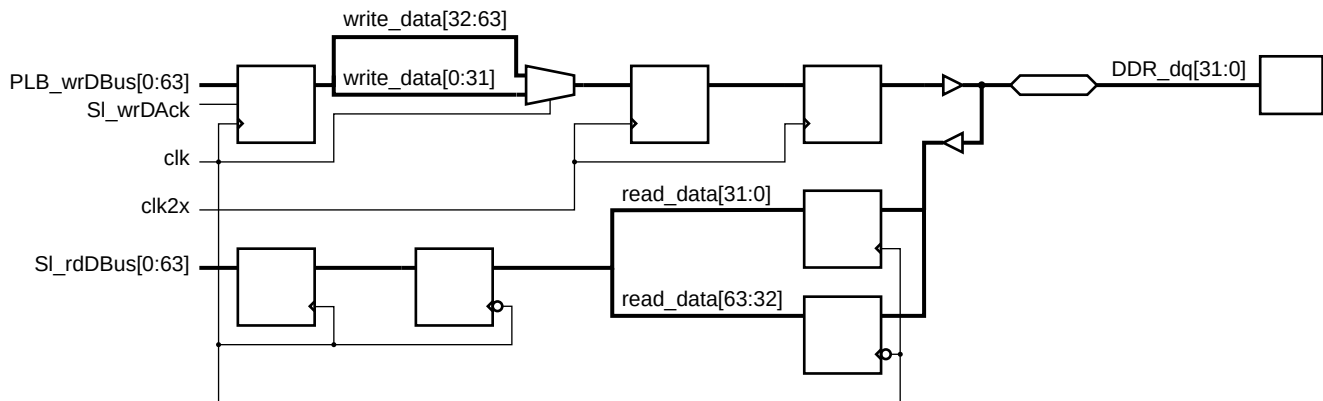
**Figure 18-3: Mapping Between the PLB and the External DDR SDRAM Address**

The upper 7 bits (**PLB\_ABus[0:6]**), configurable based on the **C\_LSB\_ADDR** parameter, are used to determine whether the DDR SDRAM is being addressed by a transaction.

Since the DDR SDRAM controller handles data in 32-bit chunks (4 bytes), and the PLB is based on byte addresses, the DDR ignores the last two bits. The DDR column address maps to **PLB\_ABus[21:29]**, the row address gets **PLB\_ABus[8:20]**, and the bank address gets **PLB\_ABus[1:0]**.

## Data Path

Figure 18-4 shows the basic data flow for read and write.



UG010\_519\_04\_061201

Figure 18-4: Read and Write Data Flow

For a write cycle, the 64-bit **PLB\_wrDBus** is multiplexed into a 32-bit data bus with **clk2x**, using **clk** to select between the upper and lower words. The data is then registered in the IOB flip-flop to minimize clk-to-out delay.

The **ip\_DDR\_dqs** signal is generated by a toggle flip-flop which is clocked with the negative edge of **clk2x**. This ensures the **ip\_DDR\_dqs** edges are at the center of **ip\_DDR\_dq**.

For a read cycle, the 64-bit **ip\_DDR\_dq** is clocked with **clk2x**. It is then assembled into 64-bit data with the upper bits registered by the positive edge of **clk**, and the lower bits registered with the negative edge of the clock.

Since the DDR SDRAM may have negative clock-to-out, the read data is delayed by being registered in the CLB instead of the IOB flip-flops.

## Controller State Machine

An overview of the state machine is shown in [Figure 18-5](#). The dashed lines indicate an automatic sequence.

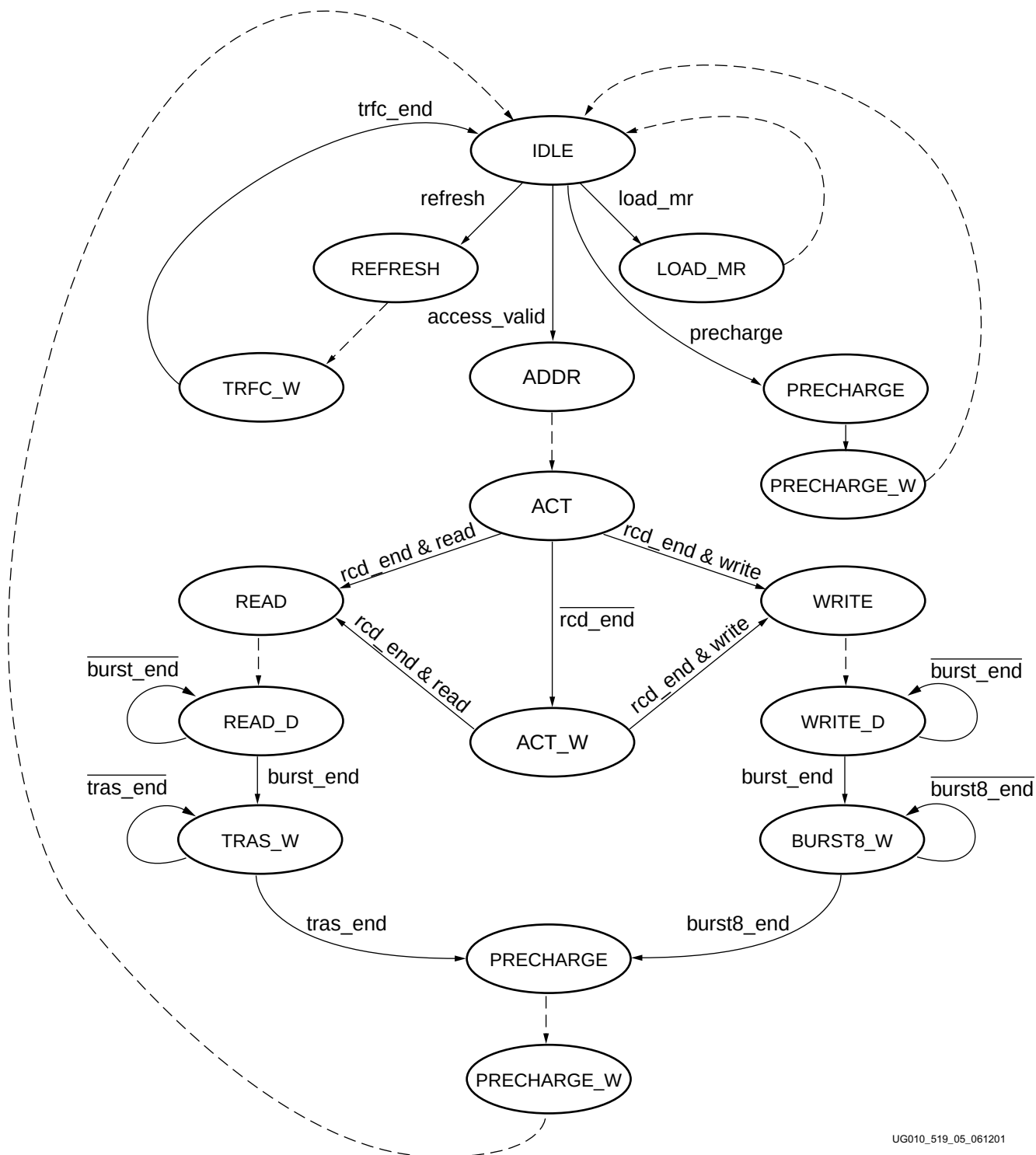


**Figure 18-5: Controller State Machine Overview**

The controller initializes in the IDLE state. The next state of the controller could be PRECHARGE, LOAD\_MR, REFRESH, or ADDR, depending on the required command.

For read and write, the controller decodes the PLB address to find transaction intended for the DDR SDRAM. If the access is valid (correct address, type, and burst sizes), then the controller goes into the ADDR and then the ACT state. During the ACT state, it generates an ACTIVE command and row address for the DDR SDRAM.

During a write transaction, the controller goes from ACT to WRITE (when **Trcd** is satisfied), then to the WRITE\_D state. The controller stays in WRITE\_D until **burst\_end** is true. It then goes into the BURST8\_W state to wait for the **burst8\_end** signal. For write transactions **ip\_DDR\_dqs** must be generated for a full burst of 8. **ip\_DDR\_dm** is low only during WRITE\_D, and is high during BURST8\_W (masking all unnecessary data from being written into SDRAM). The controller then goes into PRECHARGE before returning to the IDLE state. In this design, the RAS-to-CAS delay is three clock cycles. If a different RAS-to-CAS delay is needed, then adjust the **rcd\_cntr** and the data pipeline registers accordingly.

During a read transaction, the controller goes from ACT to READ (when **Trcd** is satisfied), then to READ\_D. The controller stays in READ\_D until **burst\_end** is true. The DDR SDRAM always does a burst of 8, but the controller will ignore the unnecessary data. To support Target Word First read bursts, the DDR SDRAM is programmed with Interleaved mode. The **rd\_wd\_cntr** will generate the correct word address for data coming from the DDR SDRAM.

The controller generates all control signals for the DDR SDRAM (**rasb**, **casb**, **web**), all the handshake signals for the PLB (**addrAck**, **wrDAck**, **rdDAck**, **wrComp**, **rdComp**), as well as all control signals for **data\_path**, **addr\_latch** and various counters in the design.

## Timing Diagrams

### Write Cycle

Figure 18-6 shows the timing diagram for writing a burst of four data words to the DDR SDRAM.

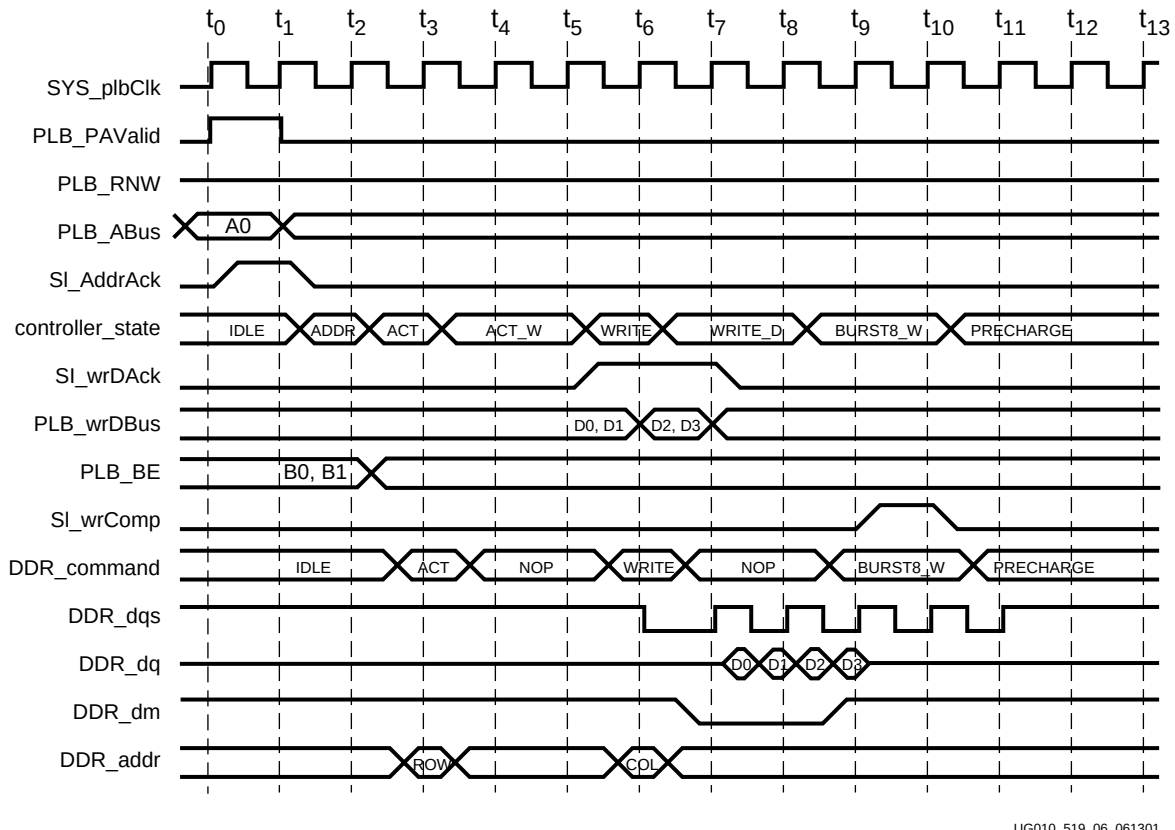


Figure 18-6: Timing Diagram for Cacheline Write of Two Double Words

- At  $t_0$ , the controller sees **PLB\_PAVali**, valid address, and transfer type.
- At  $t_1$ , it goes from the IDLE to ADDR state, generating **SI\_AddrAck**.
- At  $t_2$ , it goes from the ADDR to the ACT state. The controller generates the ACT command and row address for the DDR SDRAM on the negative **clk** edge after  $t_2$ .
- At  $t_3$ , the controller goes to ACT\_W and waits for **Trcd** (ACTIVE to READ/WRITE delay).
- At  $t_5$ , the controller goes to the WRITE state, issuing the WRITE command and column address to the DDR SDRAM.
- At  $t_6$ , it goes to the WRITE\_D state to write the correct number of data words for the current burst type. If the current burst type is less than 8, the controller waits in the BURST8\_W state before advancing to PRECHARGE. The signals **ip\_DDR\_dq** and **ip\_DDR\_dqs** are issued at a double data rate. Note that **ip\_DDR\_dqs** is always generated for a full burst of 8, but **ip\_DDR\_dm** is masking all the unnecessary words.
- At  $t_9$ , after all the data has been written, **SI\_wrComp** is issued.



## Read Cycle

Figure 18-7 shows the timing diagram for reading a burst of eight data words from the DDR SDRAM, target word first.

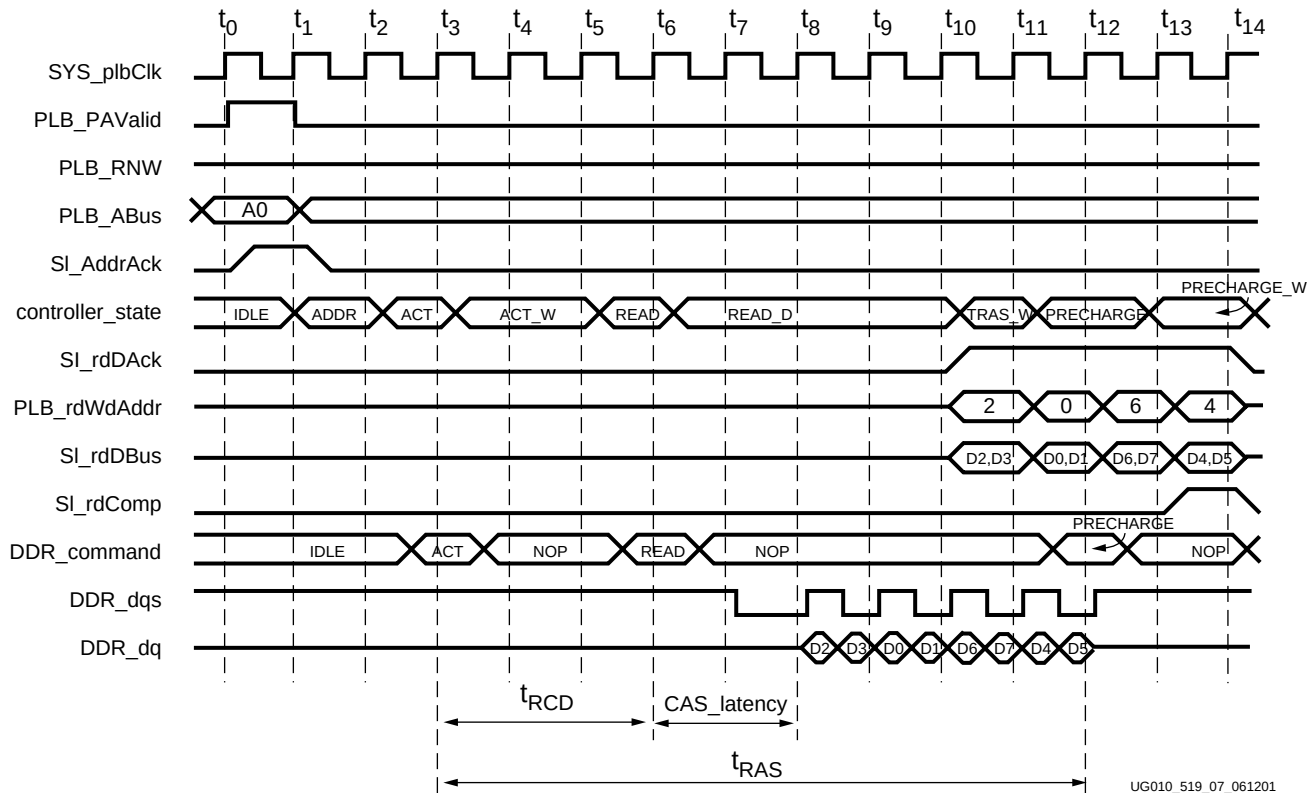


Figure 18-7: Timing Diagram for Cacheline Read of Two Double Words

The controller goes from IDLE to ADDR, ACT, ACT\_W, READ, and READ\_W states in a similar manner as in the write cycle.

1. At t8, two clock cycles after the READ command is issued, the DDR SDRAM starts to send **ip\_DDR\_dq** and **ip\_DDR\_dqs** at double data rate with the targeted word first. The controller receives the 32-bit data, and assembles the data into 64-bit double words before sending it to the PLB.
2. At t10, the controller sends **SI\_rdDack**, **SI\_rdWdAddr**, and **SI\_rdBUS** to the PLB.
3. At t13, **SI\_rdComp** is issued as the last data is sent.

## Conclusion

The DDR SDRAM Controller module reference design demonstrates the interfaces to the Processor Local Bus. It provides access to external DDR SDRAMs for systems implemented with a PowerPC core in the Virtex-E fabric. The source Verilog code is provided, and can be easily modified to meet the requirements of a user's design.

# PLB to OPB Bridge-Out Module

---

## Overview

The PLB to OPB Bridge-Out module translates PLB transactions into OPB transactions. It functions as a slave on the PLB side and a master on the OPB side. It also contains a DCR slave interface to provide access to its bus error status registers. The Bridge-Out is necessary in systems where a PLB master device, such as a CPU, requires access to OPB peripherals.

## Related Documents

- IBM CoreConnect™ 64-Bit Processor Local Bus: Architecture Specification
- IBM CoreConnect 64-Bit On-Chip Peripheral Bus: Architecture Specifications
- IBM CoreConnect 32-Bit Device Control Register Bus: Architecture Specifications
- Virtex-II Pro™ Platform FPGAs (Advance Product Specification)

## Features

- 64-bit PLB Slave interface
  - Communicates with 32- or 64-bit PLB masters
  - Non-burst transfers of 1-8 bytes
  - Burst transfers, including word and doubleword bursts of fixed or variable lengths, up to 16 words worth of data.
  - Limited support for byte, halfword, quad-word and octal-word bursts to maintain PLB compliance
  - Cacheline transactions of 4, 8, and 16 words
  - Supports up to 16 PLB masters
  - Programmable lower and upper address boundaries
- 32-bit OPB Master interface with byte enable transfers

**Note:** Does not support dynamic bus sizing without additional glue logic
- PLB and OPB clocks can have an asynchronous relationship (as long as the OPB clock frequency is less than or equal to the PLB clock frequency)
- Bus Error Address Registers (BEAR) and Bus Error Status Registers (BESR) to report errors
  - DCR Slave interface provides access to BEAR/BESR
- 16-word posted write buffer

## Module Port Interface

### PLB Slave Interface Signals

Table 19-1: Global Signals

| Name         | Direction | Description      |
|--------------|-----------|------------------|
| SYS_plbClk   | Input     | PLB system clock |
| SYS_plbReset | Input     | PLB system reset |

Table 19-2: Transfer Qualifier Signals

| Name              | Direction | Description                            |
|-------------------|-----------|----------------------------------------|
| PLB_PAVValid      | Input     | PLB primary address valid indicator    |
| PLB_busLock       | Input     | PLB bus lock                           |
| PLB_masterID[0:3] | Input     | PLB current master identifier          |
| PLB_RNW           | Input     | PLB read not write                     |
| PLB_BE[0:7]       | Input     | PLB byte enables                       |
| PLB_size[0:3]     | Input     | PLB transfer size                      |
| PLB_type[0:2]     | Input     | PLB transfer type                      |
| PLB_MSize[0:1]    | Input     | PLB master data bus size               |
| PLB_compress      | Input     | PLB compressed data transfer indicator |
| PLB_guarded       | Input     | PLB guarded transfer indicator         |
| PLB_ordered       | Input     | PLB synchronize transfer indicator     |
| PLB_lockErr       | Input     | PLB lock error indicator               |
| PLB_abort         | Input     | PLB abort bus request indicator        |
| PLB_ABus[0:31]    | Input     | PLB address bus                        |
| Sl_addrAck        | Output    | Slave address acknowledge              |
| Sl_wait           | Output    | Slave wait indicator                   |
| Sl_SSize[0:1]     | Output    | Slave data bus size                    |
| Sl_rearbitrate    | Output    | Slave rearbitrate bus indicator        |
| Sl_MBusy[0:15]    | Output    | Slave busy indicator                   |
| Sl_MErr[0:15]     | Output    | Slave error indicator                  |

Table 19-3: Address Pipelining Signals

| Name         | Direction | Description                                      |
|--------------|-----------|--------------------------------------------------|
| PLB_SAVValid | Input     | PLB secondary address valid indicator            |
| PLB_rdPrim   | Input     | PLB secondary to primary read request indicator  |
| PLB_wrPrim   | Input     | PLB secondary to primary write request indicator |

Table 19-4: Write Data Bus Signals

| Name             | Direction | Description                             |
|------------------|-----------|-----------------------------------------|
| PLB_wrDBus[0:63] | Input     | PLB write data bus                      |
| PLB_wrBurst      | Input     | PLB burst write transfer indicator      |
| Sl_wrDAck        | Output    | Slave read data acknowledge             |
| Sl_wrComp        | Output    | Slave write transfer complete indicator |
| Sl_wrBTerm       | Output    | Slave terminate write burst transfer    |

Table 19-5: Read Data Bus Signals

| Name             | Direction | Description                            |
|------------------|-----------|----------------------------------------|
| PLB_rdBurst      | Input     | PLB burst read transfer indicator      |
| Sl_rDBus[0:63]   | Output    | Slave read data bus                    |
| Sl_rdWdAddr[0:3] | Output    | Slave read word address                |
| Sl_rDAck         | Output    | Slave read data acknowledge            |
| Sl_rdComp        | Output    | Slave read transfer complete indicator |
| Sl_rdBTerm       | Output    | Slave terminate read burst transfer    |

## OPB Master Interface Signals

Table 19-6: OPB Master Interface Signals

| Name           | Direction | Description              |
|----------------|-----------|--------------------------|
| OPB_Clk        | Input     | OPB clock                |
| OPB_DBus[0:31] | Input     | OPB data bus             |
| OPB_errAck     | Input     | OPB error acknowledge    |
| OPB_MnGrant    | Input     | OPB master bus grant     |
| OPB_retry      | Input     | OPB retry                |
| OPB_timeout    | Input     | OPB timeout error        |
| OPB_xferAck    | Input     | OPB transfer acknowledge |
| M_Abus[0:31]   | Output    | Master address bus       |

Table 19-6: OPB Master Interface Signals (Continued)

| Name         | Direction | Description                 |
|--------------|-----------|-----------------------------|
| M_BE[0:3]    | Output    | Master byte enables         |
| M_busLock    | Output    | Master bus arbitration lock |
| M_DBus[0:31] | Output    | Master data bus             |
| M_DBusEn     | Output    | Master data bus enable      |
| M_request    | Output    | Master bus request          |
| M_RNW        | Output    | Master read not write       |
| M_select     | Output    | Master select               |
| M_seqAddr    | Output    | Master sequential address   |

## Bus Error Address/Status Registers

Table 19-7: DCR Slave Interface Signals

| Name              | Direction | Description      |
|-------------------|-----------|------------------|
| DCR_Abus[0:9]     | Input     | DCR address bus  |
| DCR_DBusIn[0:31]  | Input     | DCR data bus in  |
| DCR_Read          | Input     | DCR read         |
| DCR_Write         | Input     | DCR write        |
| DCR_Ack           | Output    | DCR acknowledge  |
| DCR_DBusOut[0:31] | Output    | DCR data bus out |

Table 19-8: Error Status Signal

| Name          | Direction | Description        |
|---------------|-----------|--------------------|
| Bus_Error_Det | Output    | Bus Error Detected |

## Parameters

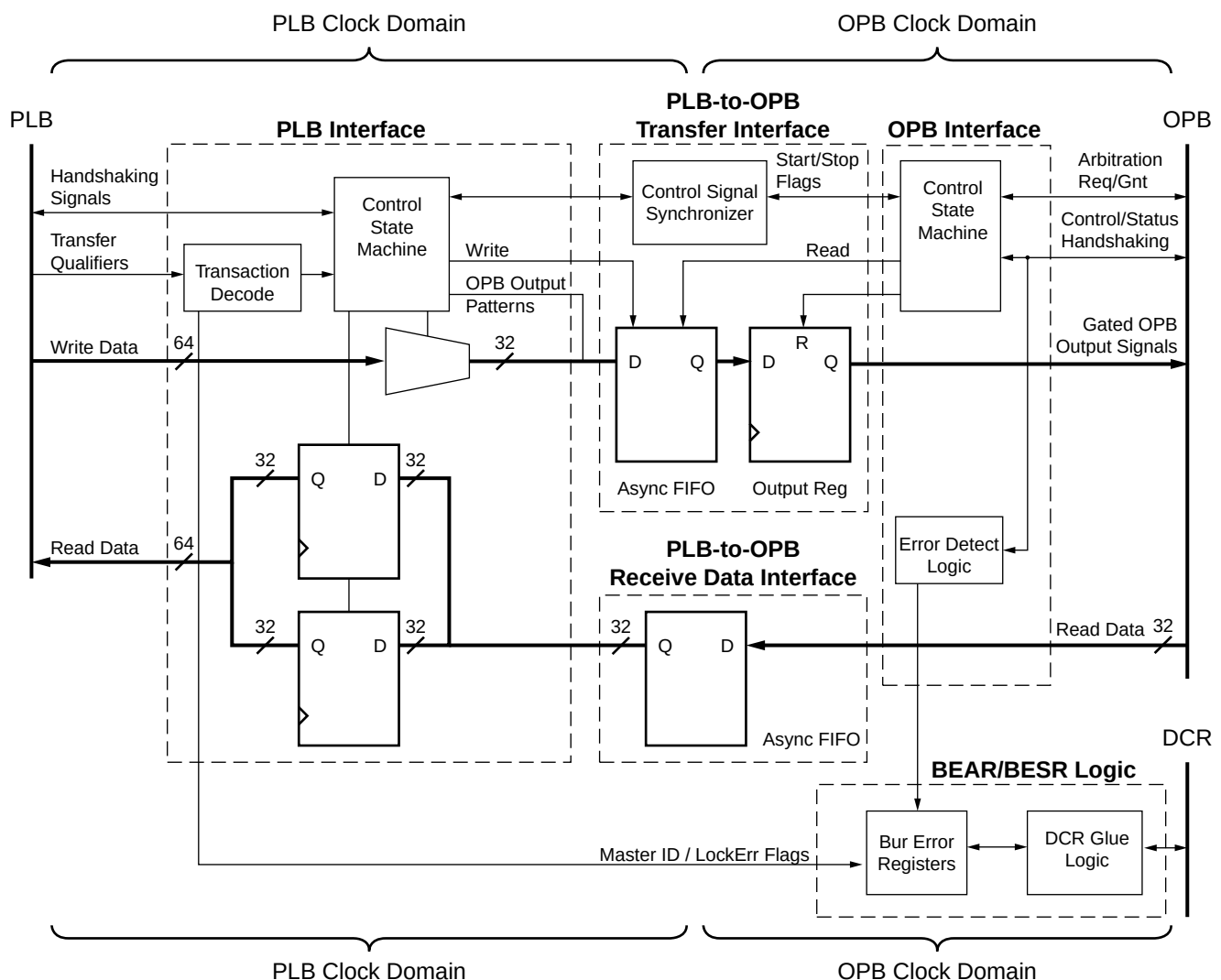
Table 19-9: Parameters

| Name            | Description                                                                                                                                                                  |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| C_LSB_ADDR_LO   | Least significant bit for address comparator to look at when checking lower address boundary (Note: must be aligned on 64-byte boundary; i.e. less than or equal to 25)      |
| C_BASE_ADDR_LO  | 32-bit lower address boundary                                                                                                                                                |
| C_LSB_ADDR_HI   | Least significant bit for address comparator to look at when checking upper address boundary ( <b>Note:</b> must be aligned on 1 KB boundary; i.e. less than or equal to 21) |
| C_BASE_ADDR_HI  | 32-bit upper address boundary                                                                                                                                                |
| C_DCR_BASE_ADDR | 10-bit base address of DCR registers                                                                                                                                         |
| C_CLK_ASYNC     | Specifies if OPB and PLB clocks are synchronous (= 0) or asynchronous (= 1) to each other                                                                                    |

# Implementation

## High-Level Description

Figure 19-1 provides a high-level overview of the design of the PLB to OPB Bridge. PLB transactions are received and decoded in the PLB interface logic. The PLB interface logic then generates the necessary sequence of OPB signals (including address, byte enables, and data) to perform the transaction. These OPB signal patterns are then loaded into the PLB-to-OPB transfer interface logic which acts like a FIFO to decouple the PLB interface from the OPB master. The OPB interface logic receives the data patterns and outputs them on OPB. The OPB logic also handles the OPB handshaking signals such as request/grant, select, retry, timeout, and transfer acknowledge. For PLB read transactions, the OPB master returns the read data to the PLB logic via the OPB-to-PLB receive data interface. The receive data interface also acts like a FIFO to decouple the data return path and status flags from the OPB to the PLB logic.



UG010\_521\_01\_062001

Figure 19-1: High-Level Overview of the PLB to OPB Bridge

## PLB Interface

The PLB interface is designed with a pipelined architecture to improve timing and to support high clock frequencies. Output signals to the PLB are designed to be driven out from flip-flops through a minimal amount of logic. PLB input signals are also limited to a few levels of logic before being registered. Pipelining introduces some additional latency in the design since some signals are delayed through registers. However, the use of pipelining balances low transaction latency with high clock frequency timing.

### Address Decode Cycle

PLB transactions begin with an address decode cycle. An extra clock cycle is used to decode the address and transfer qualifiers of a PLB request before the address can be acknowledged. This address decode cycle can be performed during a secondary address request (**PLB\_SAVaLid** = 1) to help overlap the address decode cycle with a previous transaction. Though address decoding can be performed during secondary address cycles, address acknowledge (**SL\_addrAck**) is only returned during a primary address cycle (**PLB\_PAVaLid** = 1). The user can set address decoding parameters to configure the upper and lower address boundary. The upper address boundary must be aligned to 1 K byte. The lower address boundary must be aligned to at least a 64-byte boundary. Note that the address comparators should be set so as few upper address bits as possible need to be compared. Reducing the size of the address comparators improves performance and reduces logic utilization.

### Address Sequencing

The PLB transfer is then sequenced into sets of OPB transactions. Each set of data patterns consists of a 32-bit address, 4-bit byte enables, 32-bit write data, read/write flag, bus lock flag, sequential address flag, and end-of-transfer flag. A state machine (based on a down counter) is loaded with the length of the data transfer and then decremented until it reaches zero. This controls the generation of OPB signals. For each non-zero counter value, an OPB address counter is incremented, while additional logic sets the necessary OPB transfer qualifiers. The OPB transaction patterns are then loaded in a FIFO memory for processing by the OPB logic.

### Write Transactions

For write transactions, the write data acknowledges (**SL\_wrDack**) are returned over PLB and the write data (**SL\_wrDBus**) is registered and sent on to the FIFO memory. The bridge can buffer up to 16 words of write data into the FIFO. Both 32- and 64-bit PLB transactions are supported while OPB transactions are fixed at 32 bits wide. Therefore, the PLB interface logic is aware of the master size (**PLB\_Msize**) and multiplexes data down to a 32-bit data path as necessary. After all the write data has been transferred over, the write complete (**SL\_wrComp**) signal can be asserted. Though the transaction is complete over PLB, the bridge will wait for the OPB logic to report that the transaction was completed over the OPB before it can begin processing another transaction.

### Read Transactions

For read transactions, the OPB write data signals are set to zero while the other OPB address and transfer qualifiers are generated (in the same manner as with write transactions). Read data acknowledges (**SL\_rdAck**) and read data (**SL\_rdDBus**) are returned over the PLB as the read data is received from the OPB interface. The PLB read complete signal (**SL\_rdComp**) is asserted when all the read data has been returned or when the master terminates a burst cycle. Read requests are generated for up to 16 words during read transactions. For variable length bursts where the burst length is not specified, the bridge will prefetch up to 16 words of data to satisfy the read request. For fixed-length bursts, the bridge will fetch only the words requested, up to 16 words. Cacheline read data transfers are only sequenced in linear order instead of *target word first* order.

## PLB-to-OPB Transfer Interface

This group of logic implements an asynchronous FIFO interface between the PLB and OPB interfaces (`C_CLK_ASYNC=1`). It also synchronizes and passes handshaking flags between the two sides to communicate that the FIFO is not empty or to signal that a PLB read burst has been terminated by the PLB master. A BRAM is used as the memory element to store the data, since it is a dual-port memory that can support independent clocks on each side. The BRAM can be replaced by a number of LUT-based dual-port memories to yield higher clock frequencies at the expense of greater LUT utilization. In addition, a simplified, synchronous version of this module can be used (`C_CLK_ASYNC=0`) to reduce logic utilization and increase speed. The synchronous version requires that the OPB clock be divided down from the PLB clock by an integer value and that the rising edges be phase aligned.

## OPB Interface

The OPB interface is designed with a pipelined architecture to improve timing and support high clock frequencies. Output signals to OPB are designed to be driven out from flip-flops through a minimal amount of logic. OPB input signals are also limited to a few levels of logic before being registered. To further improve timing, output signals to the OPB are internally gated off.

The OPB interface begins an OPB transaction when it receives a start flag indicating that the FIFO is not empty. It requests (**M\_request**) access to the OPB bus and begins the transaction when a grant (**OPB\_MnGrant**) is received. The OPB transaction is performed by simply driving the contents of the FIFO out onto OPB and advancing the FIFO read pointer after every transfer acknowledge (**OPB\_xferAck**). After all the data in the FIFO has been transmitted (indicated by the end-of-transfer flag being high), the OPB logic goes back to its IDLE state and waits for the FIFO to leave the empty state before beginning a new transaction. During read transactions, data is returned to the PLB logic by way of the OPB to PLB receive data interface.

All data transfers greater than one word are performed as locked transfers over OPB (**M\_busLock** = 1). To improve bus efficiency, the last word in a set of OPB transfers is unlocked as recommended by the OPB specification. If the PLB bus is locked for a transaction and remains locked (**PLB\_busLock** asserted) after the bridge completes all its OPB data transfers, the OPB master will hold **M\_busLock** until the PLB is unlocked. This allows a series of *locked PLB transfers* to be carried out as a series of *locked OPB transfers*.

All bursts transfers greater than one word and all cacheline transfers are performed with the sequential address flag asserted (**M\_seqAddr** = 1). The sequential address flag helps improve the performance of OPB slave peripherals. All sequential bursts are also performed with bus lock asserted (as required by the OPB specification). Note that non-burst, 5 to 8 byte transfers from 64-bit PLB masters are performed as two OPB transfers, but the sequential address flag is not asserted.

If an OPB timeout occurs while the OPB master is driving the bus, it backs off the bus for one cycle as required by the OPB specification, but maintains **M\_busLock**. This allows it to advanced the FIFO read pointer and continue the transaction. A timeout condition causes an error flag to be passed over to the PLB logic (to drive **M\_Err**) and to the BEAR/BESR logic to update the error registers.

Slaves that generate an **OPB\_retry** cause the OPB master to back off the bus for one cycle before attempting the transaction over again by requesting access to the bus.

OPB error acknowledge signals (**OPB\_errAck**) generate error flags similar to timeout error flags that are sent to the PLB and BEAR/BESR logic. Error acknowledge signals are otherwise treated like transfer acknowledge signals in that the OPB master continues the data transfer with the next data.



## OPB-to-PLB Receive Data Interface

This module acts like an asynchronous FIFO for data going back from OPB to PLB. It communicates read data and status flags back to OPB and synchronizes signals passing between the two clock domains. The asynchronous design of this module adds additional latency to transactions so the user may use a simplified synchronous version of this design to reduce logic utilization and lower the latency. The synchronous version requires that the OPB clock be divided down from the PLB clock by an integer value and that the rising edges be phase aligned.

## BEAR/BESR Logic

Error flags from the OPB interface as well as the **PLB\_masterID** and **PLB\_lockErr** qualifiers are forwarded to the BEAR/BESR logic to record errors. The error registers are accessible over the DCR bus. They record information such as the type of error (error acknowledge versus timeout), whether it was a read or write, and which PLB master caused the error. The address and byte enable patterns corresponding to the error are also recorded. If the **PLB\_lockErr** signal was asserted, the corresponding PLB master's BEAR/BESR is locked so that subsequent errors do not overwrite it, unless the error register is first cleared. If **PLB\_lockErr** is not asserted, then the error information can be overwritten by a subsequent error.

The DCR bus provides access to the BEAR/BESR and operates on the same clock as the OPB logic. This requires that the DCR master's clock be synchronous to the OPB clock in order for the DCR slave interface to work properly.

Whenever an error has been detected and has not been cleared, the output signal **Bus\_Error\_Det** is asserted. This signal can be used for diagnostic purposes, such as turning on an LED, or to alert some other module that an error occurred in the bridge.

## PLB and OPB Transaction Behavior

This section describes important information about the behavior of the PLB bridge with respect to the PLB and OPB protocols.

### PLB Transaction Qualifiers and Arbitration Signals

- Directly supports 32-/64-bit PLB masters
- Bridge responds to 128- or 256-bit PLB masters, but only as a 64-bit device
  - Burst transfers of quad words and octal words may terminate early
- Error and busy flags support up to 16 PLB masters
- Accepts only memory transfers (**PLB\_type** = 000)
  - Other transfer types are ignored
- Ignores **PLB\_ordered** qualifier since all transactions are completed in order by design
- Asserts **SL\_rearbitrate** if the bridge is busy with a transaction and a new transaction for the bridge is requested by the PLB arbiter
- Drives **SL\_wait** to zero since it is not utilized
- Introduces an extra clock cycle of latency for address decoding
  - However, the address decode cycle can be performed during a secondary address cycle (**PLB\_SAVAlid** = 1)
- Ignores secondary address cycles
  - Bridge does not support address pipelining with respect to data transfers
- Continuously asserts OPB bus lock (**M\_busLock**) for as long as the PLB bus remains

locked, providing PLB bus lock (**PLB\_busLock**) is held after the bridge asserts address acknowledge. Upon receiving a subsequent address acknowledge from the bridge, **M\_busLock** remains asserted until the data transfer is complete. If a slave asserts a retry, **M\_busLock** is deasserted, as required by the OPB specification.

## PLB Data Transfers

- Supports non-burst/non-cacheline transfers of 1-8 bytes
- Supports cacheline transfers of 4, 8, or 16 words in linear order  
Target word first order is not supported.
- Accepts all burst transfers of bytes (**PLB\_size** = 1000) but terminates immediately with **SL\_wrBTerm** or **SL\_rdBTerm** after one data transfer
- Accepts all burst transfers of half words (**PLB\_size** = 1001) but terminates immediately with **SL\_wrBTerm** or **SL\_rdBTerm** after one data transfer
- Supports fixed-length burst transfers (**PLB\_BE** > 0000) of words (**PLB\_size** = 1010) up to 16 words.
- Supports fixed-length burst transfers of double words (**PLB\_size** = 1011) up to 8 double words

For doubleword bursts greater than 8 double words, the bridge will terminate the burst with **SL\_wrBTerm** or **SL\_rdBTerm** at the eighth data transfer

- Burst transfers of quad words and octal words (**PLB\_size** = 1100 or 1101) will be treated like double word transfers causing **SL\_wrBTerm** or **SL\_rdBTerm** to be asserted before all data has been transferred. For example, a burst of 3 octal words will be terminated after 3 double words have been transferred
- Variable length write bursts (**PLB\_BE** = 0000) will cause up to 16 words of write data to be queued up. After 16 words of write data have been transferred, the bridge will assert **SL\_wrBTerm** or **SL\_rdBTerm** to prevent the master from continuing the variable length burst past 16 words. Note that 16 words equal 8 double words so variable length doubleword bursts are terminated at the eighth double word
- Variable length read bursts will cause the bridge to prefetch 16 words of data from the OPB. If the master terminates the burst with less than 16 words transferred, the OPB master will terminate any reads that have not been performed. (Any OPB read transaction in progress is allowed to complete). Note that the pipelined architecture of the bridge and the use of prefetching implies that the OPB master reads data in advance of knowing when the PLB master stops the burst. This may cause unnecessary read cycles over OPB. PLB masters should use fixed length bursts so that only the desired data is read.
- Fixed and variable length bursts are terminated at 1 KB address boundaries if the guarded attribute is set (**PLB\_guarded** = 1). Prefetched read data does not cross the guarded 1 K address boundary during a variable length read burst.
- The upper address boundary of the PLB slave interface must be aligned to a 1 K boundary. Bursts that cross the upper 1 K address boundary are treated as if the guard bit is set in order to ensure the burst does not proceed past the address boundary of the bridge.

## OPB Data Transfers

- Master interface only supports 32-bit byte enable slaves
- OPB byte enable transfer (**M\_beXfer**) port is not present because it is implicitly asserted with **M\_select**
- OPB byte enable and full word acknowledge (**OPB\_beAck**/**OPB\_fwAck**) are not present because a slave would implicitly assert these signal with **OPB\_xferAck**.
- Does not perform dynamic bus sizing operations for byte or half word OPB slaves

- OPB halfword transfer/acknowledge (**M\_hwXfer**/**OPB\_hwAck**) and full word transfer/acknowledge (**M\_fwXfer**/**OPB\_fwAck**) signals are not present
- Least significant two address bits are always zero since the byte/half word addressing information is available in the byte enables
- Data is not mirrored over inactive byte lanes. If byte enable (**M\_BE**) bit is zero, corresponding data byte lane has an undefined value
- Master output signals do not need AND-OR gating logic in the OPB bus since they are gated internally to improve bus timing. OR-gates are sufficient for the OPB bus logic connection
- Master abort cycles are not generated
- Interface Logic is available to allow legacy dynamic bus sizing devices to communicate with the bridge. However, OPB peripherals should be designed to directly support byte enable transfers for highest performance and lowest resource utilization.

## Bus Error Address and Status Registers

The PLB to OPB Bridge contains four DCR-accessible registers to provide error address and status information. The DCR base address can be set using parameters.

**Table 19-10: Bus Error Address Registers (BEAR) and Bus Error Status Registers (BESR)**

| Register Name | Address           | Bits    | Description                                                                                                                                                                                                                              |
|---------------|-------------------|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BESR_A        | DCR Base Addr + 0 | [0:15]  | Read: Error detect bit for masters [0:15] respectively.<br>1 = error detected<br>0 = no error detected<br>Write: Clear error bit for masters [0:15] respectively.<br>1 = clear error<br>0 = do not clear error                           |
| BESR_B        | DCR Base Addr + 0 | [16:31] | Read: Latched BEAR bit for masters [0:15] respectively.<br>1 = master is driving BEAR<br>0 = master is not driving BEAR<br>Note: Only one master may drive the BEAR, so no more than one bit can be high at any time<br>Write: No effect |
| BESR_C        | DCR Base Addr + 1 | [0:15]  | Read: Read/write bit for masters [0:15] respectively.<br>1 = error was in response to a read<br>0 = error was in response to a write<br>Write: No effect                                                                                 |
| BESR_D        | DCR Base Addr + 1 | [16:31] | Read: Error type bit for masters [0:15] respectively.<br>1 = error was due to OPB Timeout<br>0 = error was due to OPB Error Acknowledge<br>Write: No effect                                                                              |

Table 19-10: Bus Error Address Registers (BEAR) and Bus Error Status Registers (BESR) (Continued)

| Register Name | Address           | Bits    | Description                                                                                                                                                                                                                                                                           |
|---------------|-------------------|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BEAR_A        | DCR Base Addr + 2 | [0:31]  | Read: 32-bit OPB address where error occurred<br>Write: no effect                                                                                                                                                                                                                     |
| BEAR_B        | DCR Base Addr + 3 | [0:3]   | Read: 4-bit OPB byte enable value when error occurred<br>Write: No effect                                                                                                                                                                                                             |
|               | DCR Base Addr + 3 | [4:15]  | Read: Reserved/unused. Read value is undefined<br>Write: Undefined (avoid writes to this register)                                                                                                                                                                                    |
| BESR_E        | DCR Base Addr + 3 | [16:31] | Read: Lock error bit for masters [0:15] respectively.<br>1 = error fields are locked (subsequent errors cannot overwrite master's error fields until error is cleared)<br>0 = error fields are not locked (subsequent errors can overwrite master's error fields)<br>Write: No effect |



# *Interrupt Controller*

---

## Overview

This document describes the specifications for the Interrupt Controller (INTC) for use with the PowerPC® 405 processor core.

The INTC core is a simple, parameterized interrupt controller that attaches to the Device Control Register (DCR) bus.

## Features

- DCR bus compliant
- Number of interrupt inputs is configureable up to the width of the data bus width
- Interrupt controllers can be easily cascaded to provide additional interrupt inputs
- Master Enable Register (MER) for disabling the interrupt request output
- Each input is configurable for edge or level sensitivity
- Automatic edge synchronization when inputs are configured for edge sensitivity
- Prioritized interrupt sources
- Automatic index vector generation for interrupt service routine table dispatch

## Introduction to Interrupt Controllers

Interrupt controllers are used to expand the number of interrupt inputs a computer system has available to the CPU and, optionally, provide a priority encoding scheme. Modern CPUs typically provide one or more interrupt request input pins that allow external devices to request service by the CPU. Some interrupt controllers allow the user to program the polarity of the interrupt inputs and whether they are level or edge sensitive. Some may allow the priority of an interrupt to be programmed as well. Some popular embedded processors and their associated interrupt controllers/mechanisms are described in the following paragraphs.

### Intel 8051

Many single-chip solutions, such as the 8051, offer an embedded interrupt controller along with the functionality of the CPU and on-chip peripherals. The 8051 microcontroller utilizes a hard vector approach for handling interrupts. There is a unique, hard vector address associated with external interrupt 0, timer 0, external interrupt 1, timer 1, and the serial port. Interrupts can be programmed for high or low priority and there is an interrupt enable bit for each interrupt source, as well as a bit for enabling or disabling all interrupts. The two external interrupt inputs can be programmed to be either edge sensitive (falling edge) or level sensitive (active low).

## Zilog Z80

The Z80 supports both hard vector and auto vector modes for interrupts. The Non-Maskable Interrupt (NMI) input utilizes a hard vector and, as its name implies, cannot be disabled by software. This interrupt is an active low, level sensitive interrupt. The other interrupt input (INT), which is also an active low, level sensitive interrupt, supports three different modes. Mode 0 provides compatibility with the Intel 8080 microprocessor. During an interrupt acknowledge cycle the interrupting device jams a restart instruction onto the data bus, which causes program execution to continue at one of eight hard coded locations. Mode 1 is a hard vector interrupt with a single hard vector, similar to the NMI but at a different location. Mode 2 is a fully auto vectored interrupt mode. In this mode the interrupt controller is actually distributed between the processor and the Z80 family peripherals. During an interrupt acknowledge cycle the interrupting device places the low eight bits of the interrupt service routine address on the data bus. The processor provides the upper eight bits from a dedicated register that is loaded by software. NMI always has a higher priority than INT. Multiple devices can be attached to either interrupt input using a wired-or configuration. Additionally, in Mode 2, devices on the INT input can be daisy-chained to provide additional interrupt priorities. The INT input can be masked by software.

## Motorola 68332

The 68332 has seven active low, level sensitive interrupt request inputs (IRQ1 to IRQ7). These inputs correspond to the seven interrupt request levels of the CPU32 core. Devices (internal or external) request service by activating a particular interrupt level. If that level is not masked then the processor acquires the appropriate interrupt vector number and obtains the service routine address from a 256 location interrupt vector table, indexed by the interrupt vector number. The interrupt vector number is determined on a per device basis, and is either at a fixed location relative to the interrupt request level or is supplied by the interrupting device as part of the interrupt acknowledge cycle. Interrupt request level seven is non-maskable and the other six levels can be masked by software. Interrupt request level one has the lowest priority and interrupt request level seven has the highest priority. All the peripherals on-chip can be programmed to request an interrupt on any of the seven levels. Interrupt request levels one through six behave as level sensitive interrupts in that as long as that level is active interrupt requests will be generated. The level must be maintained by the interrupting device until the interrupt has been acknowledged by the processor. Interrupt request level seven behaves like an edge sensitive interrupt since only one interrupt request is generated each time that level is entered and the level must be exited and re-entered to generate another interrupt.

## MIPS

MIPS CPUs have eight interrupt sources, six of which are for hardware interrupts and the remaining two are for software interrupts. There is no priority, all interrupts are considered equal. Each interrupt can be masked by the software and there is a global interrupt enable. MIPS only supports a hard vector mechanism but the vector address can be programmed to be in a non-cacheable or cacheable memory segment. An external interrupt controller could provide additional interrupt inputs and a priority encoding scheme but there is no mechanism for providing auto vectoring. The burden for prioritizing interrupts and branching to the proper interrupt service routine is left to the software.

## ARM

The ARM architecture provides two external interrupt inputs: FIQ and IRQ. FIQ is higher priority than IRQ but both can be masked by software. Each interrupt has a hard vector associated with it and its own status register and subset of general purpose registers. An external interrupt controller could provide additional interrupt inputs with priority but there is no auto vectoring capability.

## IBM PowerPC 405GP Universal Interrupt Controller (UIC)

The UIC for the PowerPC 405GP provides 19 internal and 7 external interrupts. Eighteen of the internal interrupts are active high, level sensitive. The other internal interrupt is edge sensitive and active on the rising edge. The seven external interrupts are programmable as to polarity and sensitivity. Each interrupt source can be programmed to source the critical or non-critical input to the 405 core. All interrupts can be masked and the current interrupt state can be read by the processor. The UIC supports prioritized auto vectoring for the critical interrupts, either through a vector table or the actual address of the service routine. The UIC does not support vector generation for the non-critical interrupts, relying instead on the interrupt vector generation mechanism within the PPC405 core. This mechanism is similar to a hard vector, except that the vector used is programmable by the software.

## Interrupt Controller Organization

The INTC is organized into four main functional units: Interrupt detection, DCR Bus decoding, Vector generation, and Data Bus muxing. [Figure 20-1](#) illustrates a detailed block diagram of the INTC. The interrupt detection logic consists of a set of registers which allow for selection of edge or level sensitivity, enable/disable logic for each interrupt source, registration of the interrupt event, pending interrupt logic, a master enable register, and finally an interrupt output. The DCR Bus Decoding logic handles the generation of the clock enables and write enables for each register, along with the DCR acknowledge pulse. The vector generator consists of the pending interrupts register feeding a priority encoder



which is made available to be read from DCR data bus. Finally, the data bus muxing handles all the multiplexing for reading back the register contents.

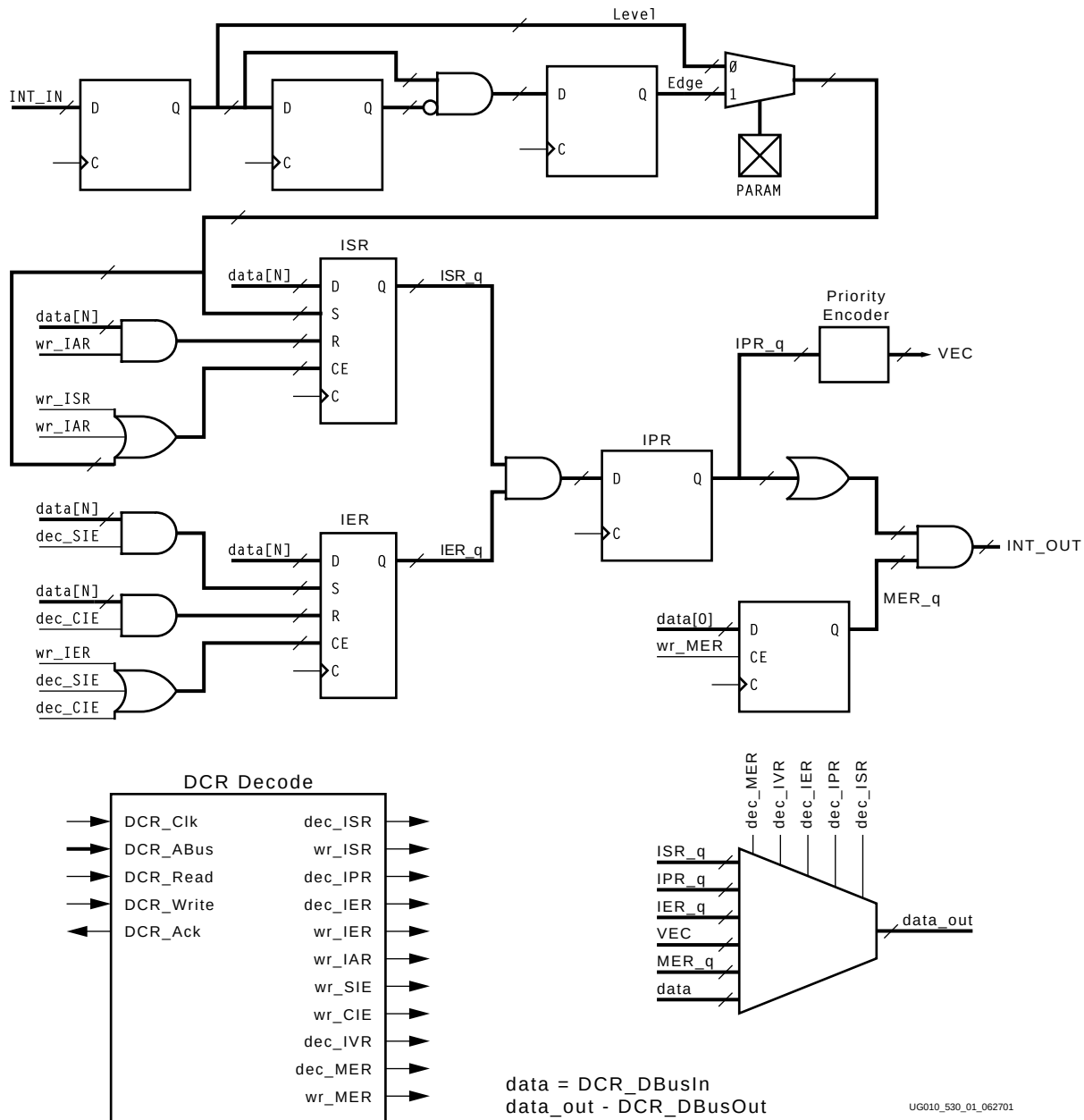
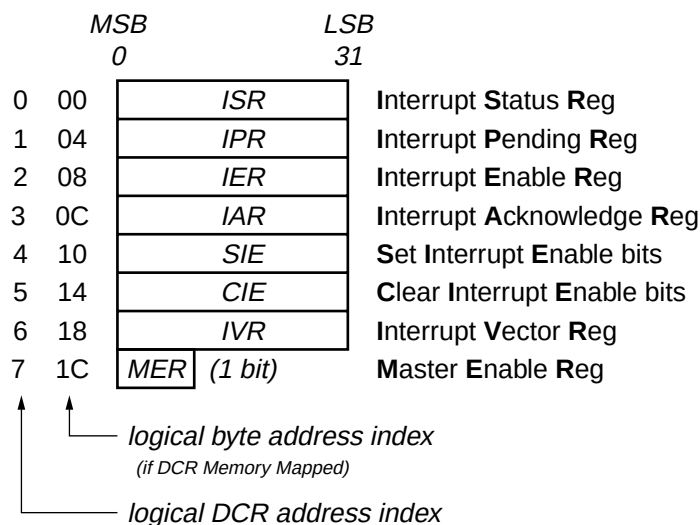


Figure 20-1: INTC Block Diagram

**Figure 20-2** summarizes the programmers view of the INTC. There are eight total addressable registers. All DCR registers are 32 bits wide, although not all of those bits must be used. Since the INTC can be parameterized in the number of interrupt inputs it can support, the number of bits in the registers will vary. Please see the detailed register description for further information.

In cases where the DCR bus has been memory mapped, the registers should be thought of as being 32-bit word aligned. All registers will consume four bytes of space, regardless of

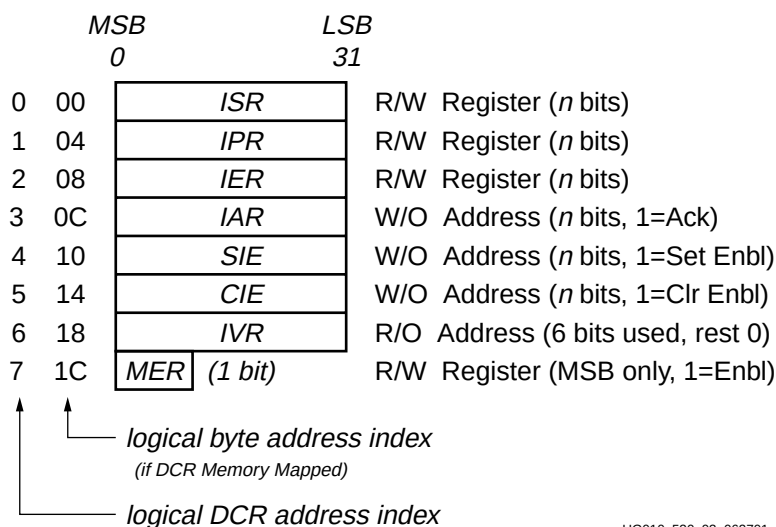
the number of interrupt inputs. This allows software a consistent model to work with independent of the number of interrupts used.



UG010\_530\_02\_062701

Figure 20-2: INTC Programmer's Model

Figure 20-3 illustrates a summary of the physical register implementations. This includes details about whether the register is readable and/or writable, and physical register implementation vs. address decode for the "register" functionality. Note that both the programmer's model (Figure 20-2) and physical hardware implementation model (Figure 20-3) include a byte address index (in hex) and the DCR bus address index. The DCR bus address index is included when the PPC405 **mtdcr** and **mfdcr** instructions are used. The byte address index is used when the DCR bus is memory mapped. The DCR bus, when memory mapped, consumes 4 bytes of address space per DCR register therefore the byte address increments by four for each register.



UG010\_530\_03\_062701

Figure 20-3: INTC Physical Hardware Model

## INTC Detailed Register Description

### General Information about INTC registers

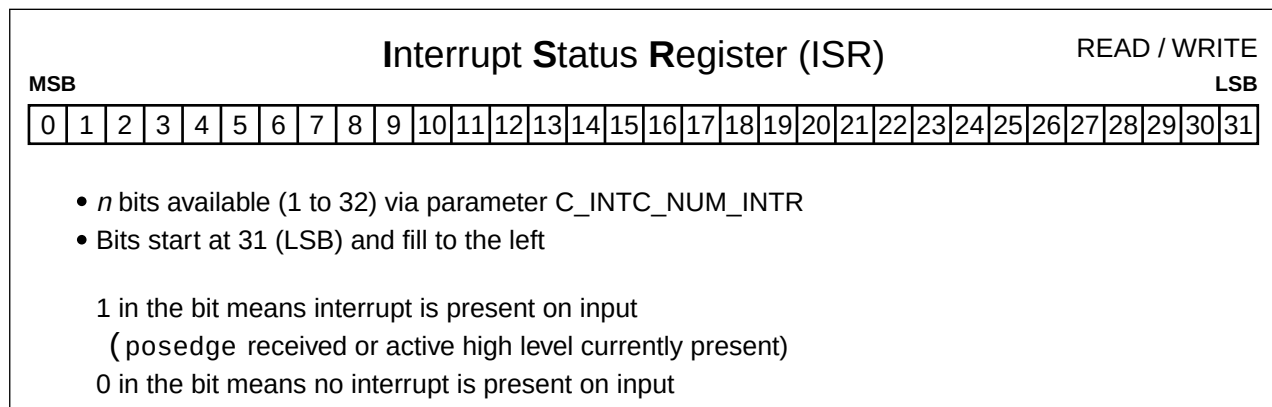
The INTC contains eight registers shown in the programmer's model. Each register will be described in detail in the following pages.

The Interrupt Vector Register, and Master Enable Registers contain a fixed set of bits, as illustrated below. However, the remaining six registers can contain from 1 to 32 bits, depending upon the number of interrupts that have been chosen. These six registers (ISR, IPR, IER, IAR, SIE, and CIE) fill from the LSB to the MSB. Thus, if 1 interrupt is requested, the INTC will use the LSB (bit 31) of these registers. If 16 interrupts were used, then these six INTC registers would use bits 16 to 31.

### Interrupt Status Register (ISR)

The Interrupt Status Register (ISR) is used to hold the contents of currently recognized interrupts. It shows the “live” contents of the interrupt inputs on a bit for bit basis. A one in any given bit means that there is an interrupt on the input pin. A zero in the bit means that no interrupt is present.

Figure 20-4 shows the general Interrupt Status Register, and describes in brief its functionality.



UG010\_530\_04\_062701

Figure 20-4: Interrupt Status Register (ISR)

The “live” contents of the interrupt sources are available via the ISR. When an interrupt is level sensitive, an active high input on an interrupt pin will cause the corresponding bit in the ISR to go to a one after a single DCR clock synchronization delay. Once the input is returned low, the ISR bit will remain at a one until it has been acknowledged by a write to the IAR. The INTC does not attempt to maintain event information. If the level disappears and reappears again prior to the acknowledgement, the “extra” interrupt will be lost.

When the interrupt is chosen to be edge sensitive, then a positive edge transition on the interrupt pin will cause the corresponding ISR bit to be set to a one. The ISR bit will remain at a one until it is cleared by a write with a one in the corresponding bit of the Interrupt Acknowledge Register (IAR). It is important to note that an edge sensitive interrupt can transition from low to high and will not cause the event to be lost. If the input subsequently transitions from low to high before the interrupt has been acknowledged, this second interrupt will be lost. The INTC does not maintain history of events and their acknowledgements.

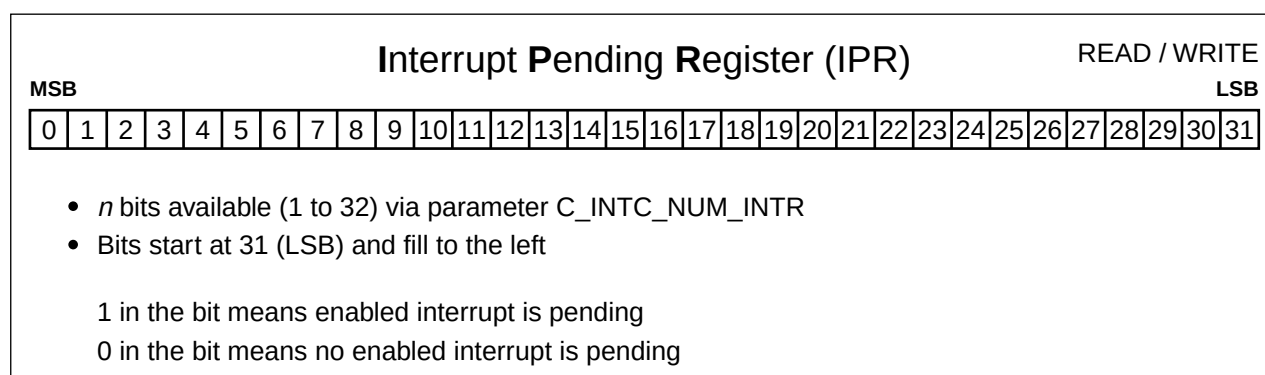
The ISR is a readable and writable register. Generally, the ISR is not used by software unless some form of polling is used. However, since the ISR is also writable, this offers software engineers the opportunity to force interrupts for driver code testing and for INTC

hardware verification testing. This facility should only be used when the hardware sources are disabled at the device. That is, software should ensure that each device disables its interrupt output prior to writing to the ISR. This will avoid any situations wherein the software generated interrupt might be confused with a hardware generated interrupt. Xilinx provides interrupt handling code to accomplish these tasks. The Interrupt Handling Code document contains more information about this.

## Interrupt Pending Register (IPR)

The Interrupt Pending Register (IPR) is used to indicate which enabled interrupt sources are pending. A one in a bit in this register means that an enabled interrupt source has requested service from the CPU. A zero in the bit mean that either the interrupt source has been disabled in the Interrupt Enable Register (IER), or that the enabled interrupt source does not have an active interrupt.

Figure 20-5 shows the general Interrupt Pending Register, and describes in brief its functionality.



UG010\_530\_05\_062701

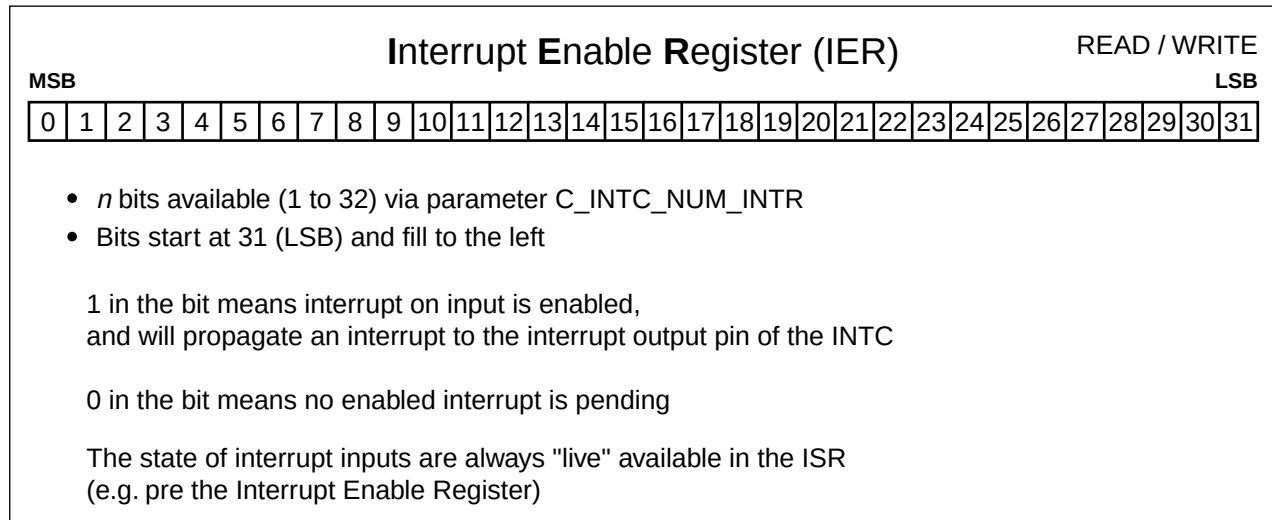
Figure 20-5: Interrupt Pending Register (IPR)

The IPR is a read only register. While it is available for software polling of enabled interrupt sources, it will not generally be used by software. It is a required function because it registers the live enabled values of interrupt sources prior to the priority encoder which generates the vector for the Interrupt Vector Register (IVR). Generally software would use the IVR in order to determine which interrupt has highest priority. However, the IPR may be used to look in parallel at which enabled interrupt sources are currently pending. This offers software the opportunity to evaluate any other outstanding interrupts which could wish to be dealt with before existing the service routines.

## Interrupt Enable Register (IER)

The Interrupt Enable Register (IER) is used to enable interrupt sources. A one a bit position within the IER means that the corresponding interrupt source is enabled. A zero in the bit will disable the interrupt source. When an incoming interrupt occurs, if it is enabled, it will be forwarded to the Interrupt Pending Register for prioritization and will cause the interrupt output of INTC to reflect that interrupt service is requested.

Figure 20-6 shows the general Interrupt Enable Register, and describes in brief its functionality.



UG010\_530\_06\_062601

Figure 20-6: Interrupt Enable Register (IER)

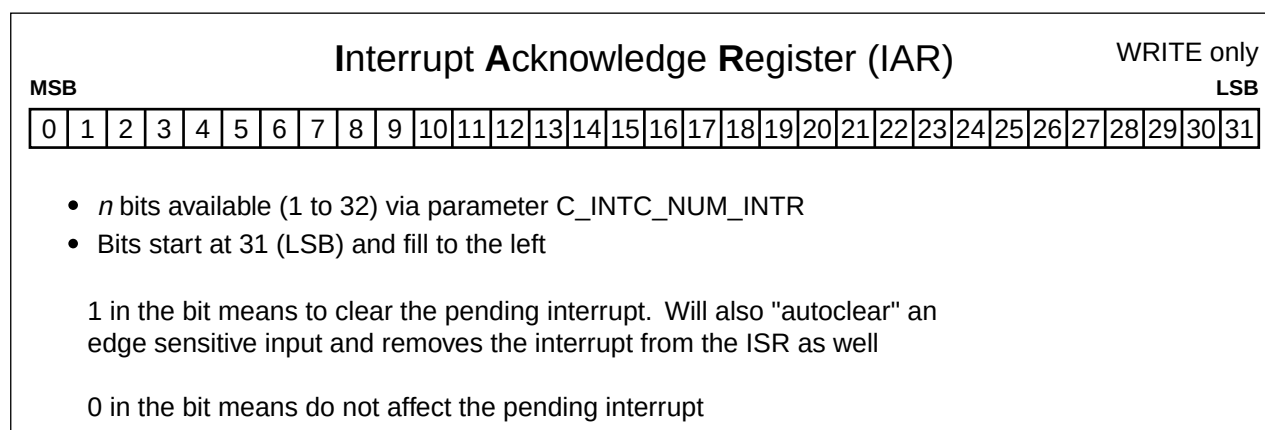
The IER is a read/write register. The IER is available to software in order to enable and disable multiple interrupts sources at the same time. This is a typical use model for many popular interrupt controllers. However, INTC provides some additional functionality in the form of the Clear Interrupt Enable and Set Interrupt Enable registers. These registers, described in greater detail below, allow software to enable or disable individual interrupt sources without affecting other interrupt sources. Without the ability to set or clear a specific enable individually, software would need to read the contents of the IER, modify them, and finally write the modified contents - a process which consumes valuable time during the interrupt service routine.

### Interrupt Acknowledge Register (IAR)

The Interrupt Acknowledge Register (IAR) is used to acknowledge pending interrupt sources. A one written to the IAR will cause the corresponding bit in the Interrupt Status Register (ISR) to be cleared. If another interrupt is pending on the interrupt source pin which is being cleared by the write to IAR, the ISR will be briefly cleared, and then will be set again. This functionality allows for a device to continue to request service.

Note: if an interrupt is edge sensitive, then another edge must occur after the IAR write in order to be recognized.

Figure 20-7 shows the general Interrupt Acknowledge Register, and describes in brief its functionality.



UG010\_530\_07\_062601

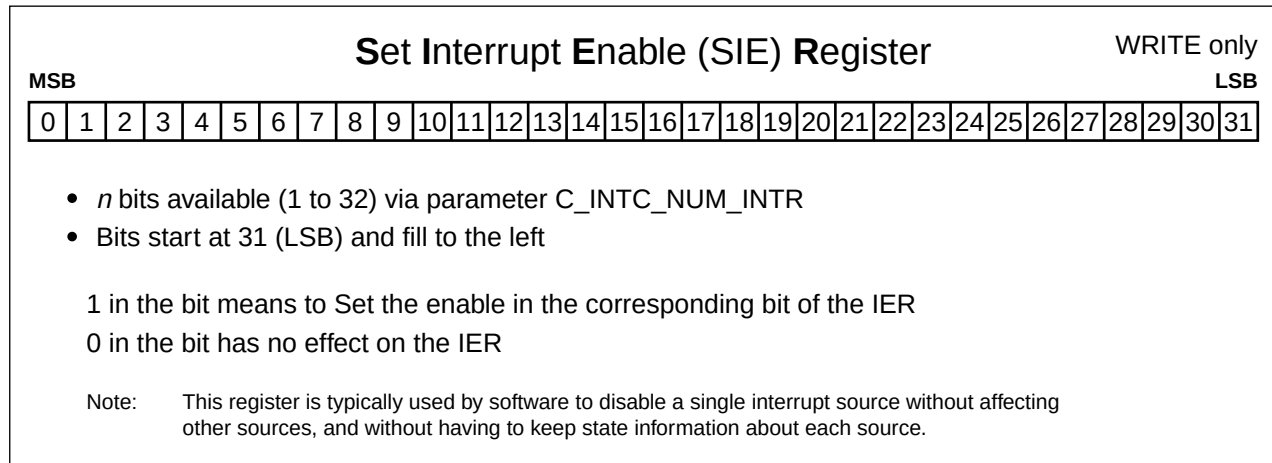
**Figure 20-7: Interrupt Acknowledge Register (IAR)**

The IAR is a write only register. This register, in concert with the Interrupt Vector Register (IVR), Set Interrupt Enable (SIE), and Clear Interrupt Enable (CIE) registers, are used by software to effect the interrupt service required by the system. In general, the SIE and CIE are used to enable and disable interrupt sources (respectively). When an interrupt occurs, it is registered in the Interrupt Status Register (ISR). If the interrupt is enabled, (based upon a '1' in the corresponding bit position of the Interrupt Enable Register), the interrupt will be propagated to the Interrupt Pending Register. Interrupts in the IPR will be prioritized, and will ultimately result in setting the interrupt output of the INTC. Once software recognizes that an interrupt has been generated by the INTC, it will read the Interrupt Vector Register to determine the highest priority enabled interrupt which is requesting service. The IVR vector is used as an index into a dispatch table which contains the address of the first line of code for the handling of this sources interrupt. Once software has determined which interrupt is to be serviced, it generally will service that interrupt. This involves going out to the device which generated the interrupt and figuring out what it needs. Once the device has been properly serviced, software should ensure that the device clears its interrupt source back into the INTC. Once this is done, the INTC can then be acknowledged by software writing a one to the appropriate bit of the Interrupt Acknowledge Register.

### Set Interrupt Enable (SIE) Register

The Set Interrupt Enable (SIE) register is used to enable one or more interrupt sources. A one in a bit position will cause the corresponding bit in the Interrupt Enable Register to be set (i.e. enabled). A zero in a bit position has no effect.

Figure 20-8 shows the general Set Interrupt Enable register, and describes in brief its functionality.



UG010\_530\_08\_062701

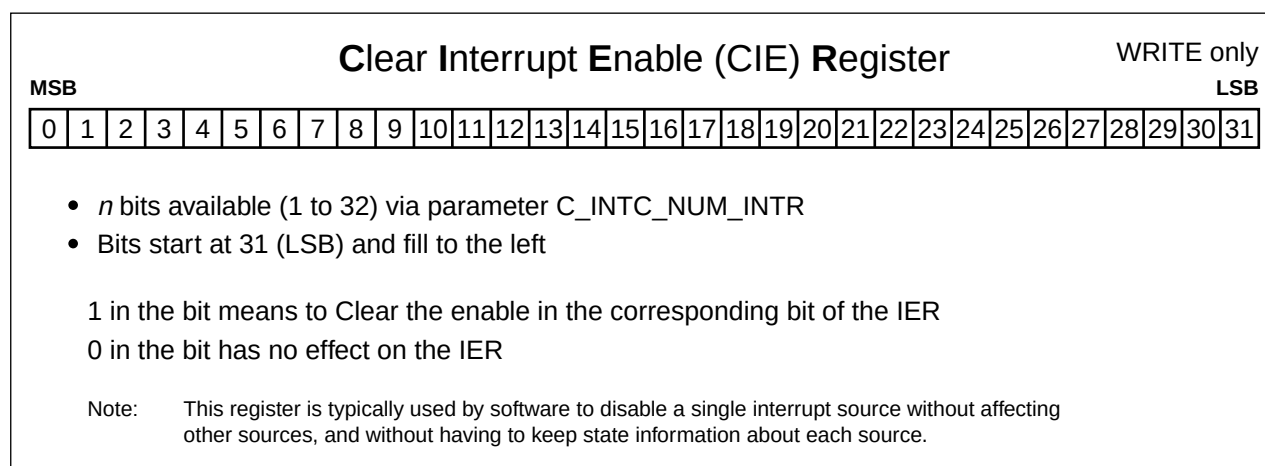
**Figure 20-8: Set Interrupt Enable (SIE) Register**

The SIE is a write only register. This register is used in concert with the Clear Interrupt Enable (CIE) register to enable or disable individual interrupt sources. In most interrupt controllers, software must read the enable register, modify the contents of this register, and then write back the modified contents. While the INTC permits this type of operation to occur, it provides the SIE and CIE registers as a means of simplifying the interrupt architecture. The main advantage of the SIE and CIE registers is that they permit one or more interrupt sources to be enables or disabled respectively without affecting other bits in the IER. This means that the interrupt service routine does not require contextual information about other interrupt sources. Rather, the interrupt service routine for the specific interrupt source can limit its scope to the specified source.

### Clear Interrupt Enable (CIE) Register

The Clear Interrupt Enable (CIE) register is used to disable one or more interrupt sources. A one in a bit position will cause the corresponding bit in the Interrupt Enable Register to be cleared (i.e. disabled). A zero in a bit position has no effect.

Figure 20-9 shows the general Clear Interrupt Enable register, and describes in brief its functionality.



UG010\_530\_09\_062701

Figure 20-9: Clear Interrupt Enable (CIE) Register

The CIE is a write only register. This register is used in concert with the Set Interrupt Enable (SIE) register to enable or disable individual interrupt sources. In most interrupt controllers, software must read the enable register, modify the contents of this register, and then write back the modified contents. While the INTC permits this type of operation to occur, it provides the SIE and CIE registers as a means of simplifying the interrupt architecture. The main advantage of the SIE and CIE registers is that they permit one or more interrupt sources to be enables or disabled respectively without affecting other bits in the IER. This means that the interrupt service routine does not require contextual information about other interrupt sources. Rather, the interrupt service routine for the specific interrupt source can limit its scope to the specified source.

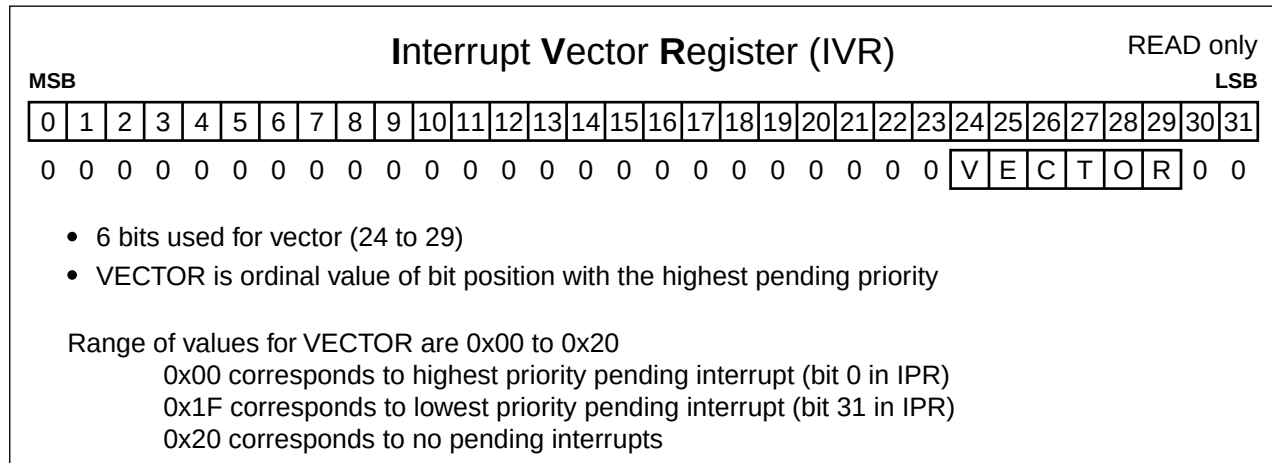
## Interrupt Vector Register (IVR)

The Interrupt Vector Register (IVR) is used to feedback a prioritized index to an enabled pending interrupt with the highest priority. The IVR provides a vector of 33 possible values: 32 interrupt sources and one "null" state. The "null" state is where there are no enabled pending interrupts and its vector value is 0x00000080.

Figure 20-10 shows the general Interrupt Vector Register, and describes in brief its functionality. Note that bits 0:23 are zero, but bits 24:29 contain the vector. Also note that bits 30 and 31 are filled with zeros. These two zeros allow software to use the vector as an



index to a table of addresses. This table of addresses would normally be filled with the pointer to each interrupt service routine.



UG010\_530\_10\_062601

Figure 20-10: Interrupt Vector Register (IVR)

The IVR is a read only register. It is used by software to return an ordinal corresponding to the highest enabled interrupt source that has a pending interrupt. The vector has 33 values: 0x00 to 0x20. The vector number corresponds to the interrupt input source. Thus, vector 0x00 corresponds to interrupt source 0, while interrupt source 31 corresponds to vector 0x1F. A null value of 0x20 is provided to detect that no enabled interrupts are pending.

Since PowerPC uses big endian naming convention, the INTC does as well. This means that interrupt input 0, IP2INTC\_IntrInput[0], corresponds to the MSB, while interrupt input 31, IP2INTC\_IntrInput[31], corresponds to the LSB. The LSB has the lowest priority and the MSB has the highest priority. This was done to keep the weighting of the priority similar to how data bits are weighted. However, the vector number result is based upon the actual bit number. Table 20-1 below enumerates each vector and the corresponding interrupt input from highest to lowest priority.

The dispatch table used by software will appear somewhat inverted in that interrupt source 0 (index 0 in the table) will have the highest priority while the last entry in the table would correspond to the lowest priority.

Reading the IVR has no effect on the interrupt source, nor on any registers within the INTC. This value will be update each time a higher pending interrupt is generated, or when the interrupt is acknowledged (edge sensitive) and removed (level sensitive) from the interrupt input pin.

The values in the IVR reflect pending interrupts. Pending interrupts are enabled in the Interrupt Enable Register (IER) and have an active interrupt source at the input.

In the cases where the DCR bus has been memory mapped into the system, the IVR could be addressed as a single byte at byte address 0x1B, if required. The previous three bytes (0x18, 0x19 and 0x1A) can be ignored since they will always be zero when read.

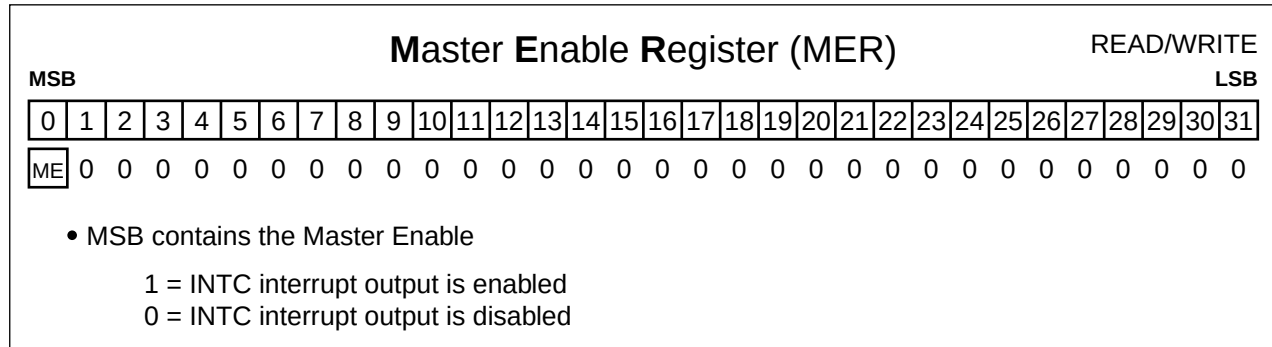
Table 20-1: Interrupt Vector Register Values

| IVR Value  | Vector | Interrupt Source Input Name | Priority    |
|------------|--------|-----------------------------|-------------|
| 0x00000000 | 0x00   | IP2INTC_IntrInput[0]        | 0 (Highest) |
| 0x00000004 | 0x01   | IP2INTC_IntrInput[1]        | 1           |
| 0x00000008 | 0x02   | IP2INTC_IntrInput[2]        | 2           |
| 0x0000000C | 0x03   | IP2INTC_IntrInput[3]        | 3           |
| 0x00000010 | 0x04   | IP2INTC_IntrInput[4]        | 4           |
| 0x00000014 | 0x05   | IP2INTC_IntrInput[5]        | 5           |
| 0x00000018 | 0x06   | IP2INTC_IntrInput[6]        | 6           |
| 0x0000001C | 0x07   | IP2INTC_IntrInput[7]        | 7           |
| 0x00000020 | 0x08   | IP2INTC_IntrInput[8]        | 8           |
| 0x00000024 | 0x09   | IP2INTC_IntrInput[9]        | 9           |
| 0x00000028 | 0x0A   | IP2INTC_IntrInput[10]       | 10          |
| 0x0000002C | 0x0B   | IP2INTC_IntrInput[11]       | 11          |
| 0x00000030 | 0x0C   | IP2INTC_IntrInput[12]       | 12          |
| 0x00000034 | 0x0D   | IP2INTC_IntrInput[13]       | 13          |
| 0x00000038 | 0x0E   | IP2INTC_IntrInput[14]       | 14          |
| 0x0000003C | 0x0F   | IP2INTC_IntrInput[15]       | 15          |
| 0x00000040 | 0x10   | IP2INTC_IntrInput[16]       | 16          |
| 0x00000044 | 0x11   | IP2INTC_IntrInput[17]       | 17          |
| 0x00000048 | 0x12   | IP2INTC_IntrInput[18]       | 18          |
| 0x0000004C | 0x13   | IP2INTC_IntrInput[19]       | 19          |
| 0x00000050 | 0x14   | IP2INTC_IntrInput[20]       | 20          |
| 0x00000054 | 0x15   | IP2INTC_IntrInput[21]       | 21          |
| 0x00000058 | 0x16   | IP2INTC_IntrInput[22]       | 22          |
| 0x0000005C | 0x17   | IP2INTC_IntrInput[23]       | 23          |
| 0x00000060 | 0x18   | IP2INTC_IntrInput[24]       | 24          |
| 0x00000064 | 0x19   | IP2INTC_IntrInput[25]       | 25          |
| 0x00000068 | 0x1A   | IP2INTC_IntrInput[26]       | 26          |
| 0x0000006C | 0x1B   | IP2INTC_IntrInput[27]       | 27          |
| 0x00000070 | 0x1C   | IP2INTC_IntrInput[28]       | 28          |
| 0x00000074 | 0x1D   | IP2INTC_IntrInput[29]       | 29          |
| 0x00000078 | 0x1E   | IP2INTC_IntrInput[30]       | 30          |
| 0x0000007C | 0x1F   | IP2INTC_IntrInput[31]       | 31 (Lowest) |
| 0x00000080 | 0x20   | no interrupt pending        | - - (none)  |

## Master Enable Register (MER)

The Master Enable Register (MER) is used to disable the entire INTC without altering the contents of any of its interrupt registers. The MER when set to a zero simply disables the interrupt output pin, INT2CPU\_Int. When the MER bit is set to a one, the INTC will rely upon values set in the other registers to determine when to output an interrupt.

Figure 20-11 shows the general Master Enable Register, and describes in brief its functionality.



UG010\_530\_11\_062601

Figure 20-11: Master Enable Register (MER)

The MER is a read/write register of a single bit (MSB, bit 0). The register is used by software to temporarily disable the interrupt output from the INTC. This is very useful when software needs to temporarily disable all interrupts from the INTC. An example of such usage is when an interrupt source requires very fast uninterruptible processing time. Without the Master Enable Register, software would be required to save and later restore the state of the registers which could leave timing holes in the interrupt coverage. It is also helpful in testing purposes to allow software to test the INTC prior to using it in the system.

**Note:** The MER bit is used to AND the INTC output pin. Therefore, when the MER is clear (i.e. zero) the INTC output pin (INTC2CPU\_Int) will forced to zero. However, any incoming interrupts will continue to propagate through the interrupt controller logic as last set by the CPU.

In the cases where the DCR bus has been memory mapped into the system, the MER could be addressed as a single byte at byte address 0x1C. The remaining bytes at 0x1D, 0x1E, and 0x1F will be zero for read, and no effect upon write.

## Parameters

The INTC's verilog or VHDL hardware implementation is controlled by a number of parameters which will allow the system integrator to only pay for as much logic as required by the intended functionality. The INTC currently uses five parameters to control the implementation.

The five are listed below, with descriptive text of how to use each.

### C\_INTC\_DCR\_DECODE\_ADDR (Default Value: 10'h3f8)

The C\_INTC\_DCR\_DECODE\_ADDR parameter is used to set the DCR base address of the INTC. Since the INTC consumes eight DCR registers, (base address +0 to base address +7), it is important to ensure that no other DCR devices conflict with the setting of the INTC. The default value for this parameter is 0x3F8.

In the PowerPC, two interrupt inputs are provided: Critical interrupt and Non-Critical (also known as External). Xilinx systems will instantiate two INTCs: one for Critical interrupt and one for Non-Critical. The base address for the Critical INTC will be set at 0x3F8, whereas the Non-Critical INTC will be set at 0x3F0.

When the DCR bus is memory mapped, the DCR base address must be multiplied by four to create the proper memory address. This is due to the fact that the DCR bus is a 32-bit wide bus, and would thus consume four bytes per DCR register. Each INTC therefore consumes 8 register x 4 bytes or 32 bytes of memory mapped space.

The C\_INTC\_DCR\_DECODE\_ADDR requires a 10-bit value under normal circumstances. If the C\_DCR\_ABUS\_NUM\_BITS parameter is adjusted, the C\_INTC\_DCR\_DECODE\_ADDR parameter will need to be adjusted to the number of bits specified by C\_DCR\_ABUS\_NUM\_BITS

## C\_INTC\_EDGE\_SENSE

(Default Value: 32'b1111\_1111\_1111\_1111\_1111\_1111\_1111\_1111)

The C\_INTC\_EDGE\_SENSE parameter is how edge vs. level detection is set on each interrupt input. C\_INTC\_EDGE\_SENSE uses a 32 bit constant to determine which pins will be set as edge sensitive. A one in a bit means the corresponding interrupt input will be positive edge sensitive. A zero in a bit means the corresponding interrupt input will be active high true level.

C\_INTC\_EDGE\_SENSE has a default value of 32'b1111\_1111\_1111\_1111\_1111\_1111\_1111\_1111. This means that all inputs are set to be edge sensitive. Note that the default values assume that all thirty two inputs are required. C\_INTC\_EDGE\_SENSE must always contain the same number of bits specified by C\_INTC\_NUM\_OF\_INTS.

The left most bit conforms to the MSB while the right most bit conforms to the LSB. The INTC will always add interrupt inputs from LSB toward MSB based upon C\_INTC\_NUM\_OF\_INTS. Thus, the right most bit of C\_INTC\_EDGE\_SENSE will always conform to IP2INTC\_Interrupt[31] and continue to the left until C\_INTC\_NUM\_OF\_INTS bits are filled.

**Note:** If active low level sensitive interrupts are desired, the user must clear the corresponding bit of C\_INTC\_EDGE\_SENSE to zero and add an inverter to the interrupt source input of the INTC. If negative edge sensitive interrupt is desired, the user must set the corresponding bit of C\_INTC\_EDGE\_SENSE to one and add an inverter to the interrupt source input of the INTC.

## C\_INTC\_NUM\_OF\_INTS (Default Value: 32)

The C\_INTC\_NUM\_OF\_INTS parameter is used to set the total number of interrupts that the INTC will support. This parameter can be set from 1 to 32, so long as it does not exceed the C\_DCR\_DBUS\_NUM\_BITS parameter.

The bits in the registers fill from the LSB to the MSB. That is, if C\_INTC\_NUM\_OF\_INTS is set at 2, then Bits 30 and 31 will be used to house the interrupt sources. If C\_INTC\_NUM\_OF\_INTS is set at 16, then bits 16:31 would be used. When C\_INTC\_NUM\_OF\_INTS is set less than C\_DCR\_DBUS\_NUM\_BITS, then data to the left of the number of bits defined by C\_INTC\_NUM\_OF\_INTS should be ignored. (Except the IVR and MER which are in fixed bit positions regardless of the number of interrupts defined).

## C\_DCR\_ABUS\_NUM\_BITS (Default Value: 10)

The parameter C\_DCR\_ABUS\_NUM\_BITS defines the number of DCR address bits which are present in the system. This is fixed at 10 for DCR bus implementations on CPUs, but could be variable if the DCR is memory mapped. The Xilinx implementation of memory mapped DCR provides ten bits in order to remain compatible with the DCR toolkit. It is not recommended to change this setting.

## C\_DCR\_DBUS\_NUM\_BITS (Default Value: 32)

The parameter C\_DCR\_DBUS\_NUM\_BITS defines the number of DCR data bits which are present in the system. This is fixed at 32 for DCR bus implementations on CPUs, but could be variable if the DCR is memory mapped. The Xilinx implementation of memory mapped DCR provides 32 bits in order to remain compatible with the DCR toolkit. It is not recommended to change this setting.

## Expanding the Number of Available Interrupts (Daisy-Chaining Controllers)

Occasionally, some systems will need more than 32 interrupt inputs. These types of systems can be implemented by simple expansion techniques using the INTC. Care must be taken to ensure correct prioritization in the hardware layout. Additionally, software must be properly constructed to take advantage of the expanded INTCs.

The simplest way to extend the number of interrupts is to connect the highest priority INTC's INTC2CPU\_Int pin to the IP2INTC\_Intrinput[0] of the next highest priority.

Figure 20-12 details this below.

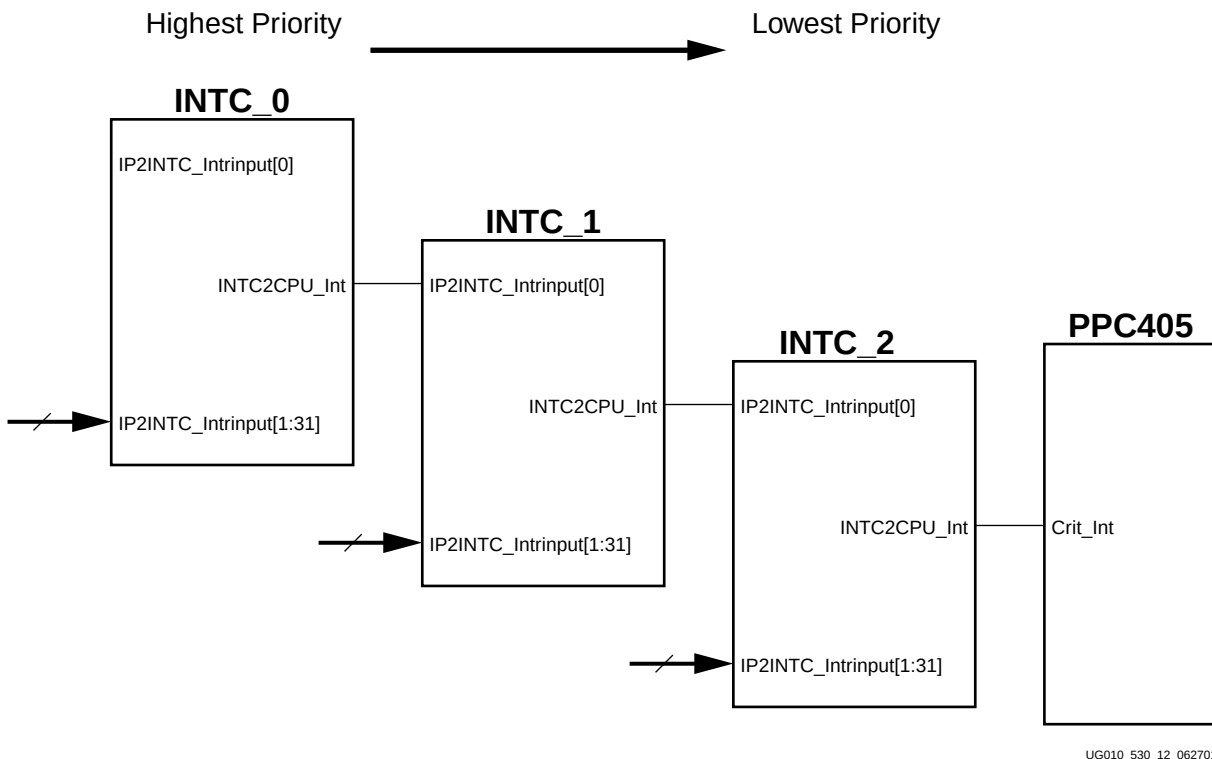


Figure 20-12: Expanding INTC interrupt sources

The connection of the higher priority INTC into the next highest priority INTC ensures that all operations from INTC\_0 will have higher priority than those in INTC\_1 and INTC\_2. Any reference to IVR register reading 0x00000000 in INTC\_2 (the one whose INTC2CPU\_Int pin is connected to the CPU) will mean that either INTC\_1 or INTC\_0 is requesting service.

**Note:** Xilinx software does not currently support daisy-chaining of INTCs.

## General Interrupt Information and Helpful Hints

The following information offers insight into the general process of interrupts.

### Interrupt Servicing vs. Interrupt Processing

It is important to note that interrupt servicing and interrupt processing are two distinctly different processes, although this line has often been blurred in the past. In general, interrupt servicing is the process wherein information which would otherwise be lost is forwarded on to a higher layer of software. In contrast, interrupt processing is the process wherein the information (which was forwarded by the interrupt service) is used for the purpose it was transmitted for.

One important issue that is sometimes overlooked is that if interrupt processing is done during the interrupt service, interrupt latency can be adversely affected. It is generally better to separate these two processes away from one another in order to maintain low latency and more “realtime-ness”.

### Interrupt Acknowledgment

It is important to acknowledge every interrupt that occurs to the INTC. The handling of level and edge sensitive interrupts are very similar. This is intentional so that software need not worry about a source being an edge or a level and thus have to treat them differently.

The general process of an interrupt is that a source will generate its interrupt to the appropriate interrupt input pin. This will in turn cause the ISR to reflect that an interrupt has been received. If this interrupt is enabled, then the interrupt will propagate to the IPR and therefore will show up on the interrupt output of the INTC. The CPU will read the IPR to determine the highest priority source to be serviced. When the source which generated the interrupt of this example has high enough priority, the CPU will begin the process of determining what service the source required. Typically, this would involve disabling the source in the INTC (via the CIE), and then going to the device to read its specific interrupt registers. Once the proper action has been taken at the device, the CPU should then acknowledge to the device that it has serviced the device's interrupt. This would typically be done by clearing the interrupt inside the device. After the device itself has been acknowledged, the CPU would then acknowledge the INTC using the IAR set to the appropriate value for the source. Lastly, the CPU would re-enable the source using the SIE. This general flow is followed regardless of whether the interrupting source is level or edge sensitive.

When an interrupt input is configured to be edge sensitive to the INTC, the input is treated as if it generates an event. Level sensitive inputs are often thought of as having a state rather than an event. When the edge interrupt occurs, the ISR contents are updated to a 1 to reflect that an interrupt has occurred. This 1 in the appropriate ISR bit will not go away until the CPU has either acknowledged the interrupt via writing to the appropriate bit in the IAR, or by forcing the bit to go away in the ISR by writing directly to the ISR. The INTC is designed to have the IAR clear the ISR. The CPU should only use the ISRs writable nature for testing purposes.

An interrupt input which is configured to be level sensitive in the INTC is often treated as a state. That is, the source is constantly requesting service until it has received service, whereupon it de-asserts its request for service. In order to create a more homogeneous environment, the INTC deals with level sensitive interrupts essentially the same as for edge. The same steps are used in either case: identify source in INTC, disable device in INTC, service source at the device, acknowledge source at the device, acknowledge source in the INTC, enable device in INTC is used. This keeps the software view of devices very orthogonal and allows the interrupt service routine code to be commonized to greater degree.

## Verilog Module Port Declarations

```

module INTC (
 DCR_Clk,
 DCR_DBusIn,
 DCR_DBusOut,
 DCR_Read,
 DCR_Write,
 DCR_ABus,
 DCR_Ack,
 INTC2CPU_Int,
 IP2INTC_IntrInput
);

parameter C_INTC_DCR_DECODE = 10'h3f8;
parameter C_INTC_DCR_DECODE_ADDR = 10'h3f8;

parameter C_INTC_EDGE_SENSE = 32'b1111_1111_1111_1111_1111_1111_1111_1111;
parameter C_INTC_NUM_OF_INTS = 32;
parameter C_DCR_ABUS_NUM_BITS = 10;
parameter C_DCR_DBUS_NUM_BITS = 32;

input DCR_Clk; // DCR Clock pin
input [0:C_DCR_DBUS_NUM_BITS - 1] DCR_DBusIn; // DCR data bus in from previous device
output [0:C_DCR_DBUS_NUM_BITS - 1] DCR_DBusOut; // DCR data bus out to next device
input DCR_Read; // DCR read signal
input DCR_Write; // DCR write signal
input [0:C_DCR_ABUS_NUM_BITS - 1] DCR_ABus; // DCR Address bus (normally only 10bits wide)
output DCR_Ack; // DCR signal to acknowledge INTC transaction, // ORed with other ACK
output INTC2CPU_Int; // Interrupt output of INTC to the CPU
input [0:C_INTC_NUM_OF_INTS - 1] IP2INTC_IntrInput; // Interrupt inputs to the INTC

```

# Packet Processing Engine Controller

---

## Overview

The Packet Processing Engine module (PPE) contains the packet processing logic, dcr\_module, and dual-port block SelectRAM (BRAM) memory.

The packet processing logic receives a transmit signal and transmit address from an external source via the dcr\_module. The logic uses these signals to read data from the BRAM while creating Start of Frame (SOF), End of Frame (EOF), and IDLE characters. A packet is created from these components and sent to the Rocket I/O™ transceiver.

During retrieval of received packets, the logic removes the SOF, EOF, and IDLE characters from the Rocket I/O receive data. Then, using the receive start and end addresses from the pkt\_proc\_dcr\_module, the data is written into the BRAM. Once a complete packet has been written to BRAM, the tail\_ptr is updated, signaling availability from the second BRAM port.

## Related Documents

- UG012, *Virtex-II Pro Handbook*
- *Rocket I/O Transceiver User Guide*
- [XAPP258](#), *FIFOs Using Virtex™-II Block RAM*
- IBM: *Device Control Register Bus Architecture Specification*

## Features

- 16-bit data interface with Rocket I/O transceivers
- 32-bit dual-port packet memory interface
- 1K, 16-bit halfword receive FIFO
- Flexible IDLE, SOF, and EOF characters for multiple protocols
- Microprocessor-controlled Packet Processing Control and Status Registers



## Module Port Interface

### Control/Status Signals

The interface logic receives its data and control from **pkt\_proc\_dcr\_module**, which is connected to the interface signals names as shown in **Table 21-1**. Input and output direction is from the Rocket I/O interface logic perspective.

**Table 21-1: PPE Interface Signals**

| Register Name                                                   | PPE Signal Name       | Direction        |
|-----------------------------------------------------------------|-----------------------|------------------|
| TX_Start_Addr                                                   | transmitaddr          | Input            |
| Frame_Size                                                      | packetlength          | Input            |
| RX_Start_Addr                                                   | rx_sa                 | Input            |
| RX_End_Addr                                                     | rx_ea                 | Input            |
| RX_Tail_Ptr                                                     | rx_tp                 | Output           |
| PPE_Control:<br>TX_Frame<br>RX_Enable<br>Loopback<br>TX_Disable | transmit<br>rx_enable | Input<br>Input   |
| PPE_Status:<br>TX_Complete<br>RX_Buffer_Full<br>Port_ID         | tdone<br>buff_full    | Output<br>Output |

## Rocket I/O Transceiver Interface Signals

The Rocket I/O transceiver interface consists of a 16-bit transmit bus, a 16-bit receive bus, a reference clock, and some control signals. The design uses the 8B/10B encoder and 8B/10B decoder functions in the transceiver. [Table 21-2](#) shows the interface signal and signal description. For detailed information about the Rocket I/O transceiver, see the *Rocket I/O Transceiver User Guide*.

**Table 21-2: Rocket I/O Transceiver Interface Signals**

| Rocket I/O Interface Signal Name | Direction | Description                                                                                                                                                                             |
|----------------------------------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TXDATA[31:0]                     | Input     | Upper 16 bits are forced to 0. Lower 16 bits contain character to transmit input. Character is either “data” or “k” controlled by <b>TXCHARISK[3:0]</b> inputs.                         |
| RXDATA[31:0]                     | Output    | Upper 16 bits are not used. Lower 16 bits contain character received from external RX serial input. Character is either “data” or “k” reported by <b>RXCHARISK[3:0]</b> signals.        |
| TXCHARISK[3:0]                   | Input     | Character is “k” code, one bit for each <b>TXDATA</b> byte lane. <b>TXCHARISK[3]</b> defines <b>TXDATA[31:24]</b> content. Upper two bits are forced to 0.                              |
| RXCHARISK[3:0]                   | Output    | Receive character is “k” code, one bit for each <b>RXDATA</b> byte lane. <b>RXCHARISK[3]</b> defines <b>RXDATA[31:24]</b> content. Upper two bits are not used in the reference system. |
| TXRESET                          | Input     | Transmitter reset. After reset is deasserted, there is a 9-clock delay before TX input character is valid.                                                                              |
| RXRESET                          | Input     | Receiver reset. After reset is deasserted, there is a 9-clock delay before RX output character is valid.                                                                                |
| TXBYPASS8B10B [3:0]              | Input     | Transmit 8B/10B Bypass. Controls if 8B/10B encoding is enabled.                                                                                                                         |
| REFCLK                           | Input     | Reference clock                                                                                                                                                                         |
| REFCLK2                          | Input     | Reference clock 2                                                                                                                                                                       |
| REFCLKSEL                        | Input     | Selects reference clock input (0 = REFCLK2, 1 = REFCLK)                                                                                                                                 |

There are 16 Rocket I/O transceiver primitives in the **unisim** library. These primitives have certain attributes that are set for specific protocol and transfer interfaces—for example, 8-, 16-, or 32-bit interfaces to the FPGA fabric. The Packet Processing Engine Reference System uses the GT\_CUSTOM primitive, which allows the most flexibility. Its interface with the FPGA fabric is shown in [Table 21-2](#).

Several of the GT\_CUSTOM primitive's default values must be changed for this packet processing application. Many of these attributes control the clock recovery of the transceiver. The values that must be changed for the PPE design are shown in [Table 21-3](#).

Table 21-3: GT\_CUSTOM Primitive Attributes Used in Packet Processing Engine

| Attribute            | Value       | Meaning                                      |
|----------------------|-------------|----------------------------------------------|
| DEC_VALID_COMMA_ONLY | "FALSE"     | Comma validity testing <i>not used</i>       |
| CLK_COR_SEQ_1_1      | 00110111100 | IDLE pattern                                 |
| CLK_COR_SEQ_1_2      | 00010010101 | IDLE pattern                                 |
| CLK_COR_SEQ_1_3      | 00010010101 | IDLE pattern                                 |
| CLK_COR_SEQ_1_4      | 00010110101 | IDLE pattern                                 |
| RX_LOSS_OF_SYNC_FSM  | "FALSE"     | Out-of-sync output detection <i>not used</i> |

## BRAM Signals

Refer to **18 Kb Block SelectRAM Resources in Virtex-II Pro™ FPGAs: Functional Description** (Data Sheet, Module 2).

## Implementation

### Rocket I/O Transceiver Interface Logic

The packet processing user block is the interface that connects the PPC405 to the Rocket I/O transceiver. It interacts with the PPC405 by way of the **pkt\_proc\_dcr\_module** and directly to the transceiver. The packet processing user block creates the **SOF**, **EOF**, and **IDLE** characters that are needed to send a packet. Once the transceiver starts receiving data, the packet processing user block removes the **IDLE** characters and places the packet in a receive FIFO until the receive portion of the packet processing acquires control of the BRAM from the transmitter portion. Once data from one packet has been written to BRAM, the **RX\_Tail\_Ptr** register is updated to allow the processor to retrieve data.

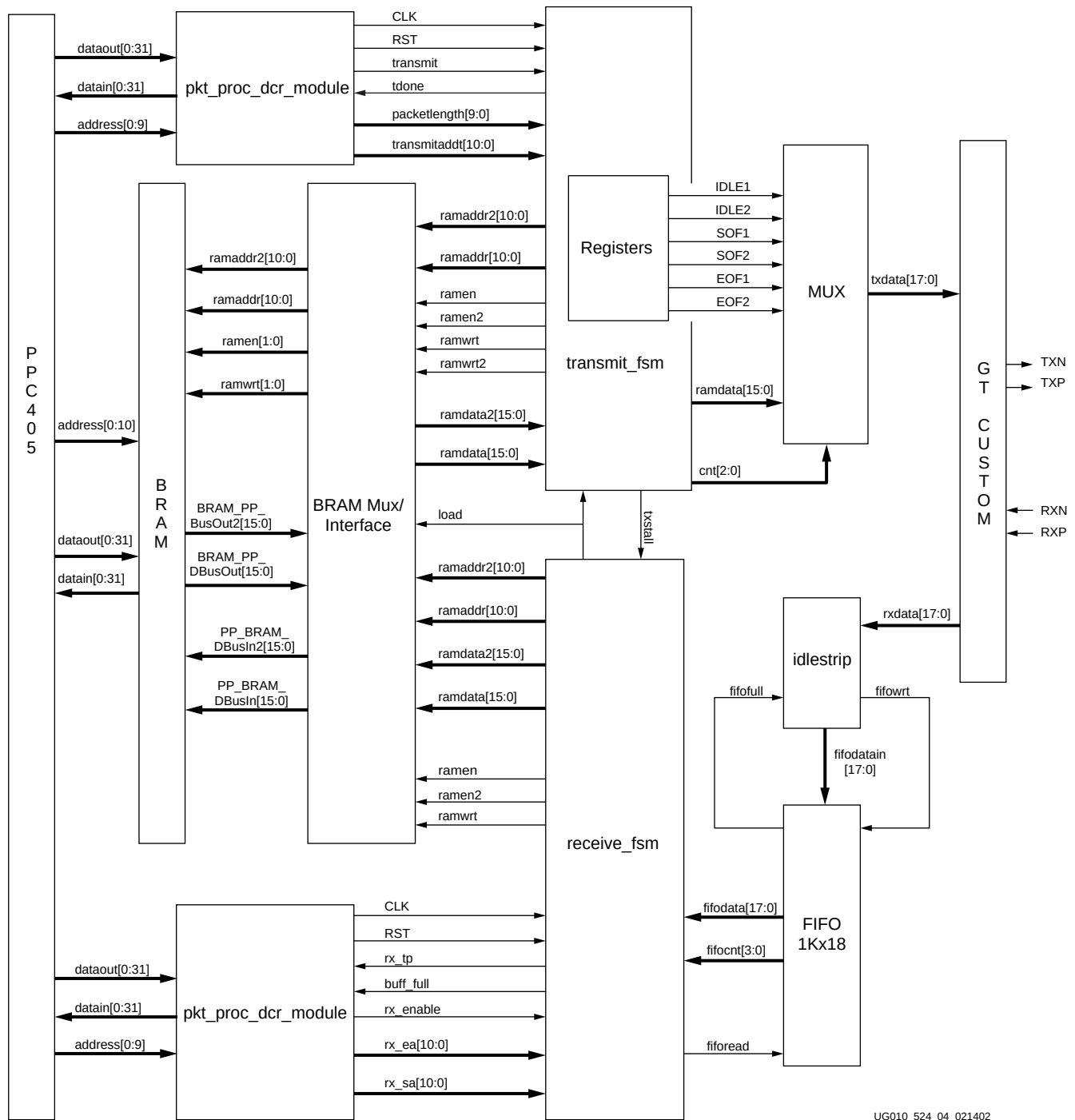


Figure 21-1: Overview of the Packet Processing Engine Interface Logic

## Transmit Function

A portion of the BRAM is used as a frame buffer to store outgoing frames.

A DMA controller is used to read the BRAM and transfer data from the memory to the Rocket I/O interface. The following registers are required for PPC405 control of the DMA controller:

- Start Address register (4KB BRAM address range)

- Frame Size register (0 to 1024 word max frame size)
- Start DMA transfer control bit. Start transfer on a rising edge of this signal.
- DMA Transfer Complete status bit. The completion bit clears the Start DMA Transfer bit so the processor can initiate another transfer.
- Address counter to address the BRAM TX buffer. The Start Address register is loaded into the address counter when the **Start\_TX** bit is asserted. Both transmit and receive logic must drive the BRAM address.
- A transmit logic frame generator including a data multiplexor and control logic. The frame generator controls when to send **SOE**, data, and **EOE** once the **Start\_TX** bit is asserted. The control logic also controls the **TXCHARISK** inputs to the Rocket I/O core.

The PPE interface logic transmits an IDLE sequence until the **Start\_TX** bit is asserted. A minimum of six IDLE ordered sets must be transmitted between frames. The Start Address register gets loaded into a counter that addresses the BRAM when the **Start\_TX** signal is asserted. The DMA controller reads 16-bits from BRAM and sends to the Rocket I/O interface logic.

## Receive Functions

The PPE interface logic contains the following functions:

- **RX\_Start\_Addr** and **RX\_End\_Addr** registers that the processor uses to define the physical start and end address of the data buffer for receive frames.
- Frame Size register that defines the fixed size of the incoming frame in halfwords.
- A counter to address to BRAM. The BRAM address is driven by both the transmit DMA function and the receive frame logic. This requires a multiplexor on the BRAM address from the PPE interface logic.
- A read only **RX\_Tail\_Ptr** register that the hardware increments when a complete frame has been written into the BRAM. The firmware polls this register to determine when a frame has been received. If this register equals the **RX\_End\_Addr** register, the hardware must stop receiving incoming frames until the processor has created additional buffer space. The **RX\_Buffer\_Full** status bit is asserted when the buffer is full.
- Frame decoding logic to look for incoming frames, strip off the **SOE** and **EOE** characters, and store the data in the BRAM.

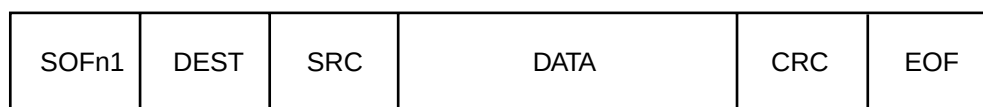
The Frame Size register determines the fixed size of the incoming frame.

Once the firmware has initialized the start and end buffer pointers, and asserted the **RX\_ENABLE** control bit, the hardware looks at incoming frames and writes into the receive FIFO. The hardware increments **RX\_Tail\_Ptr** after it has completed writing the receive frame into the RX buffer. Initially, the hardware uses **RX\_Start\_Addr** to load into the RX address counter used to access the BRAM.

For more information on the receive FIFO, see [XAPP258, FIFOs Using Virtex-II Block RAM](#).

## Frame Format

The reference design implements a frame, shown in [Figure 21-2](#). The frame consists of a start-of-frame primitive, a header field, a data field, an optional CRC field, and an end-of-frame primitive.



UG010\_524\_03\_100901

Figure 21-2: Rocket I/O Transceiver Serial Frame Format

The start-of-frame (SOFn1) and end-of-frame (EOF) primitives are defined in [Table 21-4](#).

The frame header includes the following information:

- A 32-bit DESTINATION port address that identifies the receiver port information
- A 32-bit SOURCE port address that identifies the sender of the frame

### Frame Size

The Frame Size register defines the fixed length of the frame in halfwords. The Frame Size register defines both the transmit and receive frame size. The SOFn1, optional CRC, and EOF characters are not included in the Frame Size register. The Frame Size register is 11 bits wide, which provides a maximum frame size of 4KB.

### Protocol Characters

When the port is not transmitting, an IDLE primitive signal, defined in [Table 21-4](#), is transmitted. The design does not attempt to implement higher-level communications protocols such as port initialization, port reset, port discovery, physical addressing, or buffer credit management. The K28.5 character, referred to as a *comma*, is used for character and word boundary alignment. A minimum of six IDLEs are transmitted between frames.

Table 21-4: Ordered Set Primitives

| Framing Protocol Character | Ordered Set  | TXCHARISK[1:0]<br>or<br>RXCHARISK[1:0] | TXDATA[15:0]<br>or<br>RXDATA[15:0] |
|----------------------------|--------------|----------------------------------------|------------------------------------|
| IDLE                       | K28.5, D21.4 | 10                                     | BC95                               |
|                            | D21.5, D21.5 | 00                                     | B5B5                               |
| SOFn1                      | K28.5, D21.5 | 10                                     | BCB5                               |
|                            | D23.1, D23.1 | 00                                     | 3737                               |
| EOFn                       | K28.5, D21.4 | 10                                     | BC95                               |
|                            | D21.6, D21.6 | 00                                     | D5D5                               |

### Half-Duplex Operations

The design supports half-duplex operation. (Full-duplex operation would require separate BRAMs for simultaneous receive and transmit operations.) The BRAM is partitioned into a transmit buffer and a receive buffer. The dual-port BRAM does not support simultaneous reads and writes to different addresses. A receive FIFO is placed in the RX data path, between the gigabit transceiver and the BRAM. The RX FIFO provides temporary storage for incoming frames if the transmit frame logic is accessing BRAM. The RX FIFO is 1K halfwords deep.

### Transmit Buffer

The TX buffer holds a maximum of 4K bytes. The **SO**F and **EO**F characters are not stored in the buffer. The **SO**F and **EO**F characters are constants that are multiplexed into the TX data stream by a state machine sequencer in the Rocket I/O interface logic module, **pkt\_proc**.

### Receive Buffer

Receive frames arrive asynchronously. In order to handle simultaneous transmit and receipt of frames, a receive FIFO is required to hold incoming frames until the frame transmit function can be paused. The RX FIFO holds a maximum of (2K –2) bytes.

Moving receive frames into the BRAM has priority over transmitting frames.

## Data Acquisition/Storage

### Device Control Register (DCR) Bus Interface

The processor controls the transmission and reception of serial frames through the PPE Control Registers. The OCM controller forwards all non-OCM DCR accesses on to the FPGA. The FPGA asserts the DCR acknowledge signal to terminate the DCR bus cycle. The DCR cycle must complete within 64 processor clocks.

Table 21-5: Packet Processing Engine DCR Control Registers

| Address (hex) | R/W | Register Name | Register Function                                                                                                                                                                  |
|---------------|-----|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 200           | R/W | TX_Start_Addr | This register defines the starting address for the transmit frame. The TX buffer is a 2KB buffer.                                                                                  |
| 201           | R/W | Frame_Size    | This register defines the frame size in halfwords (16 bits). The frame size does not include the <b>SO</b> F and <b>EO</b> F characters. A maximum frame size of 2KB is supported. |
| 202           | R/W | RX_Start_Addr | Defines the starting address of the receive buffer.                                                                                                                                |
| 203           | R/W | RX_End_Addr   | Defines the ending address of the receive buffer.                                                                                                                                  |

Table 21-5: Packet Processing Engine DCR Control Registers (Continued)

| Address (hex) | R/W                 | Register Name | Register Function                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|---------------|---------------------|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 204           | RO                  | RX_Tail_Ptr   | <p>The PPE control logic uses this register to inform the firmware that it has put a frame into the RX buffer. The value changes only when the entire frame has been written into BRAM.</p> <p>If <b>RX_Tail_Ptr</b> == <b>RX_End_Addr</b>, then the hardware ceases writing incoming frames into the buffer and assert the <b>RX_Buffer_Full</b> status bit.</p>                                                                                                                                                   |
| 205           | R/W                 | PPE_Control   | <p>This register controls the PPE control logic and Rocket I/O transceiver's operation, enabling the processor to transmit and receive frames, test the transceiver, and disable the transmitter output.</p> <p>Bit 0: <b>TX_Frame</b></p> <p>Bit 1: <b>RX_Enable</b></p> <p>Bit 2-3: <b>Loopback</b>:</p> <p>00 = normal transmit/receive operation<br/> 01 = loopback, parallel data path<br/> 10 = loopback, serial data path<br/> 11 = invalid</p> <p>Bit 4: <b>TX_Disable</b>: disable transmitter outputs</p> |
| 206           | 0-1: R/W<br>2-3: RO | PPE_Status    | <p>Rocket I/O transceiver status information. Firmware must write register bit to clear status (bits 0 &amp; 1 only). This register controls the Rocket I/O transceiver's operation, enabling the processor to transmit and receive frames.</p> <p>Bit 0: <b>TX_Complete</b> (@RESET = 1)</p> <p>Bit 1: <b>RX_Buffer_Full</b></p> <p>Bit 2-3: <b>Port_ID</b> (read only):</p> <p>00 = invalid port ID (reset state)<br/> 01 = Port A<br/> 10 = Port B<br/> 11 = Port C</p>                                          |



## Processor DCR Interface

All packet processing control and status registers are designed to interface to the processor DCR bus. The DCR interface signal names are shown in [Table 21-6](#).

The **pkt\_proc\_dcr\_module** is used to configure peripherals with the PPC405. The module is composed of registers that can be read/write, read only, or write only. The Packet Processing Engine status and control bits are contained in the module, as well as transmit, tx\_complete, and address indicators. Input and output direction is from the processor perspective.

Table 21-6: PPC405 DCR Interface Signals

| DCR Signal Name       | Direction |
|-----------------------|-----------|
| C405_dcrABus[0:9]     | Output    |
| C405DCRREAD           | Output    |
| C405DCRWRITE          | Output    |
| DCRC405ACK            | Input     |
| C405_dcrDBusOut[0:31] | Output    |
| PP_dcrDBusIn[0:31]    | Input     |

# **Volume 8: Software IP and Applications**

***Virtex-II Pro™ Platform FPGA  
Developer's Kit***

March 2002 Release





The Xilinx logo shown above is a registered trademark of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

"Xilinx" and the Xilinx logo are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, XC5210 are registered Trademarks of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Benchner, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Benchner, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, Nano-Blaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, Rocket I/O, SelectI/O, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM      IBM Logo      PowerPC      PowerPC Logo      Blue Logic      CoreConnect      CodePack

All other trademarks are the property of their respective owners.

Xilinx does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 2002 Xilinx, Inc. All Rights Reserved.

# *Introduction to Software IP and Applications*

---

**TBD**



## Stand-Alone Board Support Package

### Overview

The Board Support Package (BSP) is located in the `$V2PRO/source/sw/libs/bsp` installation directory. The BSP is a set of software modules combined into the `bsp.a` library. On one hand, the BSP offers an interface to peripheral devices and to low-level PowerPC® core functions. A stand-alone application uses this interface to access the hardware. On the other hand, the BSP is required when an application is linked with the C library. In this case the BSP provides functionality that allows the C library to access the hardware.

The BSP is a work in progress. Changes to the interface, implementation, and functionality are planned for the future.

*Table 2-1: BSP Directory Structure*

| Subdirectory  | Contents                                |
|---------------|-----------------------------------------|
| include       | include files for the BSP               |
| v2pdk         | source files specific for V2PDK         |
| v2pdk/debug   | BSP library compiled with debug flags   |
| v2pdk/release | BSP library compiled with release flags |
| build         | object files                            |

### Building the BSP

The BSP can be built or rebuilt by running the **make** command for a desired target. The Makefile specifies how the files are to be built. For example, the following command builds the BSP for V2PDK and rebuilds only changed source files:

```
$ make bsp-v2pdk
```

To perform a complete rebuild of the entire BSP for V2PDK, execute the following command:

```
$ make clean bsp-v2pdk
```

## Boot Code

The **boot.S**, **crt0.S**, and **eabi.S** files contain a minimal set of code for initializing the processor and starting an application.

### boot.S

Code in the **boot.S** consists of the two sections **boot** and **boot0**. The **boot** section contains only one instruction that is labeled with **\_boot**. During the link process, this instruction is mapped to the reset vector and the **\_boot** label marks the application's entry point. The boot instruction is a jump to the **\_boot0** label. The **\_boot0** label must reside within a  $\pm 23$ -bit address space of the **\_boot** label. It is defined in the **boot0** section. The code in the **boot0** section calculates the 32-bit address of the **\_start** label and jumps to it.

### crt0.S

Code in the **crt0.S** file starts executing at the **\_start** label. It initializes the **.sbss** and **.bss** sections to zero, as required by the ANSI C specification, sets up the stack, initializes some processor registers, and calls the **main()** function.

The program remains in an endless loop on return from **main()**.

### eabi.S

When an application is compiled and linked with the **-msdata=eabi** option, GCC inserts a call to the **\_\_eabi** label at the beginning of the **main()** function. This is the place where register R13 must be set to point to the **.sdata** and **.sbss** data sections and register R2 must be set to point to the **.sdata2** read-only data section.

Code in **eabi.S** sets these two registers to the correct values. The **\_SDA\_BASE\_** and **\_SDA2\_BASE\_** labels are generated by the linker.

## Bus Error Registers

The **bexr.c** file and corresponding **bexr.h** include file provide access to the bus error syndrome and the bus error address registers. The bus error registers hold the cause and the instruction or data address of a bus error.

### **unsigned int bexr\_BaseAdr;**

The **bexr\_BaseAdr** variable points to the base address of the bus error registers. It defaults to 0xD0000000.

### **unsigned int bexr\_Read(unsigned int adr);**

This function reads the contents of the bus error register at address **bexr\_BaseAdr+adr**.

### **void bexr\_Write(unsigned int adr, unsigned int val);**

This function writes the contents of the bus error register at address **bexr\_BaseAdr+adr**.



## Cache

The **cache.c** file and corresponding **cache.h** include file provide access to cache and cache-related operations.

### void writeCCR0(unsigned int val);

The writeCCR0() function writes an integer value to the CCR0 register. Below is a sample code sequence. Before writing to this register, the instruction cache must be enabled to prevent a lockup of the processor core. After writing the CCR0, the instruction cache can be disabled, if not needed.

```
...
enable_icache(0x80000000) /* enable instruction cache for first 128 MB
memory region */
writeCCR0(0x2700E00) /* enable 8 word pre-fetching */
disable_icache() /* disable instruction cache */
...
```

### void enable\_dcach(unsigned int regions);

The enable\_dcach() function enables the data cache for a specific memory region. Each bit in the **regions** parameter represents 128 MB of memory.

A value of 0x80000000 enables the data cache for the first 128 MB of memory (0 - 0x7FFFFFFF). A value of 0x1 enables the data cache for the last 128 MB of memory (0xF8000000 - 0xFFFFFFFF).

### void disable\_dcach(void);

The disable\_dcach() function disables the data cache for all memory regions.

### void flush\_dcach\_line(unsigned int adr);

The flush\_dcach\_line() function flushes and invalidates the data cache line that contains the address specified by the **adr** parameter. A subsequent data access to this address results in a cache miss and a cache line refill.

### void store\_dcach\_line(unsigned int adr);

The store\_dcach\_line() function stores in memory the data cache line that contains the address specified by the **adr** parameter. A subsequent data access to this address results in a cache hit if the address was already cached; otherwise, it results in a cache miss and cache line refill.

### void enable\_icach(unsigned int regions);

The enable\_icach() function enables the instruction cache for a specific memory region. Each bit in the **regions** parameter represents 128 MB of memory.

A value of 0x80000000 enables the instruction cache for the first 128 MB of memory (0 - 0x7FFFFFFF). A value of 0x1 enables the instruction cache for the last 128 MB of memory (0xF8000000 - 0xFFFFFFFF).

### void disable\_icach(void);

The disable\_icach() function disables the instruction cache for all memory regions.

### void invalidate\_icach(void);

The invalidate\_icach() function invalidates the whole instruction cache. Subsequent instructions produce cache misses and cache line refills.

## `void invalidate_icache_line(unsigned int adr);`

The `invalidate_icache_line()` function invalidates the instruction cache line that contains the address specified by the **adr** parameter. A subsequent instruction to this address produces a cache miss and a cache line refill.

## Configuration

The **config.c** file and corresponding **config.h** include file define unique settings for a specific target.

### unsigned int Processor\_Frequency;

The **Processor\_Frequency** variable is set to the clock frequency of the processor. It defaults to 300000000 (300 MHz) for V2PDK.

### unsigned int PLB\_Frequency;

The **PLB\_Frequency** variable is set to the clock frequency of the processor local bus. It defaults to 100000000 (100 MHz) for V2PDK.

### unsigned int OPB\_Frequency;

The **OPB\_Frequency** variable is set to the clock frequency of the on-chip peripheral bus. It defaults to 100000000 (100 MHz) for V2PDK.

## Exception Handling

This section documents the exception handling API that is provided in the Board Support Package. For an in-depth explanation on how exceptions and interrupts work on the PPC405, please refer to the chapter “Exceptions and Interrupts” in the *PPC User’s Manual*.

The exception handling API consists of a set of the files **vector.S**, **exception.c**, and the according header file **exception.h**.

### void exception\_Init(void);

This function sets up the interrupt vector table and registers a “do nothing” function for each exception. This function has no parameters and does not return a value.

This function must be called before registering any exception handlers or enabling any interrupts. When using the exception handler API in this function should be called at the beginning of your main() routine.

**IMPORTANT:** You need to reserve memory space for storing the vector table in your map file. The memory space must begin on a 64k boundary. The mapfile entry should look like this example:

```
.vectors :
{
 . = ALIGN(64k);
 *(.vectors)
} > zbt
```

For further information on mapfiles, please refer to the Linker documentation.

### void exception\_RegisterHandler( exception\_intVector Exception\_number, void \* exception\_handler, void \* user\_value );

This function is used to register an exception handler for a specific exception. It does not return a value. Please refer to [Table 2-2](#) for a list of parameters.

Table 2-2: Exception Handler Parameters

| Parameter Name    | Parameter Type       | Description                                                                                                                                                                                      |
|-------------------|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| exception_number  | Exception_IntVectors | Exception to which this handler should be registered. The type and the values are defined in the header file <b>exception.h</b> . Please refer to <a href="#">Table 2-3</a> for possible values. |
| exception_handler | Void *               | Pointer to the exception handling function                                                                                                                                                       |
| user_value        | Void *               | User value to be passed when the handling function is called                                                                                                                                     |

Table 2-3: Registered Exception Types and Values

| Exception Type                 | Value |
|--------------------------------|-------|
| EXCEPT_JUMP_TO_ZERO            | 0     |
| EXCEPT_MACHINE_CHECK           | 1     |
| EXCEPT_CRITICAL_INT            | 2     |
| EXCEPT_DATA_STORAGE_INT        | 3     |
| EXCEPT_INSTRUCTION_STORAGE_INT | 4     |

Table 2-3: Registered Exception Types and Values

| Exception Type                  | Value |
|---------------------------------|-------|
| EXCEPT_NON_CRITICAL_INT         | 5     |
| EXCEPT_ALIGNMENT_INT            | 6     |
| EXCEPT_PROGRAM_INT              | 7     |
| EXCEPT_FPU_UNAVAILABLE_INT      | 8     |
| EXCEPT_SYSTEM_CALL              | 9     |
| EXCEPT_APU_AVAILABLE            | 10    |
| EXCEPT_PIT_INT                  | 11    |
| EXCEPT_FIT_INT                  | 12    |
| EXCEPT_WATCHDOG_TIMER_INT       | 13    |
| EXCEPT_DATA_TLB_MISS_INT        | 14    |
| EXCEPT_INSTRUCTION_TLB_MISS_INT | 15    |
| EXCEPT_DEBUG_INT                | 16    |

The function provided as the **exception\_handler** parameter must have the following function prototype:

```
typedef void (*exceptionHandlerType)(exception_intVector vector_number, void * user_value);
```

This prototype is declared in the **exception.h** header file.

When this exception handler function is called, the parameters **vector\_number** and **user\_value** will contain the same values as you provided when you registered the handler.

### void exception\_DeregisterHandler(exception\_intVectors exception\_number)

This function is used to deregister a handler function for a given exception. For possible values of parameter **exception\_number**, please refer to [Table 2-3](#).

### void exception\_EnableInterrupts (exception\_intTypes val);

This function is used to enable Interrupts. It must be called after initializing the vector table with function `exception_Init` and registering exception handlers with function `exception_RegisterHandler`. The parameter **val** selects the type of interrupts (critical or non-critical) you want to enable. Possible values for **val** are:

Table 2-4: Possible Values of **val** for Enabling Interrupts

|                     |
|---------------------|
| EXCEPT_CRITICAL     |
| EXCEPT_NON_CRITICAL |
| EXCEPT_ALL          |

These values are defined in the **exception.h** header file.

### void exception\_DisableInterrupts (exception\_intTypes val);

This function is called to disable Interrupts. For possible values of the parameter **val** refer to [Table 2-4](#).

## Files

File support is limited to the **stdin**, **stdout**, and **stderr** streams. In such an environment, the following functions do not make much sense:

- `open()` (in **open.c**)
- `close()` (in **close.c**)
- `fstat()` (in **fstat.c**)
- `unlink()` (in **unlink.c**)
- `lseek()` (in **lseek.c**)

These files are included for completeness and because they are referenced by the C library.

### `int read(int fd, char *buf, int nbytes);`

The `read()` function in **read.c** reads one byte from the standard input by calling `UART_ReadByte()`. It blocks until a character is available since `UART_ReadByte()` is blocking. `Read()` returns the number of characters read, which in this implementation is always 1, regardless of **nbytes**.

### `int write(int fd, char *buf, int nbytes);`

The `write()` function in **write.c** writes **nbytes** to the standard output by calling `UART_WriteByte()`. It blocks until all characters have been written. `Write()` returns the number of characters written, which in this implementation is always equal to **nbytes**.

### `int select(int fd, int *readmask, int *writemask, int *excpmask, int timeout);`

The `select()` function in **select.c** checks whether characters are available on **stdin**. A **readmask != 0** on return of the function indicates that a character is available. In combination with `read()`, `select()` allows a non-blocking read of **stdin**. All parameters except **readmask** are ignored.

### `int isatty(int fd);`

The `isatty()` function in **isatty.c** reports if a file is connected to a tty. This function always returns 1, since only the **stdin**, **stdout**, and **stderr** streams are supported.

## General Purpose I/O

The **gpio.c** file and corresponding **gpio.h** include file provide access to the general purpose I/O (GPIO) registers.

**unsigned int gpio\_BaseAdr;**

The **gpio\_BaseAdr** variable points to the base address of the GPIO registers. It defaults to 0x90000000.

**unsigned int gpio\_Read(unsigned int adr);**

This function reads the contents of the GPIO register at address **gpio\_BaseAdr+adr**.

**void gpio\_Write(unsigned int adr, unsigned int val);**

This function writes the contents of the GPIO register at address **gpio\_BaseAdr+adr**.

## IIC Support

The **iic.c** file and corresponding **iic.h** include file provide access to the IIC IP core controller. The code in these files implements the basic functionality for IIC bus operations and enables an IIC device driver to easily access an IIC device on the bus. Currently only master-mode read and write operations are supported.

### unsigned int IICBaseAdr;

The **IICBaseAdr** variable points to the base address of the IIC controller. It defaults to 0xA8000000.

### void iic\_Init(unsigned int baseAdr);

The **iic\_Init()** function initializes the IIC controller at the address **baseAdr**. This function must be called before any accesses to the IIC controller on the IIC bus can be made.

### unsigned int iic\_Read(unsigned int baseAdr, unsigned char deviceAdr, unsigned char \*buf, unsigned int len);

The **iic\_Read()** function reads characters from a device on the IIC bus. The function returns the number of characters read, and it is successful if the number of characters read equals the **len** parameter. [Table 2-5](#) explains the **iic\_Read()** parameters.

Table 2-5: **iic\_Read()** Parameters

| Parameter | Definition                                                                                 |
|-----------|--------------------------------------------------------------------------------------------|
| baseAdr   | Base address of the IIC controller on which the device is attached                         |
| deviceAdr | Address of the device on the IIC bus                                                       |
| buf       | Buffer for the characters to be read (This buffer must be at least <b>len</b> bytes long.) |
| len       | Number of bytes to be read                                                                 |

### unsigned int iic\_Write(unsigned int baseAdr, unsigned int deviceAdr, unsigned char \*buf, unsigned int len);

The **iic\_Write()** function writes characters to a device on the IIC bus. This function returns the number of characters written, and it is successful if the number of characters written equals the **len** parameter. [Table 2-6](#) explains the **iic\_Write()** parameters.

Table 2-6: **iic\_Write()** Parameters

| Parameter | Definition                                                                                     |
|-----------|------------------------------------------------------------------------------------------------|
| baseAdr   | Base address of the IIC controller on which the device is attached                             |
| deviceAdr | Address of the device on the IIC bus                                                           |
| buf       | Buffer with the characters to be written (This buffer must be at least <b>len</b> bytes long.) |
| len       | Number of bytes to be written                                                                  |



## IIC AT24CXX

The **iicat24cxx.c** file and corresponding **iicat24cxx.h** include file implement a device driver for an AT24CXX EEPROM by Atmel attached to the IIC bus.

**unsigned int iicat24cxx\_Read(unsigned int baseAdr, unsigned char deviceAdr, unsigned int memAdr, unsigned char \*buf, unsigned int len);**

The **iicat24cxx\_Read()** function reads characters from an IIC AT24CXX device attached to the IIC bus. This function returns the number of characters read, and it is successful if the number of characters read equals the **len** parameter. **Table 2-7** explains the **iicat24cxx\_Read()** parameters.

**Table 2-7: iicat24cxx\_Read() Parameters**

| Parameter | Definition                                                                                 |
|-----------|--------------------------------------------------------------------------------------------|
| baseAdr   | Base address of the IIC controller on which the IIC AT24CXX device is attached             |
| deviceAdr | Address of the IIC AT24CXX device on the IIC bus                                           |
| memAdr    | Address within the IIC AT24CXX device from which to read                                   |
| buf       | Buffer for the characters to be read (This buffer must be at least <b>len</b> bytes long.) |
| len       | Number of bytes to be read                                                                 |

**unsigned int iicat24cxx\_Write(unsigned int baseAdr, unsigned char deviceAdr, unsigned int memAdr, unsigned char \*buf, unsigned int len);**

The **iicat24cxx\_Write()** function writes characters into an IIC AT24CXX device attached to the IIC bus. This function returns the number of characters written, and it is successful if the number of characters written equals the **len** parameter. **Table 2-8** explains the **iicat24cxx\_Write()** parameters.

**Table 2-8: iicat24cxx\_Write() Parameters**

| Parameter | Definition                                                                                     |
|-----------|------------------------------------------------------------------------------------------------|
| baseAdr   | Base address of the IIC controller on which the IIC AT24CXX device is attached                 |
| deviceAdr | Address of the IIC AT24CXX device on the IIC bus                                               |
| memAdr    | Address within the IIC AT24CXX device to which to write                                        |
| buf       | Buffer with the characters to be written (This buffer must be at least <b>len</b> bytes long.) |
| len       | Number of bytes to be written                                                                  |

## LCD

The **lcd.c** file and corresponding **lcd.h** include file provide access to the DMC16433 LCD by Optrex. This LCD displays up to 64 characters organized in 4 lines with 16 characters each. The LCD driver numbers the character positions from 0 to 63. Position #0 is located in the top left corner and position #63 is located in the bottom right corner.

### unsigned int lcd\_BaseAdr;

The **lcd\_BaseAdr** variable points to the base address of the LCD display controller. It defaults to 0xC0000000.

### void lcd\_Open(void);

The **lcd\_Open()** function initializes the LCD controller. This function must be called before any accesses to the LCD controller can be made.

### void lcd\_Write(unsigned char val);

The **lcd\_Write()** function writes a character at the current cursor position and advances the cursor by one. At the end of a line the cursor wraps around to the beginning of the next line. At the end of line four the cursor wraps around to the first line.

### void lcd\_ClearScreen(void);

The **lcd\_ClearScreen()** function clears the LCD and sets the cursor position to zero.

### void lcd\_SetPos(unsigned char val);

The **lcd\_SetPos()** function sets the cursor position to the value specified by the **val** parameter.

## Memory and I/O Access

The `memaccess.c` and `ioaccess.c` files and the corresponding `memaccess.h` and `ioaccess.h` include files provide access to the memory and to I/O devices mapped into the memory space. The functions of accesses to the memory space and to the I/O space are comparable. In the case of memory, access operations may be re-ordered. In the case of I/O, access operations are guaranteed to occur in order.

```
unsigned char io_rb(unsigned int adr);
unsigned char io_read_byte(unsigned int adr);
unsigned char m_rb(unsigned int adr);
unsigned char mem_read_byte(unsigned int adr);
```

Read a byte at address "adr".

```
unsigned short io_rh(unsigned int adr);
unsigned short io_read_halfword(unsigned int adr);
unsigned short m_rh(unsigned int adr);
unsigned short mem_read_halfword(unsigned int adr);
```

Read a halfword at address "adr".

```
unsigned int io_rw(unsigned int adr);
unsigned int io_read_word(unsigned int adr);
unsigned int m_rw(unsigned int adr);
unsigned int mem_read_word(unsigned int adr);
```

Read a word at address "adr".

```
void io_wb(unsigned int adr, unsigned char val);
void io_write_byte(unsigned int adr, unsigned char val);
void m_wb(unsigned int adr, unsigned char val);
void mem_write_byte(unsigned int adr, unsigned char val);
```

Write the byte "val" to the address "adr".

```
void io_wh(unsigned int adr, unsigned short val);
void io_write_halfword(unsigned int adr, unsigned short val);
void m_wh(unsigned int adr, unsigned short val);
void mem_write_halfword(unsigned int adr, unsigned short val);
```

Write the halfword "val" to the address "adr".

```
void io_ww(unsigned int adr, unsigned int val);
void io_write_word(unsigned int adr, unsigned int val);
void m_ww(unsigned int adr, unsigned int val);
void mem_write_word(unsigned int adr, unsigned int val);
```

Write the word "val" to the address "adr".

```
unsigned short io_rhr(unsigned int adr);
unsigned short io_read_halfword_reversed(unsigned int adr);
unsigned short m_rhr(unsigned int adr);
unsigned short mem_read_halfword_reversed(unsigned int adr);
```

Read a halfword from address "adr" and change its endianness.

```
unsigned int io_rwr(unsigned int adr);
unsigned int io_read_word_reversed(unsigned int adr);
unsigned int m_rwr(unsigned int adr);
unsigned int mem_read_word_reversed(unsigned int adr);
```

Read a word from address "adr" and change its endianness.

```
void io_whr(unsigned int adr, unsigned short val);
void io_write_halfword_reversed(unsigned int adr, unsigned short val);
void m_whr(unsigned int adr, unsigned short val);
void mem_write_halfword_reversed(unsigned int adr, unsigned short val);
```

Change the endianness of the halfword "val" and write the result to the address "adr".

```
void io_wwr(unsigned int adr, unsigned int val);
void io_write_word_reversed(unsigned int adr, unsigned int val);
void m_wwr(unsigned int adr, unsigned int val);
void mem_write_word_reversed(unsigned int adr, unsigned int val);
```

Change the endianness of the word "val" and write the result to the address "adr".

## Memory Management

`char *sbrk(int nbytes);`

The `sbrk()` function in the `sbrk.c` file allocates **nbytes** of heap and returns a pointer to that piece of memory. This function is called from the memory allocation functions of the C library.

## PCI

The **pci.c** file and corresponding **pci.h** include file provide access to the PCI bus and its devices.

A pair consisting of a bus number and device number uniquely identifies a PCI device. A PCI device can provide different functions, which are defined by a function number. The **busDevFunc** variable is an abbreviation for the bus, device, and function triplet.

For future portability, the `pci_in()` and `pci_out()` functions should be used whenever a value is read from or written to the PCI address space.

### **void pci\_init(void);**

The `pci_init()` function initializes the PCI controller. This function must be called before any other PCI function or a bus error occurs.

### **int pci\_find\_device(int busDevFunc, unsigned short vendorid, unsigned short deviceid);**

The `pci_find_device()` function searches the PCI bus for a device with a given pair consisting of a vendor number and device identification number. If such a device is found, the function returns the **busDevFunc** parameter. Otherwise, the function returns -1.

This function can be used to find multiple devices with the same vendor and device identification number pair. When the function is called for the first time, the **busDevFunc** parameter should be set to -1. In subsequent calls the last returned **busDevFunc** is given as the **busDevFunc** parameter. See the `pci_scan()` code example below.

### **int pci\_find\_device\_type(int busDevFunc, int classCode);**

The `pci_find_device_type()` function searches the PCI bus for a device with a given class code (for example, 0x300000 for a graphics card). If such a device is found, the function returns the **busDevFunc** parameter. Otherwise, the function returns -1.

This function can be used to find multiple devices with the same class code. When the function is called for the first time, the **busDevFunc** parameter should be set to -1. In subsequent calls the last returned **busDevFunc** is given as the **busDevFunc** parameter. See the `pci_scan()` code example below.

### **int pci\_scan(int busDevFunc);**

The `pci_scan()` function searches the PCI bus for all devices.

If a device is found, the function returns the **busDevFunc** parameter. Otherwise, the function returns -1. This function can be used to find all devices on the PCI bus. When the function is called for the first time, the **busDevFunc** parameter should be set to -1. In subsequent calls the last returned **busDevFunc** is given as the **busDevFunc** parameter.

The following source code gives an example of how to find all devices on the PCI bus.

```
int busDevFunc;

busDevFunc = -1;
while (1)
{
 busDevFunc = pci_scan(busDevFunc);
 if (busDevFunc == -1)
 break;
 dump registers of PCI device busDevFunc;
}
```

```
int pci_map(int busDevFunc, int bar, unsigned int *physAdr, unsigned int
*size);
```

The `pci_map()` function maps the memory of a device to the PCI address space.

The `busDevFunc` parameter denotes the PCI device. The `bar` parameter is either the number of one of the six base address registers in the PCI configuration space of the device or `-1` if memory is to be allocated for a legacy I/O device. On success the function returns 0, the `physAdr` parameter is set to the base address of the allocated memory, and the `size` parameter is set to the size of the allocated memory. If unsuccessful, the function returns `-1`. `pci_map()` maps memory to one of two memory regions, as shown in Table 2-9.

Table 2-9: `pci_map()` Function

| PCI Address Range       | PCI Memory Description           |
|-------------------------|----------------------------------|
| 0xB0000000 - 0xB7FFFFFF | PCI memory space (memory cycles) |
| 0xB8000000 - 0xBBFFFFFF | PCI I/O space (I/O cycles)       |

```
unsigned int pci_read_config_reg(int busDevFunc, int reg, int width);
```

The `pci_read_config_reg()` function reads a PCI device's configuration register. The **`busDevFunc`** parameter denotes the PCI device. The **`reg`** parameter specifies which register is read. Table 2-10 shows **`width`** parameter values and the function's return values.

Table 2-10: `pci_read_config_reg()` Function

| Access Type | Width | Return Value if PCI Device <code>busDevFunc</code> is: |             |
|-------------|-------|--------------------------------------------------------|-------------|
|             |       | Present                                                | Not Present |
| byte        | 1     | register value (byte)                                  | 0xFF        |
| short       | 2     | register value (short)                                 | 0xFFFF      |
| int         | 4     | register value (int)                                   | 0xFFFFFFFF  |

```
void pci_write_config_reg(int busDevFunc, int reg, unsigned int val, int
width);
```

The `pci_write_config_reg()` function writes a configuration register of a PCI device.

The **`busDevFunc`** parameter denotes the PCI device. The **`reg`** parameter specifies which register is written with the **`val`** parameter at the size of the **`width`** parameter. Writing to a non-existing PCI device results in a bus error.

```
void pci_out32(unsigned int adr, unsigned int val);
```

The `pci_out32()` function writes a 32-bit value to the PCI address space. The **`adr`** parameter points to the physical PCI address where the **`val`** parameter is written.

```
void pci_out16(unsigned int adr, unsigned short val);
```

The `pci_out16()` function writes a 16-bit value to the PCI address space. The **`adr`** parameter points to the physical PCI address where the **`val`** parameter is written.

```
void pci_out8(unsigned int adr, unsigned char val);
```

The `pci_out8()` function writes an 8-bit value to the PCI address space.

The **`adr`** parameter points to the physical PCI address where the **`val`** parameter is written.

Write an 8 bit value to the PCI address space. The parameter *adr* points to the physical PCI address where the parameter *val* is written.

### unsigned int pci\_in32(unsigned int adr);

The `pci_in32()` function reads a 32-bit value from the PCI address space. The function returns the value read from the PCI address space pointed to by the **adr** parameter.

### unsigned short pci\_in16(unsigned int adr);

The `pci_in16()` function reads a 16-bit value from the PCI address space. The function returns the value read from the PCI address space pointed to by the **adr** parameter.

### unsigned char pci\_in8(unsigned int adr);

The `pci_in8()` function reads an 8-bit value from the PCI address space. The function returns the value read from the PCI address space pointed to by the **adr** parameter.



## Process

The functions `getpid()` in `getpid.c` and `kill()` in **`kill.c`** are included for completeness and because they are referenced by the C library.

## Processor-Specific Include Files

The **reg405.h** include file contains the register numbers and the register bits for the PPC 405 processor.

The **pseudo-asm.h** include file contains the definitions for the most often used inline assembler instructions.

These inline assembler instructions can be used from drivers and user applications written in C.

## StrataFlash

The **strataflash.c** and corresponding **strataflash.h** include files provide functions to perform operations on Intel StrataFlash. The current implementation assumes that four StrataFlash chips are used in parallel. The following implementations exist:

- The block size is 512KB, four times as large as the block size of a single chip.
- The address of data to be written must be aligned on a word boundary.
- The length of the data to be written must be a multiple of a word.
- Buffer writes are not implemented.

### unsigned int strataflash\_BaseAdr;

The **strataflash\_BaseAdr** variable points to the base address of the Intel StrataFlash. It defaults to 0x18000000.

### unsigned int strataflash\_Size;

The **strataflash\_Size** variable contains the total size of the StrataFlash. It defaults to 32MB. This corresponds to four chips where each chip has a size of 8MB.

### int strataflash\_AddressToBlock(unsigned int adr, unsigned int \*block);

The **strataflash\_AddressToBlock()** function calculates the block number **block** for a specific address **adr**. The function succeeds and returns 0 if **strataflash\_BaseAdr** <= **adr** < **strataflash\_BaseAdr** + **strataflash\_Size**.

### int strataflash\_BlockToAddress(unsigned int block, unsigned int \*adr);

The **strataflash\_BlockToAddress()** function calculates the start address **adr** of a block **block**. The function succeeds and returns 0 if a block can be translated into an address, i.e. **block** < maximum number of blocks.

### int strataflash\_Write(unsigned int adr, unsigned char \*buf, unsigned int len);

The **strataflash\_Write()** function programs **len** bytes contained in the buffer **buf** to the address **adr** of the StrataFlash. Please note that due to implementation restrictions, the address must be aligned to a word, and the length must be a multiple of a word. The function returns the number of successfully written bytes.

The user must make sure that the memory is erased before it is written. Otherwise the result of a write operation might be unexpected.

### int strataflash\_EraseBlock(unsigned int block);

The **strataflash\_EraseBlock()** function erases the block with block number **block**. The function returns 0 on success.

### int strataflash\_LockBlock(unsigned int block);

The **strataflash\_LockBlock()** function locks the block with block number **block**. This disables write and erase operations on the block. The function returns 0 on success.

### int strataflash\_UnlockBlock(unsigned int block);

The **strataflash\_UnlockBlock()** function unlocks the block with block number **block**. This enables write and erase operations on the block. The function returns 0 on success.

## Time

The **mltime.c** file and corresponding **mltime.h** include file provide access to the 64-bit time base counter inside the PowerPC core. The counter increases by one at every processor cycle.

The **sleep.c** file and corresponding **sleep.h** include file implement functions for tired programs. All sleep functions are implemented as busy loops.

### typedef struct MTime;

The **MTime** struct in **mltime.h** represents the Time Base register. This struct consists of the Time Base Low (TBL) and Time Base High (TBH) registers, each of which is a 32-bit wide register. The definition of **MTime** is as follows:

```
typedef MTime
{
 unsigned int high; /* high 32 bits of Time Base register (TBH) */
 unsigned int low; /* low 32 bits of Time Base register (TBL) */
} MTime;
```

### int mltime\_compare(MTime c0, MTime c1);

The **mltime\_compare()** function in **mltime.c** compares the values of **c0** and **c1**.

**Table 2-11** shows the value returned by the function, depending on the value of **c0** and **c1**.

**Table 2-11: mltime\_compare() Function**

| Comparison Result | Function Returns Value |
|-------------------|------------------------|
| <b>c0 &lt; c1</b> | < 0                    |
| <b>c0 == c1</b>   | == 0                   |
| <b>c0 &gt; c1</b> | > 0                    |

### void mltime\_settime(MTime mltime);

The **mltime\_settime()** function in **mltime.c** sets the time base register to the value in **mltime**.

### void mltime\_gettime(MTime \*mltime);

The **mltime\_gettime()** function in **mltime.c** returns the current value of the time base register.

### void mltime\_add(MTime a0, MTime a1, MTime \*r);

The **mltime\_add()** function in **mltime.c** adds the values of **a0** and **a1** and returns the result in the **r** parameter ( $r = a0 + a1$ ).

### void mltime\_subtract(MTime s0, MTime s1, MTime \*r);

The **mltime\_subtract()** function in **mltime.c** subtracts the value of **s1** from the value of **s0** and returns the result in the **r** parameter ( $r = s0 - s1$ ).

### void mltime\_TSRClearStatusBits(unsigned long Bitmask);

The **mltime\_TSRClearStatusBits()** function in **mltime.c** is used to clear bits in the Timer Status Register (TSR). The parameter **Bitmask** designates the bits to be cleared. A 0b1 in any position of the **Bitmask** parameter clears the corresponding bit in the TSR. This function does not return a value.

The header file **reg405.h** defines the following values for the Bitmask parameter:

Table 2-12: Bitmask Parameter Values

| Name                     | Value      | Description                                                                                                                        |
|--------------------------|------------|------------------------------------------------------------------------------------------------------------------------------------|
| TSR_PIT_INTERRUPT_STATUS | 0x08000000 | Clears the Programmable Interrupt Timer Status bit. This bit is set after a PIT interrupt has occurred.                            |
| TSR_CLEAR_ALL            | 0xFFFFFFFF | Clears all bits in the TSR. After a Reset, the content of the TSR is not specified. Use this Bitmask to clear all bits in the TSR. |

Example:

```
mltime_TSRClearStatusBits(TSR_CLEAR_ALL);
```

### void mltime\_PITSetInterval(unsigned long interval);

The `mltime_PITSetInterval()` function in **mltime.c** is used to load a new value into the Programmable-Interval Timer Register. This register is a 32-bit decrementing counter clocked at the same frequency as the time-base register. Depending on the `AutoReload` setting the PIT is automatically reloaded with the last written value or has to be reloaded manually. This function does not return a value.

Example:

```
mltime_PITSetInterval(0x00ffffff);
```

### void mltime\_PITEnableInterrupt(void);

The `mltime_PITEnableInterrupt()` function in **mltime.c** enables the generation of PIT interrupts. An interrupt occurs when the PIT register contains a value of 1, and is then decremented. This function does not return a value.

Example:

```
mltime_PITEnableInterrupt();
```

### void mltime\_PITDisableInterrupt(void);

The `mltime_PITDisableInterrupt()` function in **mltime.c** disables the generation of PIT interrupts. It does not return a value.

Example:

```
mltime_PITDisableInterrupt();
```

### void mltime\_PITEnableAutoReload(void);

The `mltime_PITEnableAutoReload()` function in **mltime.c** enables the auto-reload function of the PIT Register. When auto-reload is enabled the PIT Register is automatically reloaded with the last value loaded by calling the **mltime\_PITSetInterval** function when the PIT Register contains a value of 1 and is decremented. When auto-reload is enabled, the PIT Register never contains a value of 0. This function does not return a value.

Example:

```
mltime_PITEnableAutoReload();
```

## void mltime\_PITDisableAutoReload(void);

The mltime\_PITDisableAutoReload() function in mltime.c disables the auto-reload feature of the PIT Register. When auto-reload is disabled the PIT decrements from 1 to 0. If it contains a value of 0 it stops decrementing until it is loaded with a non-zero value. This function does not return a value.

Example:

```
mltime_PITDisableAutoReload();
```

## void mltime\_PITClearInterrupt(void);

The mltime\_PITClearInterrupt() function in mltime.c is used to clear PIT-Interrupt-Status bit in the Timer-Status Register. This bit specifies whether a PIT interrupt occurred. You must call this function in your interrupt-handler to clear the Status bit other another PIT interrupt will occur immediately after exiting the interrupt -handler function. This function does not return a value.

Example:

```
mltime_PITClearInterrupt();
```

## unsigned int usleep(unsigned int \_\_useconds);

The usleep() function in **sleep.c** delays the execution of a program by **\_\_useconds** microseconds. It always returns zero.

## unsigned int sleep(unsigned int \_\_seconds);

The sleep() function in **sleep.c** delays the execution of a program by **\_\_seconds** seconds. It always returns zero.

## int nanosleep(const struct timespec \*rqtp, struct timespec \*rmtp);

The nanosleep() function in **sleep.c** is currently not implemented. It is a placeholder for linking applications against the C library. It always returns zero.

## UART

The **uart.c** file and corresponding **uart.h** include file provide access to a UART. The current implementation supports multiple UARTs on the OPB.

### unsigned int Uart\_Frequency;

The **Uart\_Frequency** variable must be set to the clock frequency of the UART. It defaults to the value of **OPB\_Frequency**.

For initialization reasons this variable is instantiated in **config.c** even if it logically belongs to **uart.c**. It is defined in **uart.h**.

### unsigned int Uart1BaseAdr;

The **Uart1BaseAdr** variable points to the base address of the first UART. It defaults to 0xA0001003.

### unsigned int Uart2BaseAdr;

The **Uart2BaseAdr** variable points to the base address of the second UART. It defaults to 0xA0011003.

### void UART\_Init(unsigned int baseAdr, int baud, char data, char parity, char stop);

The **UART\_Init()** function initializes the UART at **baseAdr** with the given parameters. **Table 2-13** explains the meaning of this function's parameters.

**Table 2-13: Parameters for Initializing the UART at REGBASE**

| Parameter | Definition                                                                                                                                                        |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Baud      | Baud Rate: arbitrary baud rates can be programmed                                                                                                                 |
| Data      | Number of data bits                                                                                                                                               |
| Parity    | Method for parity checking:<br>PARNO: no parity<br>PARODD: odd parity<br>PAREVEN: even parity<br>PARMARK: mark parity checking<br>PARSPACE: space parity checking |
| Stop      | Number of stop bits:<br>STOP1: 1 stop bit<br>STOP1DOT5: 1.5 stop bits<br>STOP2: 2 stop bits                                                                       |

### unsigned char UART\_Read(unsigned int baseAdr);

The **UART\_Read()** function returns one character read from the UART at **baseAdr**. The function blocks until a character is available.

### void UART\_Write(unsigned int baseAdr, unsigned char c);

The **UART\_Write()** function writes the character **c** to the UART at **baseAdr**. The function blocks until the character has been written.

```
void UART_Select(unsigned int baseAdr, int *readmask, int *writemask,
int *excpmask);
```

The UART\_Select() function implements the select functionality for file accesses in the case of a UART. See the select() function under **Files** for more information.





# *PPC405 Software: Demonstration and Test Applications*

---

## Overview

The sub-directories of `$V2PRO/source/sw/apps` contain software demonstration and test applications to

- introduce the user to the technology of the Virtex-II Pro™ FPGA family in regard of how to design, write, and build custom software applications
- demonstrate and test the functionality of the hardware
- demonstrate and test software tools and libraries

The demonstration and test applications introduce the user to the technology of writing custom software applications for the Virtex-II Pro FPGA family with its embedded PowerPC® 405 core. They are intentionally kept simple to allow the user to quickly understand, follow and adapt to the software development process. As a consequence all demonstration and test applications are stand-alone, i.e. they directly run on top of the hardware and no operating system is used.

On the hardware side, the applications demonstrate and test the functionality of the embedded PowerPC 405 core, the memories, and the peripheral devices. On the PowerPC 405 core instructions are executed, internal general purpose and special registers are accessed, instruction and data caches are enabled and disabled and different bus transactions are initiated. Instructions and data are read from and written to block RAM (BRAM), double data rate memory (DDR), static RAM (SRAM), and on-chip memory (OCM). For the peripheral devices the functions of a 16x50 compatible UART, an IIC controller with an attached IIC EEPROM device, a LCD controller, a PCI controller, and a general purpose IO (GPIO) controller are used.

On the software side, the applications demonstrate and test the functionality of the board support package (BSP), the newlib libraries, and the GNU software tools. The BSP contains drivers and software interfaces to the hardware. It is used by stand-alone applications and by the newlib libraries. The newlib libraries offer standard ANSI C functionality for embedded designs. The GNU software tools are used to build applications suited for the reference designs.

**Table 3-1** and **Table 3-2** give an overview of the different demonstration and test applications, and which hardware and software they demonstrate and test. Hardware that is not used in one of the applications is omitted from **Table 3-1**. BSP source files which the applications use only through the C library, not directly, are omitted from **Table 3-2**.

The applications `hello_gpio`, `hello_iic_at24cxx`, `hello_lcd`, and `hello_uart` are designed to demonstrate and test the hardware of a single peripheral device and its device driver in the simplest possible way. The `hello` application is similar to the `hello_uart` application but links against the newlib C library instead of directly using the low-level functions in the

BSP. The scan\_pci application is also linked against the C library. The xrom application demonstrates and tests a whole system like the embedded PPC405 reference design.

Table 3-1: Hardware Used with Various Demo and Test Applications

| Hardware Used:                        | Applications           |                      |                           |                                  |                          |                          |                           |                         |                     |
|---------------------------------------|------------------------|----------------------|---------------------------|----------------------------------|--------------------------|--------------------------|---------------------------|-------------------------|---------------------|
|                                       | flasher <sup>(1)</sup> | hello <sup>(1)</sup> | hello_gpio <sup>(1)</sup> | hello_iic_at24cxx <sup>(1)</sup> | hello_lcd <sup>(1)</sup> | hello_pit <sup>(1)</sup> | hello_uart <sup>(2)</sup> | scan_pci <sup>(1)</sup> | xrom <sup>(1)</sup> |
| GPIO                                  |                        |                      | X                         |                                  |                          |                          |                           |                         | X                   |
| IIC controller                        |                        |                      |                           | X                                |                          |                          |                           |                         | X                   |
| Atmel IIC EEPROM AT24C64              |                        |                      |                           | X                                |                          |                          |                           |                         | X                   |
| LCD controller                        |                        |                      |                           |                                  | X                        | X                        |                           |                         | X                   |
| PCI controller                        |                        |                      |                           |                                  |                          |                          |                           | X                       | X                   |
| UART                                  | X                      | X                    |                           |                                  |                          | X                        | X                         | X                       | X                   |
| Instruction Cache                     |                        |                      |                           |                                  |                          |                          |                           |                         | X                   |
| Data Cache                            |                        |                      |                           |                                  |                          |                          |                           |                         | X                   |
| Time Base Register                    |                        |                      | X <sup>(3)</sup>          |                                  | X <sup>(3)</sup>         |                          |                           |                         | X <sup>(3)</sup>    |
| Programmable Interrupt Timer          |                        |                      |                           |                                  |                          | X                        |                           |                         |                     |
| Bus Error Registers                   |                        |                      |                           |                                  |                          |                          |                           |                         | X                   |
| DDR SDRAM [0x00000000 ... 0x03FFFFFF] |                        |                      |                           |                                  |                          |                          |                           |                         | X                   |
| SRAM [0x10000000 ... 0x101FFFFF]      |                        |                      |                           |                                  |                          |                          |                           |                         | X                   |
| FLASH [0x18000000 ... 0x19FFFFFF]     | X                      |                      |                           |                                  |                          |                          |                           |                         | X                   |
| ZBT [0x30000000 ... 0x300FFFFFF]      |                        |                      |                           |                                  |                          |                          |                           |                         | X                   |
| BRAM [0xFFFF0000 ... 0xFFFF8000]      | X                      | X                    | X                         | X                                | X                        | X                        | X                         | X                       | X                   |

**Notes:**

1. Simulates in the embedded PPC405 reference design
2. Simulates in the simple and embedded PPC405 reference designs
3. Only used when SPEED\_UP is not specified during build of application

Table 3-2: Software Used with Various Demo and Test Applications

| Software Used:     | Applications           |                      |                           |                                  |                          |                          |                           |                         |                     |
|--------------------|------------------------|----------------------|---------------------------|----------------------------------|--------------------------|--------------------------|---------------------------|-------------------------|---------------------|
|                    | flasher <sup>(1)</sup> | hello <sup>(1)</sup> | hello_gpio <sup>(1)</sup> | hello_iic_at24cxx <sup>(1)</sup> | hello_lcd <sup>(1)</sup> | hello_pit <sup>(1)</sup> | hello_uart <sup>(2)</sup> | scan_pci <sup>(1)</sup> | xrom <sup>(1)</sup> |
| BSP/config.c       | X                      | X                    | X                         | X                                | X                        | X                        | X                         | X                       | X                   |
| BSP/boot.S         | X                      | X                    | X                         | X                                | X                        | X                        | X                         | X                       | X                   |
| BSP/crt0.S         | X                      | X                    | X                         | X                                | X                        | X                        | X                         | X                       | X                   |
| BSP/eabi.S         | X                      | X                    | X                         | X                                | X                        | X                        | X                         | X                       | X                   |
| BSP/gpio.c         |                        |                      | X                         |                                  |                          |                          |                           |                         | X                   |
| BSP/iic.c          |                        |                      |                           | X                                |                          |                          |                           |                         | X                   |
| BSP/iicat24cxx.c   |                        |                      |                           | X                                |                          |                          |                           |                         | X                   |
| BSP/lcd.c          |                        |                      |                           |                                  | X                        | X                        |                           |                         | X                   |
| BSP/pci.c          |                        |                      |                           |                                  |                          |                          |                           | X                       | X                   |
| BSP/uart.c         | X                      | X                    |                           |                                  |                          | X                        | X                         | X                       | X                   |
| BSP/cache.c        |                        |                      |                           |                                  |                          |                          |                           |                         | X                   |
| BSP/sleep.c        |                        |                      | X <sup>(3)</sup>          |                                  | X <sup>(3)</sup>         |                          |                           |                         | X <sup>(3)</sup>    |
| BSP/mltime.c       |                        |                      |                           |                                  | X                        | X                        |                           |                         | X                   |
| BSP/bexr.c         |                        |                      |                           |                                  |                          |                          |                           |                         | X                   |
| BSP/exception.c    |                        |                      |                           |                                  |                          | X                        |                           |                         |                     |
| BSP/strataflash.c  | X                      |                      |                           |                                  |                          |                          |                           |                         | X                   |
| C library          |                        | X                    |                           |                                  |                          |                          |                           | X                       |                     |
| Terminal           | X                      | X                    |                           |                                  |                          | X                        | X                         | X                       | X                   |
| GNU Software Tools | X                      | X                    | X                         | X                                | X                        | X                        | X                         | X                       | X                   |

**Notes:**

1. Simulates in the embedded PPC405 reference design
2. Simulates in the simple and embedded PPC405 reference designs
3. Only used when SPEED\_UP is not specified during build of application

## Related Documentation

- *Simple PPC405 Reference System*
- *PPC405 Packet Processing Engine Reference System Using Rocket I/O*
- *Embedded PPC405 Reference System*
- *Introduction to GNU Software*
- *Stand-Alone Board Support Package*
- *Simulation Terminal*

## flasher

### Description

The flasher application tests and demonstrates the use of the Strataflash API (see *Stand-Alone Board Support Package*). The application communicates through the UART. It prints a greeting message and reads in a datafile in the **S-Record Format**. This datafile is then written to the FLASH memory. After that the application reads back the contents of the FLASH and verifies it against the original data.

### S-Record Format

S-Records are an industry-standard format created by Motorola for transmitting binary files to target systems and PROM programmers. Documentation on the S-Record format can be found in various sources on the internet. You can use *objcopy* to convert a binary file to S-Record:

Example:

```
$ powerpc-eabi-objcopy -O srec hello.elf hello.srec
```

NOTES:

- The address field in each S-Record line must be word aligned.
- The “size” field in each S-Record line must be word-aligned.
- The maximum size of the datafile is restricted to 128KB. (This can be changed in the flasher.c source code.)

### Usage

**Table 3-3** lists with which reference designs the flasher demonstration and test application can be used.

**Table 3-3: Reference Designs Usable with the *flasher* Application**

| Reference Design         | Usable |
|--------------------------|--------|
| Simple PPC405            | No     |
| Embedded PPC405          | Yes    |
| Packet Processing Engine | No     |

### Files

**Table 3-4: Files Used in the *flasher* Application**

| File                      | Description                                        |
|---------------------------|----------------------------------------------------|
| README                    | Brief description of the application.              |
| Makefile                  | Makefile to build the application.                 |
| hello.srec                | Example S-Record file created from hello.elf       |
| flasher.c                 | File containing the main program.                  |
| v2pdk/debug/flasher.o     | Object file after build with debug options.        |
| v2pdk/release/flasher.o   | Object file after build with optimization options. |
| v2pdk/debug/flasher.elf   | Executable file with debug options.                |
| v2pdk/release/flasher.elf | Executable file with optimization options.         |

## Building for V2PDK

The file Makefile contains the target rules to build the debug and release versions of flasher. **Table 3-5** lists the possible targets for V2PDK. To build a specific target, execute:

```
$ make <target>
```

**Table 3-5: V2PDK Possible Targets**

| Target (<target>)   | Description                                                                                     |
|---------------------|-------------------------------------------------------------------------------------------------|
| hello_debug_v2pdk   | Builds executable with debug options (-gdwarf). Links against the debug version of the BSP      |
| hello_release_v2pdk | Builds executable with optimization options (-O3). Links against the release version of the BSP |
| clean               | Removes object files and executables of the debug and release versions.                         |
| megaclean           | Calls target clean and removes temporary files.                                                 |
| BUILD-V2PDK         | Calls the targets megaclean and hello_release_v2pdk.                                            |

Some definitions added with “-D” to the CCFLAGS in the “define compiler settings” section of the Makefile influence the build process, as listed in **Table 3-6**.

**Table 3-6: Optionally Added Definitions to CFLAGS**

| Define   | Description                                                  |
|----------|--------------------------------------------------------------|
| SPEED_UP | Speed up for simulation – i.e., do not call “sleep function” |

**Table 3-7** lists which libraries are required for building the application.

**Table 3-7: Required Libraries**

| Library | Description           |
|---------|-----------------------|
| BSP     | Board Support Package |

# hello

## Description

The hello application tests and demonstrates the hardware and software for a 16x50 compatible UART. It initializes the UART with the baud rate, the number of data bits, the desired parity, and the number of stop bits. It then prints the string “Hello world\r\n” to the UART. After that, it waits for characters from the UART and echoes them. The escape character (ESC, 0x1b, 27d) terminates the application.

The application is similar to the hello\_uart application but uses the newlib C library, and the stdin and stdout streams to communicate through the UART.

**NOTE:** In this implementation the UART is polled. Typing ahead more than one character while the “Hello world” string is printed results in the loss of characters. Characters are only echoed correctly after the entire string is displayed.

## Usage

Table 3-8 lists with which reference designs the hello demonstration and test application can be used.

Table 3-8: Reference Designs Usable with the *hello* Application

| Reference Design         | Usable |
|--------------------------|--------|
| Simple PPC405            | No     |
| Embedded PPC405          | Yes    |
| Packet Processing Engine | No     |

## Files

Table 3-9: Files Used in the *hello* Application

| File                    | Description                                          |
|-------------------------|------------------------------------------------------|
| README                  | Brief description of the application.                |
| Makefile                | Makefile to build the application.                   |
| hello.c                 | File containing the main program.                    |
| v2pdk/debug/hello.o     | Object file after a build with debug options.        |
| v2pdk/release/hello.o   | Object file after a build with optimization options. |
| v2pdk/debug/hello.elf   | Executable after a build with debug options.         |
| v2pdk/release/hello.elf | Executable after a build with optimization options.  |

## Building for V2PDK

The file Makefile contains the target rules to build the debug and release versions of hello. Table 3-10 lists the possible targets for V2PDK. To build a specific target execute:

```
$ make <target>
```

Table 3-10: V2PDK Possible Targets

| Target (<target>)   | Description                                                                                      |
|---------------------|--------------------------------------------------------------------------------------------------|
| hello_debug_v2pdk   | Builds executable with debug options (-gdwarf). Links against the debug version of the BSP.      |
| hello_release_v2pdk | Builds executable with optimization options (-O3). Links against the release version of the BSP. |
| clean               | Removes object files and executables of the debug and release version                            |
| megaclean           | Calls target clean and removes temporary files                                                   |
| BUILD-V2PDK         | Calls the targets megaclean and hello_release_v2pdk                                              |

For V2PDK, the baud rate (2.083333 Mbaud) is calculated so that the divisor latch inside the UART is set to three. This speeds up the simulation significantly.

Table 3-11 lists which libraries are required for building the application.

Table 3-11: Required Libraries

| Library | Description           |
|---------|-----------------------|
| BSP     | Board Support Package |
| libC    | C library             |



## hello\_gpio

### Description

The hello\_gpio application tests and demonstrates the hardware and software for the general purpose IO (GPIO). It enables all pins of the GPIO for output and sequentially writes all the characters of the “Hello world\r\n” string to the 8 least significant bits of the GPIO.

### Usage

Table 3-12 lists with which reference designs the hello\_gpio demonstration and test application can be used.

Table 3-12: Reference Designs Usable with the *hello\_gpio* Application

| Reference Design         | Usable |
|--------------------------|--------|
| Simple PPC405            | No     |
| Embedded PPC405          | Yes    |
| Packet Processing Engine | No     |

### Files

Table 3-13: Files Used in the *hello\_gpio* Application

| File                    | Description                                          |
|-------------------------|------------------------------------------------------|
| README                  | Brief description of the application.                |
| Makefile                | Makefile to build the application.                   |
| hello.c                 | File containing the main program.                    |
| v2pdk/debug/hello.o     | Object file after a build with debug options.        |
| v2pdk/release/hello.o   | Object file after a build with optimization options. |
| v2pdk/debug/hello.elf   | Executable after a build with debug options.         |
| v2pdk/release/hello.elf | Executable after a build with optimization options.  |

### Building for V2PDK

The file Makefile contains the target rules to build the debug and release versions of hello\_gpio. Table 3-14 lists the possible targets for V2PDK. To build a specific target, execute:

```
$ make <target>
```

Table 3-14: V2PDK Possible Targets

| Target (<target>)   | Description                                                                                      |
|---------------------|--------------------------------------------------------------------------------------------------|
| hello_debug_v2pdk   | Builds executable with debug options (-gdwarf). Links against the debug version of the BSP.      |
| hello_release_v2pdk | Builds executable with optimization options (-O3). Links against the release version of the BSP. |

Table 3-14: V2PDK Possible Targets (Continued)

| Target (<target>) | Description                                                           |
|-------------------|-----------------------------------------------------------------------|
| clean             | Removes object files and executables of the debug and release version |
| megaclean         | Calls target clean and removes temporary files                        |
| BUILD-V2PDK       | Calls the targets megaclean and hello_release_v2pdk                   |

Some definitions added with “-D” to the CFLAGS in the “define compiler settings” section of the Makefile influence the build process, as listed in [Table 3-15](#).

Table 3-15: Optionally Added Definitions to CFLAGS

| Define   | Description                                                 |
|----------|-------------------------------------------------------------|
| SPEED_UP | Speed up for simulation—i.e., do not call “sleep” functions |

[Table 3-16](#) lists which libraries are required for building the application.

Table 3-16: Required Libraries

| Library | Description           |
|---------|-----------------------|
| BSP     | Board Support Package |

## hello\_iic\_at24cxx

### Description

The hello\_iic\_at24cxx application tests and demonstrates the hardware and software for the 2-wire serial EEPROM AT24C64 by Atmel. The AT24C64 is attached to the IIC bus. Data is read or written to the EEPROM through the IIC controller. The application initializes the IIC controller, writes the string “Hello world\0” to address zero of the AT24C64, reads twelve characters back, and compares them against the written string.

The source code of the program can be changed to run more than one iteration. Each subsequent iteration increments the memory address where the string is written and read by one.

### Usage

Table 3-17 lists with which reference designs the hello\_iic\_at24cxx demonstration and test application can be used.

Table 3-17: Reference Designs Usable with the *hello\_iic\_at24cxx* Application

| Reference Design         | Usable |
|--------------------------|--------|
| Simple PPC405            | No     |
| Embedded PPC405          | Yes    |
| Packet Processing Engine | No     |

### Files

Table 3-18: Files Used in the *hello\_iic\_at24cxx* Application

| File                    | Description                                          |
|-------------------------|------------------------------------------------------|
| README                  | Brief description of the application.                |
| Makefile                | Makefile to build the application.                   |
| hello.c                 | File containing the main program.                    |
| v2pdk/debug/hello.o     | Object file after a build with debug options.        |
| v2pdk/release/hello.o   | Object file after a build with optimization options. |
| v2pdk/debug/hello.elf   | Executable after a build with debug options.         |
| v2pdk/release/hello.elf | Executable after a build with optimization options.  |

### Building for V2PDK

The file Makefile contains the target rules to build the debug and release versions of hello\_iic\_at24xx. Table 3-19 lists the possible targets for V2PDK. To build a specific target, execute:

```
$ make <target>
```

Table 3-19: V2PDK Possible Targets

| Target (<target>)   | Description                                                                                      |
|---------------------|--------------------------------------------------------------------------------------------------|
| hello_debug_v2pdk   | Builds executable with debug options (-gdwarf). Links against the debug version of the BSP.      |
| hello_release_v2pdk | Builds executable with optimization options (-O3). Links against the release version of the BSP. |
| clean               | Removes object files and executables of the debug and release version                            |
| megaclean           | Calls target clean and removes temporary files                                                   |
| BUILD-V2PDK         | Calls the targets megaclean and hello_release_v2pdk                                              |

Table 3-20 lists which libraries are required for building the application.

Table 3-20: Required Libraries

| Library | Description           |
|---------|-----------------------|
| BSP     | Board Support Package |

## hello\_lcd

### Description

The hello\_lcd application tests and demonstrates the hardware and software for the **DMC16433 LCD** by Optrex. It enables the LCD, clears the screen, and writes the string “Hello world\r\n” to the upper left corner of the screen.

The source code of the program can be changed to run more than one iteration. Each subsequent iteration increments the start position of the string by one.

### Usage

**Table 3-21** lists with which reference designs the hello\_lcd demonstration and test application can be used.

**Table 3-21: Reference Designs Usable with the hello\_lcd Application**

| Reference Design         | Usable |
|--------------------------|--------|
| Simple PPC405            | No     |
| Embedded PPC405          | Yes    |
| Packet Processing Engine | No     |

### Files

**Table 3-22: Files Used in the hello\_lcd Application**

| File                    | Description                                          |
|-------------------------|------------------------------------------------------|
| README                  | Brief description of the application.                |
| Makefile                | Makefile to build the application.                   |
| hello.c                 | File containing the main program.                    |
| v2pdk/debug/hello.o     | Object file after a build with debug options.        |
| v2pdk/release/hello.o   | Object file after a build with optimization options. |
| v2pdk/debug/hello.elf   | Executable after a build with debug options.         |
| v2pdk/release/hello.elf | Executable after a build with optimization options.  |

### Building for V2PDK

The file Makefile contains the target rules to build the debug and release versions of hello\_lcd. **Table 3-23** lists the possible targets for V2PDK. To build a specific target, execute:

```
$ make <target>
```

**Table 3-23: V2PDK Possible Targets**

| Target (<target>)   | Description                                                                                      |
|---------------------|--------------------------------------------------------------------------------------------------|
| hello_debug_v2pdk   | Builds executable with debug options (-gdwarf). Links against the debug version of the BSP.      |
| hello_release_v2pdk | Builds executable with optimization options (-O3). Links against the release version of the BSP. |

Table 3-23: V2PDK Possible Targets (Continued)

| Target (<target>) | Description                                                           |
|-------------------|-----------------------------------------------------------------------|
| clean             | Removes object files and executables of the debug and release version |
| megaclean         | Calls target clean and removes temporary files                        |
| BUILD-V2PDK       | Calls the targets megaclean and hello_release_v2pdk                   |

Some definitions added with “-D” to the CFLAGS in the “define compiler settings” section of the Makefile influence the build process, as listed in [Table 3-24](#).

Table 3-24: Optionally Added Definitions to CFLAGS

| Define   | Description                                                 |
|----------|-------------------------------------------------------------|
| SPEED_UP | Speed up for simulation—i.e., do not call “sleep” functions |

[Table 3-25](#) lists which libraries are required for building the application.

Table 3-25: Required Libraries

| Library | Description           |
|---------|-----------------------|
| BSP     | Board Support Package |

## hello\_pit

### Description

The hello\_pit application tests and demonstrates the use of the Exception Handling API (see [Volume 8, Chapter 2, Stand-Alone Board Support Package](#)) and the PowerPC-405 Programmable Interrupt Timer (PIT). The application is a combination of the [hello\\_lcd](#) and the [hello\\_uart](#) programs. The function for writing to the LCD is registered as an exception handler for the PIT while the UART is continuously polled.

### Usage

[Table 3-26](#) lists with which reference designs the hello\_pit demonstration and test application can be used.

**Table 3-26: Reference Designs Usable with the hello\_pit Application**

| Reference Design         | Usable |
|--------------------------|--------|
| Simple PPC405            | No     |
| Embedded PPC405          | Yes    |
| Packet Processing Engine | No     |

### Files

**Table 3-27: Files Used in the hello\_pit Application**

| File                    | Description                                        |
|-------------------------|----------------------------------------------------|
| README                  | Brief description of the application.              |
| Makefile                | Makefile to build the application.                 |
| hello.c                 | File containing the main program.                  |
| v2pdk/debug/hello.o     | Object file after build with debug options.        |
| v2pdk/release/hello.o   | Object file after build with optimization options. |
| v2pdk/debug/hello.elf   | Executable file with debug options.                |
| v2pdk/release/hello.elf | Executable file with optimization options.         |

### Building for V2PDK

The file Makefile contains the target rules to build the debug and release versions of hello\_pit. [Table 3-28](#) lists the possible targets for V2PDK. To build a specific target, execute:

```
$ make <target>
```

**Table 3-28: V2PDK Possible Targets**

| Target (<target>)   | Description                                                                                     |
|---------------------|-------------------------------------------------------------------------------------------------|
| hello_debug_v2pdk   | Builds executable with debug options (-gdwarf). Links against the debug version of the BSP      |
| hello_release_v2pdk | Builds executable with optimization options (-O3). Links against the release version of the BSP |

Table 3-28: V2PDK Possible Targets (Continued)

| Target (<target>) | Description                                                             |
|-------------------|-------------------------------------------------------------------------|
| clean             | Removes object files and executables of the debug and release versions. |
| megaclean         | Calls target clean and removes temporary files.                         |
| BUILD-V2PDK       | Calls the targets megaclean and hello_release_v2pdk.                    |

Some definitions added with “-D” to the CCFLAGS in the “define compiler settings” section of the Makefile influence the build process, as listed in [Table 3-29](#).

Table 3-29: Optionally Added Definitions to CFLAGS

| Define   | Description                                                  |
|----------|--------------------------------------------------------------|
| SPEED_UP | Speed up for simulation – i.e., do not call “sleep function” |

[Table 3-30](#) lists which libraries are required for building the application.

Table 3-30: Required Libraries

| Library | Description           |
|---------|-----------------------|
| BSP     | Board Support Package |



## hello\_uart

### Description

The hello\_uart application tests and demonstrates the hardware and software for a 16x50 compatible UART. It initializes the UART with the baud rate, the number of data bits, the desired parity, and the number of stop bits. It then prints the string "Hello world\r\n" to the UART. After that, it waits for characters from the UART and echoes them. The escape character (ESC, 0x1b, 27d) terminates the application.

**NOTE:** In this implementation the UART is polled. Typing ahead more than one character while the "Hello world" string is printed results in the loss of characters. Characters are only echoed correctly after the entire string is displayed.

### Usage

Table 3-31 lists with which reference designs the hello\_uart demonstration and test application can be used.

Table 3-31: Reference Designs Usable with the hello\_uart Application

| Reference Design         | Usable |
|--------------------------|--------|
| Simple PPC405            | Yes    |
| Embedded PPC405          | Yes    |
| Packet Processing Engine | No     |

### Files

Table 3-32: Files Used in the hello\_uart Application

| File                    | Description                                          |
|-------------------------|------------------------------------------------------|
| README                  | Brief description of the application.                |
| Makefile                | Makefile to build the application.                   |
| hello.c                 | File containing the main program.                    |
| v2pdk/debug/hello.o     | Object file after a build with debug options.        |
| v2pdk/release/hello.o   | Object file after a build with optimization options. |
| v2pdk/debug/hello.elf   | Executable after a build with debug options.         |
| v2pdk/release/hello.elf | Executable after a build with optimization options.  |

### Building for V2PDK

The file Makefile contains the target rules to build the debug and release versions of hello\_uart. Table 3-33 lists the possible targets for V2PDK. To build a specific target, execute:

```
$ make <target>
```

Table 3-33: V2PDK Possible Targets

| Target (<target>)   | Description                                                                                      |
|---------------------|--------------------------------------------------------------------------------------------------|
| hello_debug_v2pdk   | Builds executable with debug options (-gdwarf). Links against the debug version of the BSP.      |
| hello_release_v2pdk | Builds executable with optimization options (-O3). Links against the release version of the BSP. |
| clean               | Removes object files and executables of the debug and release version                            |
| megaclean           | Calls target clean and removes temporary files                                                   |
| BUILD-V2PDK         | Calls the targets megaclean and hello_release_v2pdk                                              |

For V2PDK, the baud rate (2.083333 Mbaud) is calculated so that the divisor latch inside the UART is set to three. This speeds up the simulation significantly.

Table 3-34 lists which libraries are required for building the application.

Table 3-34: Required Libraries

| Library | Description           |
|---------|-----------------------|
| BSP     | Board Support Package |

## scan\_pci

### Description

The scan\_pci application tests and demonstrates the hardware and software for the PCI controller and PCI devices on the PCI bus. It enables the PCI controller and then scans the PCI bus for devices. It lists all registers of the PCI configuration space for each found device and reports the number of devices on the bus on the terminal.

### Usage

Table 3-35 lists with which reference designs the scan\_pci demonstration and test application can be used.

Table 3-35: Reference Designs Usable with the scan\_pci Application

| Reference Design         | Usable |
|--------------------------|--------|
| Simple PPC405            | No     |
| Embedded PPC405          | Yes    |
| Packet Processing Engine | No     |

### Files

Table 3-36: Files Used in the scan\_pci Application

| File                       | Description                                          |
|----------------------------|------------------------------------------------------|
| README                     | Brief description of the application.                |
| Makefile                   | Makefile to build the application.                   |
| scan_pci.c                 | File containing the main program.                    |
| v2pdk/debug/scan_pci.o     | Object file after a build with debug options.        |
| v2pdk/release/scan_pci.o   | Object file after a build with optimization options. |
| v2pdk/debug/scan_pci.elf   | Executable after a build with debug options.         |
| v2pdk/release/scan_pci.elf | Executable after a build with optimization options.  |

## Building for V2PDK

The file Makefile contains the target rules to build the debug and release versions of scan\_pci. **Table 3-37** lists the possible targets for V2PDK. To build a specific target, execute:

```
$ make <target>
```

**Table 3-37: V2PDK Possible Targets**

| Target (<target>)      | Description                                                                                      |
|------------------------|--------------------------------------------------------------------------------------------------|
| scan_pci_debug_v2pdk   | Builds executable with debug options (-gdwarf). Links against the debug version of the BSP.      |
| scan_pci_release_v2pdk | Builds executable with optimization options (-O3). Links against the release version of the BSP. |
| clean                  | Removes object files and executables of the debug and release version                            |
| megaclean              | Calls target clean and removes temporary files                                                   |
| BUILD-V2PDK            | Calls the targets megaclean and scan_pci_release_v2pdk                                           |

**Table 3-38** lists which libraries are required for building the application.

**Table 3-38: Required Libraries**

| Library | Description           |
|---------|-----------------------|
| BSP     | Board Support Package |
| libC    | C library             |

## xrom

### Description

The xrom application demonstrates and tests the hardware and software of a whole system like the embedded PPC 405 reference design. It initializes all controllers in the system and then displays a menu. The user interacts with the application through the Simulation Terminal. The application never exits, but the simulation can be terminated by sending the escape character (ESC, 0x1b, 27d).

### Usage

Table 3-39 lists with which reference designs the xrom demonstration and test application can be used.

Table 3-39: Reference Designs Usable with the xrom Application

| Reference Design         | Usable |
|--------------------------|--------|
| Simple PPC405            | No     |
| Embedded PPC405          | Yes    |
| Packet Processing Engine | No     |

### Menu Structure

The menu structure is two levels deep. All sub-menus can be reached from the Main menu.

#### Main Menu

The Main menu offers two options that are shown and described in Table 3-40.

Table 3-40: Main Menu

| Menu Points       | Description                         |
|-------------------|-------------------------------------|
| 1. Tests          | Calls the 'Tests' sub-menu.         |
| 9. Cache settings | Calls the 'Cache settings' sub-menu |

#### Main.Tests Menu

The Main.Tests menu starts different tests on the system. The menu options are described in Table 3-41.

Table 3-41: Main.Tests Menu

| Menu Points         | Description                        |
|---------------------|------------------------------------|
| 1. Test SRAM        | Runs the RAM test on the SRAM.     |
| 2. Test BRAM        | Runs the RAM test on the BRAM.     |
| 3. Test DDR SDRAM   | Runs the RAM test on the DDR SDRAM |
| 5. Scan PCI Bus     | Scans the PCI bus for devices.     |
| 6. Test LCD Display | Runs different tests on the LCD.   |

Table 3-41: Main.Tests Menu

| Menu Points  | Description                                   |
|--------------|-----------------------------------------------|
| 7. Test LEDs | Tests LEDs attached to the GPIO.              |
| 8. Test IIC  | Tests the IIC controller and attached devices |
| 0. Main Menu | Returns to the Main menu.                     |

## Main.Cache Settings Menu

The Main.Cache Settings menu allows the user to enable/disable the instruction and/or the data cache. Per default both caches are enabled. Changing a setting takes effect immediately. The menu options are described in [Table 3-42](#).

Table 3-42: Main.Cache Settings Menu

| Menu Points                              | Description                                                   |
|------------------------------------------|---------------------------------------------------------------|
| 1. Instruction cache is disabled/enabled | Toggles between enabling and disabling the instruction cache. |
| 2. Data cache is disabled/enabled        | Toggles between enabling and disabling the data cache.        |
| 0. Main Menu                             | Returns to the Main menu.                                     |

## Test Characterization

The xrom demonstration and test application allows to run tests on the hardware. The characterization of the tests is specific for the simulation in V2PDK.

### RAM Test

The RAM test verifies the integrity of the different types of RAM in the system, searches for problems with the data and address lines, and exercises all the different types of bus operations. Per default the test only checks the first 128 byte of RAM to speed up the simulation. The amount of RAM tested can be changed in the source code.

The following four tables list all the steps that are executed during a RAM test, and describes what each single step is doing.

[Table 3-43, page 1645](#): **Single Byte, Halfword, and Word Accesses**

[Table 3-44, page 1647](#): **Four-Word Transactions**

[Table 3-45, page 1648](#): **Eight-Word Transactions**

[Table 3-46, page 1649](#): **Combinations**

Table 3-43: Single Byte, Halfword, and Word Accesses

| Test                     | Description                                                                                                                                     |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| Availability             | The whole memory region is read byte per byte. A bus error results that is reported by the test if the memory or only part of it are available. |
| Data lines stuck at one  | Test all 64 data lines and check if any of them is stuck at one.                                                                                |
| Data lines stuck at zero | Test all 64 data lines and check if any of them is stuck at zero.                                                                               |

Table 3-43: Single Byte, Halfword, and Word Accesses (Continued)

| Test                                    | Description                                                                                                                                                                                                         |
|-----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Walking ones on data lines              | Shift a single one bit through all 64 data lines. All other data lines are set to zero. Finds shorts between data lines.                                                                                            |
| Walking zeros on data lines             | Shift a single zero bit through all 64 data lines. All other data lines are set to one. Finds shorts between data lines.                                                                                            |
| Address lines stuck at one              | Test as many address lines as memory region has and check if any of them is stuck at one.                                                                                                                           |
| Address lines stuck at zero             | Test as many address lines as memory region has and check if any of them is stuck at zero.                                                                                                                          |
| Walking ones on address lines           | Shift a single one bit through as many address lines as memory region has. All other address lines are set to zero. Finds shorts between address lines.                                                             |
| Walking zeros on address lines          | Shift a single zero bit through as many address lines as memory region has. All other address lines are set to one. Finds shorts between address lines.                                                             |
| Walking ones on address and data lines  | Shift a single one bit through all 64 data lines and as many address lines as memory region has. All other data and address lines are set to zero. Finds shorts between data and address lines.                     |
| Walking zeros on address and data lines | Shift a single zero bit through all 64 data lines and as many address lines as memory region has. All other data and address lines are set to one. Finds shorts between data and address lines.                     |
| Random data                             | The whole memory region is non-sequentially written and verified with a random but deterministic data pattern. Finds problems with overlapping data transactions.                                                   |
| Refresh test.                           | Stop memory accesses for 30 seconds if memory needs to be refreshed. Finds problems with the refresh controller. This test is disabled for V2PDK.                                                                   |
| Write bytes<br>Read halfwords           | Write 9 bytes. Read back 4 halfwords twice. The first time with offset zero. The second time with offset one. Finds problems with byte ordering and unaligned halfword reads.                                       |
| Write bytes<br>Read words               | Write 11 bytes. Read back 2 words four times. The first time with offset zero, then with offset one, offset two, and offset three. Finds problems with byte ordering and unaligned word reads.                      |
| Write halfwords<br>Read bytes           | Write 4 halfwords twice. The first time with offset zero, the second time with offset one. After writing 4 halfwords read 8 bytes. Finds problems with byte ordering and unaligned halfword writes.                 |
| Write words<br>Read bytes               | Write 2 words four times. The first time with offset zero, then with offset one, offset two, and offset three. After writing 2 words read 8 bytes. Finds problems with byte ordering and unaligned word writes.     |
| Read/Write sequence                     | A read operation is executed which is immediately followed by a write operation. This test is done with equal and different addresses for the read and the write operation. Finds problems in read/write sequences. |

Table 3-43: Single Byte, Halfword, and Word Accesses (Continued)

| Test                       | Description                                                                                                                                                                                                                                                                                 |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Write/Read sequence        | A write operation is executed which is immediately followed by a read operation. This test is done with equal and different addresses for the read and the write operation. Finds problems in write/read sequences.                                                                         |
| Line Switching high to low | Two consecutive memory accesses are executed. The first time the data word 0xFFFFFFFF is written to the end of the memory, the second time the data word 0x0 is written to the beginning of the memory. Finds problems that occur when many address and data lines change from high to low. |
| Line Switching low to high | Two consecutive memory accesses are executed. The first time the data word 0x0 is written to the beginning of the memory, the second time the data word 0xFFFFFFFF is written to the end of the memory. Finds problems that occur when many address and data lines change from low to high. |

Table 3-44: Four-Word Transactions

| Test                | Description                                                                                                                                                                                                              |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Read bytes          | Execute 16 four-word transactions and each time read 16 bytes starting with different offsets (see <a href="#">Table 3-47, page 1650</a> ). The number of four-word transactions has to be verified in the simulation.   |
| Write bytes         | Execute 16 four-word transactions and each time write 16 bytes starting with different offsets (see <a href="#">Table 3-47, page 1650</a> ). The number of four-word transactions has to be verified in the simulation.  |
| Read halfwords      | Execute eight four-word transactions and each time read eight halfwords starting with different offsets. The number of four-word transactions has to be verified in the simulation.                                      |
| Write halfwords     | Execute eight four-word transactions and each time write eight halfwords starting with different offsets. The number of four-word transactions has to be verified in the simulation.                                     |
| Read words          | Execute four four-word transactions and each time read four words starting with different offsets. The number of four-word transactions has to be verified in the simulation.                                            |
| Write words         | Execute four four-word transactions and each time write four words starting with different offsets. The number of four-word transactions has to be verified in the simulation.                                           |
| Read/Write sequence | Issue a four-word read operation immediately followed by a four-word write operation. This test is done with equal and different addresses for the read and the write operation. Finds problems in read/write sequences. |



Table 3-44: Four-Word Transactions (Continued)

| Test                | Description                                                                                                                                                                                                                                                                               |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Write/Read sequence | Issue a four-word write operation immediately followed by a four-word read operation. This test is done with equal and different addresses for the read and the write operation. Finds problems in write/read sequences.                                                                  |
| Line switching      | Issue a four-word transaction consisting of 0x0 and 0xFFFFFFFF words following each other. Finds problems that occur when many data lines change at the same time.                                                                                                                        |
| Execute code        | A program with 4 instructions is copied to the RAM and executed four times each time with a starting address incremented by four. This issues four four-word transactions. Finds problems with executing code. The number of four-word transactions has to be verified in the simulation. |

Table 3-45: Eight-Word Transactions

| Test                | Description                                                                                                                                                                                                                  |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Read bytes          | Execute 32 eight-word transactions and each time read 32 bytes starting with different offsets. The number of eight-word transactions has to be verified in the simulation.                                                  |
| Write bytes         | Execute 32 eight-word transactions and each time write 32 bytes starting with different offsets. The number of eight-word transactions has to be verified in the simulation.                                                 |
| Read halfwords      | Execute 16 eight-word transactions and each time read 16 halfwords starting with different offsets. The number of eight-word transactions has to be verified in the simulation.                                              |
| Write halfwords     | Execute 16 eight-word transactions and each time write 16 halfwords starting with different offsets. The number of eight-word transactions has to be verified in the simulation.                                             |
| Read words          | Execute eight eight-word transactions and each time read eight words starting with different offsets. The number of eight-word transactions has to be verified in the simulation.                                            |
| Write words         | Execute eight eight-word transactions and each time write eight words starting with different offsets. The number of eight-word transactions has to be verified in the simulation.                                           |
| Read/Write sequence | Issue an eight-word read operation immediately followed by an eight-word write operation. This test is done with equal and different addresses for the read and the write operation. Finds problems in read/write sequences. |

Table 3-45: Eight-Word Transactions (Continued)

| Test                | Description                                                                                                                                                                                                                                                                                     |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Write/Read sequence | Issue an eight-word write operation immediately followed by an eight-word read operation. This test is done with equal and different addresses for the read and the write operation. Finds problems in write/read sequences.                                                                    |
| Line switching      | Issue an eight-word transaction consisting of 0x0 and 0xFFFFFFFF words following each other. Finds problems that occur when many data lines change at the same time.                                                                                                                            |
| Execute code        | A program with eight instructions is copied to the RAM and executed eight times each time with a starting address incremented by four. This issues four four-word transactions. Finds problems with executing code. The number of eight-word transactions has to be verified in the simulation. |

Table 3-46: Combinations

| Test                                                                            | Description                                                                                          |
|---------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| Data: Single byte/<br>word/halfword<br>Instruction: four-<br>word transactions  | Issue single-word transactions on the data side and four-word transactions on the instruction side.  |
| Data: Single byte/<br>word/halfword<br>Instruction: eight-<br>word transactions | Issue single-word transactions on the data side and eight-word transactions on the instruction side. |
| Data: four-word<br>transactions<br>Instruction: four-<br>word transactions      | Issue four-word transactions on the data side and four-word transactions on the instruction side.    |
| Data: four-word<br>transactions<br>Instruction: eight-<br>word transactions     | Issue four-word transactions on the data side and eight-word transactions on the instruction side.   |
| Data: eight-word<br>transactions<br>Instruction: four-<br>word transactions     | Issue eight-word transactions on the data side and four-word transactions on the instruction side.   |
| Data: eight-word<br>transactions<br>Instruction: eight-<br>word transactions    | Issue eight-word transactions on the data side and eight-word transactions on the instruction side.  |

**Table 3-47** is read from top-left to bottom-right. The first four-word transaction is issued by reading from offset 0. No other transactions should be issued when the bytes with offset 1 to 15 are accessed. The second four-word transaction is issued by reading from offset 17. No other transactions should be issued when the bytes with offset 18 to 31 and 16 are accessed.

Similar access patterns can be generated for halfword and word accesses, both for four-word and eight-word transactions.

**Table 3-47: Access Pattern for Four-Word Transactions with Byte Read or Write Operations**

|                              |    | Byte Access Number |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|------------------------------|----|--------------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|                              |    | 1                  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| Four-Word Transaction Number | 1  | 0                  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
|                              | 2  | 17                 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 16 |
|                              | 3  | 2                  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 0  | 1  |
|                              | 4  | 19                 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 16 | 17 | 18 |
|                              | 5  | 4                  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 0  | 1  | 2  | 3  |
|                              | 6  | 21                 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 16 | 17 | 18 | 19 | 20 |
|                              | 7  | 6                  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 0  | 1  | 2  | 3  | 4  | 5  |
|                              | 8  | 23                 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|                              | 9  | 8                  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
|                              | 10 | 25                 | 26 | 27 | 28 | 29 | 30 | 31 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|                              | 11 | 10                 | 11 | 12 | 13 | 14 | 15 | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
|                              | 12 | 27                 | 28 | 29 | 30 | 31 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
|                              | 13 | 12                 | 13 | 14 | 15 | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 |
|                              | 14 | 29                 | 30 | 31 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |
|                              | 15 | 14                 | 15 | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 |
|                              | 16 | 31                 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |

## Scan PCI Bus

The scan pci test is equivalent to the scan\_pci demonstration and test application (see [page 1642](#)).

## LCD Test

The LCD test verifies that the LCD hardware and the software driver in the BSP are working properly. [Table 3-48](#) lists all the steps that are executed during an LCD test and describes what each single step is doing.

**Table 3-48: LCD Test Steps**

| Test                 | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| "Hello world" banner | <p>The "Hello world" banner test consists of three steps:</p> <ul style="list-style-type: none"> <li>• Clear display</li> <li>• Set cursor position</li> <li>• Print "Hello world" at cursor position</li> </ul> <p>These steps are executed 64 times. The first time the string is printed to the top left corner (cursor position 0). The other 63 times, the cursor position is each time incremented by one.</p> <p>Finds problems with end of line wraps, end of display wrap, and positioning characters on the display.</p> |
| Defective Pixels     | <p>Fill the display with the ASCII character 0xFF (filled blank). The user visually searches for defective pixels. Offers the user a means to test the display for defective pixels.</p>                                                                                                                                                                                                                                                                                                                                           |
| Writing Pattern      | <p>Fill the display with a pattern. The first character ("0") is written to cursor position zero. The second character ("1") is written to cursor position three. In the same manner, each following character is written to the current cursor position plus three.</p> <p>Finds problems with random write accesses to the display.</p>                                                                                                                                                                                          |

## LED Test

The LED test assumes that eight LEDs are attached to the least significant eight bits of the GPIO. It verifies that the LEDs and the software driver for the GPIO are working properly. [Table 3-49](#) lists all the steps that are executed during a LED test and describes what each single step is doing.

**Table 3-49: LED Test Steps**

| Test <sup>(1)</sup> | Description                                                                                                                                   |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| Turn LEDs off       | <p>Turn all LEDs off. The user visually verifies that all LEDs are off. Finds problems with LEDs constantly turned on.</p>                    |
| Turn LEDs on        | <p>Turn all LEDs on. The user visually verifies that all LEDs are on. Finds problems with LEDs constantly turned off.</p>                     |
| Walking one         | <p>Walk a single one through all LEDs. The user visually verifies that at any time only one LED is on. Finds shorts between GPIO lines.</p>   |
| Walking zero        | <p>Walk a single zero through all LEDs. The user visually verifies that at any time only one LED is off. Finds shorts between GPIO lines.</p> |

**Notes:**

1. The test only checks the least significant eight bits of the 32-bit wide GPIO.

## AT24C64 IIC EEPROM Test

The IIC test verifies that the IIC controller and the AT24C64 EEPROM and their software drivers in the BSP are working properly. To speed up the simulation for V2PDK the test is limited to the 32 bytes in the first and the 32 bytes in the last page of the EEPROM. It is assumed that the start of the test area is the first byte of the last page and the end of the test area is the last byte of the first page. Table 3-50 lists all the steps that are executed during an IIC test and describes what each step is doing.

**Table 3-50: IIC Test Steps**

| Test                                                | Description                                                                                                                                                                                                              |
|-----------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Single-byte write cycles<br>Single-byte read cycles | Backwards fill the test area with single byte writes. Forwards read the test area with single byte reads. Finds problems with too long write sequences and overwriting data during the dummy write cycles while reading. |
| Page write cycles<br>Multiple-byte read cycles      | Write a page twice. The first time starting at offset zero, the second time starting at another offset. After writing a page verify all bytes by multiple-byte read cycles. Finds problems with writing pages.           |
| Multiple-byte read cycles                           | Read the whole test area with a single read operation. Finds problems with reading multiple bytes and read cycles over page boundaries.                                                                                  |

## Files

**Table 3-51: Files Used by the xrom Application**

| File                     | Description <sup>(1)</sup>                                                                                                  |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| README                   | Brief description of the application                                                                                        |
| Makefile                 | Makefile to build the application                                                                                           |
| globaltypes.h            | Contains global types used in several source files                                                                          |
| version.h                | Contains version information. Automatically generated during a build                                                        |
| xrom.c, xrom.h           | Main program. Prints current menu, handles user input, and blinks LEDs                                                      |
| xromabort.c, xromabort.h | Checks whether user aborts operation by pressing escape key (ESC, 0x1B, 27d) or tabulator key (TAB, 0x9, 9d) <sup>(1)</sup> |
| xromcache.c, xromcache.h | Provides functions to initialize, enable and disable the data and instruction cache                                         |
| xromiic.c, xromiic.h     | Provides functions to initialize the IIC controller and to test the ATMEL IIC EEPROM AT24C64                                |
| xrominput.c, xrominput.h | Receives and processes input from the simulation terminal                                                                   |
| xromlcd.c, xromlcd.h     | Provides functions to initialize and test the LCD                                                                           |
| xromled.c, xromled.h     | Provides functions to initialize the GPIO and to handle and test LEDs attached to the GPIO                                  |
| xromlog.c, xromlog.h     | Provides functions to output strings and numbers to the simulation terminal                                                 |

Table 3-51: Files Used by the xrom Application (Continued)

| File                         | Description <sup>(1)</sup>                                                              |
|------------------------------|-----------------------------------------------------------------------------------------|
| xrompci.c, xrompci.h         | Provides functions to initialize the IIC controller and to scan the PCI bus for devices |
| xromtestram.c, xromtestram.h | Provides functions to test different types of RAM                                       |
| xromtests.c, xromtests.h     | Contains the test menu                                                                  |
| v2pdk/debug/*.o              | Object files after a build with debug options                                           |
| v2pdk/release/*.o            | Object files after a build with optimization options                                    |
| v2pdk/debug/xrom.elf         | Executable after a build with debug options                                             |
| v2pdk/release/xrom.elf       | Executable after a build with optimization options                                      |

**Notes:**

1. Pressing the Esc key to abort an operation is not recommended, as it will end the simulation.

## Building for V2PDK

The file Makefile contains the target rules to build the debug and release versions of xrom. Table 3-52 lists the possible targets for V2PDK. To build a specific target execute:

```
$ make <target>
```

Table 3-52: V2PDK Possible Targets

| Target (<target>)  | Description                                                                                     |
|--------------------|-------------------------------------------------------------------------------------------------|
| xrom_debug_v2pdk   | Builds executable with debug options (-gdwarf). Links against the debug version of the BSP      |
| xrom_release_v2pdk | Builds executable with optimization options (-O3). Links against the release version of the BSP |
| clean              | Removes object files and executables of the debug and release version                           |
| megaclean          | Calls target clean and removes temporary files                                                  |
| BUILD-V2PDK        | Calls the targets megaclean and xrom_release_v2pdk                                              |

For V2PDK, the baud rate (2.083333 Mbaud) is calculated so that the divisor latch inside the UART is set to three. This speeds up the simulation significantly.

Some definitions added with “-D” to the CFLAGS in the “define compiler settings” section of the Makefile influence the build process as listed in Table 3-53.

Table 3-53: Optionally Added Definitions to CFLAGS

| Define   | Description                                                 |
|----------|-------------------------------------------------------------|
| SPEED_UP | Speed up for simulation—i.e., do not call “sleep” functions |

Table 3-54 lists which libraries are required for building the application.

Table 3-54: Required Libraries

| Library | Description           |
|---------|-----------------------|
| BSP     | Board Support Package |



# Simulation Terminal

---

## Overview

The Xilinx Simulation Terminal is located in `$V2PRO/tools/$V2PRO_HOST_OSTYPE/xilinx/bin/terminal`. It offers user-interactive access to a UART during simulation.

The tool requires two parameters for an input and an output file. The Simulation Terminal writes characters typed by the user to the input file. The input file is read by the testbench during simulation. In the opposite direction, the testbench writes characters to an output file. The Simulation Terminal reads the characters from the output file and displays them on the screen.

The escape character (**ESC**, **0x19**, **27d**) terminates the Simulation Terminal.

## Syntax

The command

```
$ terminal <input file> <output file>
```

runs the Simulation Terminal, where *<input file>* and *<output file>* have to be replaced with the file names for the input and the output file.

The shell script

`$V2PRO/tools/$V2PRO_HOST_OSTYPE/xilinx/bin/start_terminal` is an example that shows how the terminal is used. It removes old files and starts the Simulation Terminal with the correct parameters.

## File Format

When using the Simulation Terminal for interaction with the simulation, the user does not need to edit the input or the output file. The **EBNF** specification of the input and the output file are given here for reference.

### Terminology

- A decimal is a decimal number  $\geq 0$ .
- An 8-bit-hexadecimal is a hexadecimal number in the range [0x0, 0xFF].

### Input File

1. `inputFile = countLine {characterLine} .`
2. `countLine = "@" decimal " " decimal .`
3. `characterLine = "@" decimal " " 8-bit-hexadecimal .`



- The first line (**countLine**) contains the number of characters in the file.
- All sub-subsequent lines (**characterLine**) each contain the ASCII code of one character.
- The decimal after the "@" denotes the line number and increases by one per line.

## Output File

1. outputFile = **countLine** {**characterLine**} .
2. countLine = **decimal** .
3. characterLine = **8-bit-hexadecimal** .

- The first line (**countLine**) contains the number of characters in the file.
- All sub-subsequent lines (**characterLine**) each contain the ASCII code of one character.

## Remarks

- **Implementation restriction:** the number of characters in the input file is restricted to 1024.
- **File naming convention:** Current testbenches expect the input files to be named as in [Table 4-1, page 1656](#):

Table 4-1: Simulation File Names

|        | Uart1        | Uart2        |
|--------|--------------|--------------|
| input  | Uart1.input  | Uart2.input  |
| output | Uart1.output | Uart2.output |