Virtex-II Pro™ Platform FPGA Documentation

- ***Advance Product Specification***
- ***PPC405 User Manual***
- ***PPC405 Processor Block Manual***
- ***Rocket I/O™ Transceiver User Guide***

March 2002 Release

XILINX®

Table of Contents

Volume 1:
Virtex-II Pro™ Platform FPGA Advance Product Specification

Data Sheet Module 1:
Virtex-II Pro™ Platform FPGAs: Introduction and Overview

Data Sheet Module 2:
Virtex-II Pro™ Platform FPGAs: Functional Description

Data Sheet Module 3: Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics

Data Sheet Module 4: Virtex-II Pro™ Platform FPGAs: Pinout Information

Volume 2:
Virtex-II Pro™ Processor

Volume 2(a):
PPC405 User Manual

About This Book

Chapter 1: Introduction to the PPC405

Chapter 2: Operational Concepts

Chapter 3: User Programming Model

Chapter 4: PPC405 Privileged-Mode Programming Model

Chapter 5: Memory-System Management

Chapter 6: Virtual-Memory Management

Chapter 7: Exceptions and Interrupts

Chapter 8: Timer Resources

Chapter 9: Debugging

Chapter 10: Reset and Initialization

Chapter 11: Instruction Set

Appendix A: Register Summary

Appendix B: Instruction Summary

Appendix C: Simplified Mnemonics

Appendix D: Programming Considerations

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

Chapter 2: Input/Output Interfaces

Chapter 3: PowerPC® 405 OCM Controller

Appendix A: RISCWatch and RISCTrace Interfaces

Volume 3: Rocket I/O™ Transceiver User Guide

Chapter 1: Introduction

Chapter 2: Rocket I/O™ Transceiver Overview

Chapter 3: Digital Design Considerations

Chapter 4: Analog Design Considerations

Chapter 5: Simulation and Implementation

Appendix A: Rocket I/O™ Cell Models

Schedule of Figures

Volume 1:
Virtex-II Pro™ Platform FPGA Advance Product Specification

Data Sheet Module 1:
Virtex-II Pro™ Platform FPGAs: Introduction and Overview

Data Sheet Module 2:
Virtex-II Pro™ Platform FPGAs: Functional Description

Data Sheet Module 3: Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics

Data Sheet Module 4: Virtex-II Pro™ Platform FPGAs: Pinout Information

Volume 2:
Virtex-II Pro™ Processor

Volume 2(a):
PPC405 User Manual

About This Book

Chapter 1: Introduction to the PPC405

Chapter 2: Operational Concepts

Chapter 3: User Programming Model

Chapter 4: PPC405 Privileged-Mode Programming Model

Chapter 5: Memory-System Management

Chapter 6: Virtual-Memory Management

Chapter 7: Exceptions and Interrupts

Chapter 8: Timer Resources

Chapter 9: Debugging

Chapter 10: Reset and Initialization

Chapter 11: Instruction Set

Appendix A: Register Summary

Appendix B: Instruction Summary

Appendix C: Simplified Mnemonics

Appendix D: Programming Considerations

Chapter 4: Analog Design Considerations

Chapter 5: Simulation and Implementation

Appendix A: Rocket I/O™ Cell Models

Schedule of Tables

Volume 1:
Virtex-II Pro™ Platform FPGA Advance Product Specification

Data Sheet Module 1:
Virtex-II Pro™ Platform FPGAs: Introduction and Overview

Data Sheet Module 2:
Virtex-II Pro™ Platform FPGAs: Functional Description

Data Sheet Module 3: Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics

Data Sheet Module 4: Virtex-II Pro™ Platform FPGAs: Pinout Information

Volume 2: Virtex-II Pro™ Processor

Volume 2(a): PPC405 User Manual

About This Book

Chapter 1: Introduction to the PPC405

Chapter 2: Operational Concepts

Chapter 3: User Programming Model

Chapter 4: PPC405 Privileged-Mode Programming Model

Chapter 5: Memory-System Management

Chapter 6: Virtual-Memory Management

Chapter 7: Exceptions and Interrupts

Chapter 8: Timer Resources

Chapter 9: Debugging

Chapter 10: Reset and Initialization

Chapter 11: Instruction Set

Appendix A: Register Summary

Appendix B: Instruction Summary

Appendix C: Simplified Mnemonics

Chapter 1: Introduction to the PowerPC® 405 Processor

Chapter 2: Input/Output Interfaces

Chapter 3: PowerPC® 405 OCM Controller

Appendix A: RISCWatch and RISCTrace Interfaces

Signal Summary

Volume 3: Rocket I/O™ Transceiver User Guide

Chapter 1: Introduction

Chapter 2: Rocket I/O™ Transceiver Overview

Chapter 3: Digital Design Considerations

Chapter 4: Analog Design Considerations

Chapter 5: Simulation and Implementation

Appendix A: Rocket I/O™ Cell Models

Volume 1: Virtex-II Pro™ Platform FPGA Advance Product Specification

Virtex-II Pro™ Platform FPGA Documentation

March 2002 Release

XILINX®

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 (UISA) 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:

- LVTTL
- 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 configuring 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 Enviornment (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 CompactFlash 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-ptich 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.

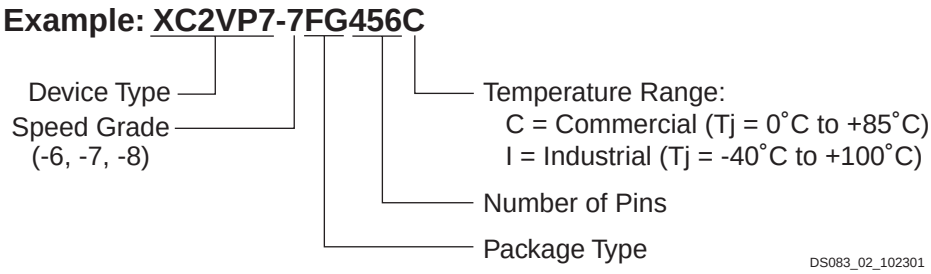


Figure 1: **Virtex-II Pro Ordering Information**

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

Virtex-II Pro Array Functional Description

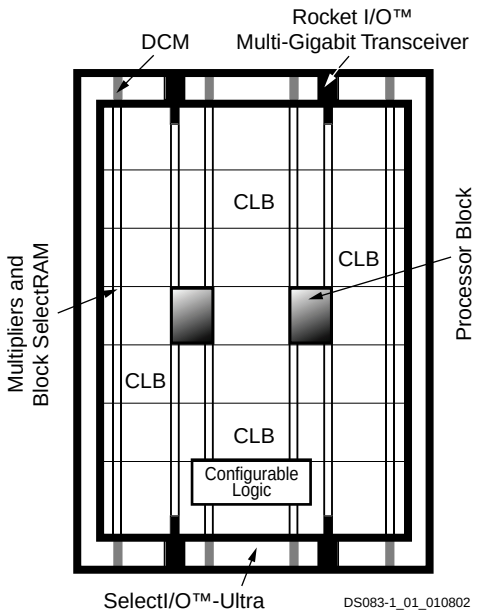


Figure 1: **Virtex-II Pro Generic Architecture Overview**

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 µm 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.

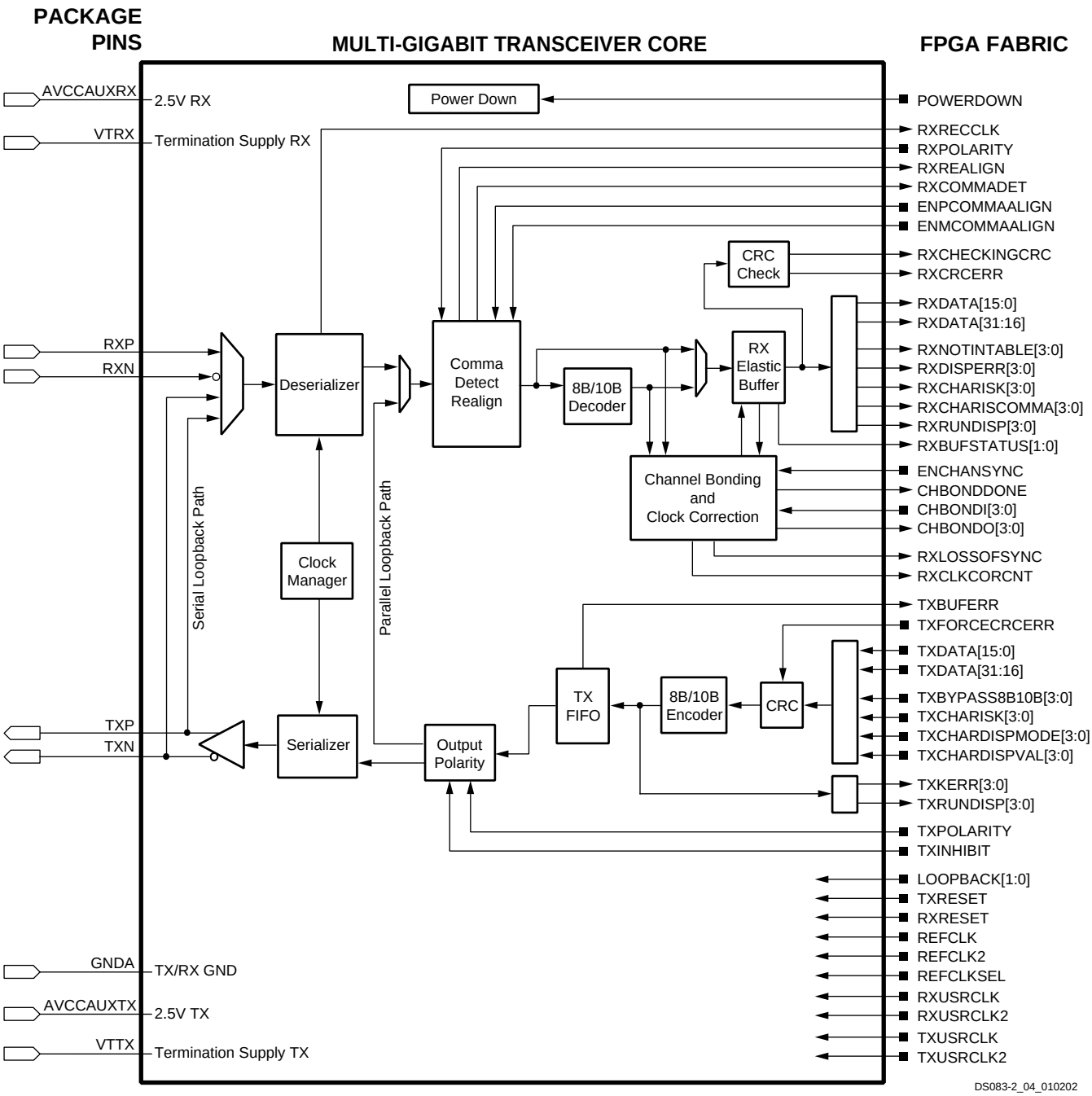


Figure 2: **Rocket I/O Block Diagram**

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 50Ω (default) and 75Ω 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.

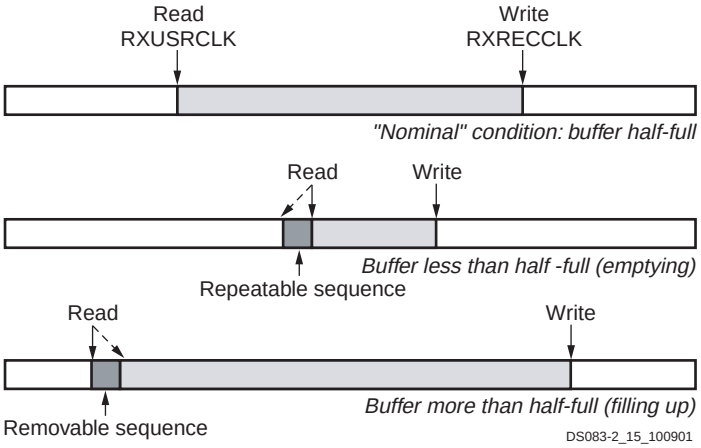


Figure 3: **Clock Correction in Receiver**

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.

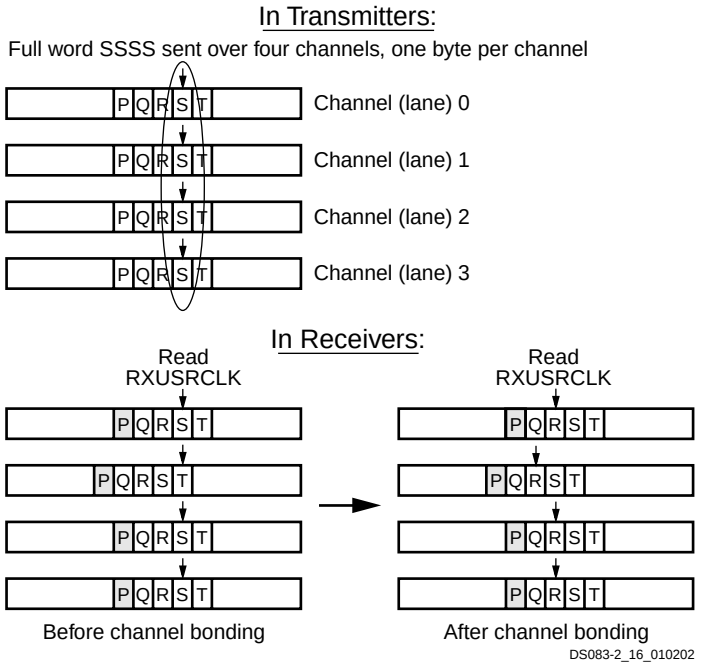


Figure 4: **Channel Bonding (Alignment)**

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 stucture beginning and ending with user-defined SOP and EOP characters.

Configuration

This section outlines functions that may be selected or controlled 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 POWERDOWN 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 AVCCAUXTX, V_{TTX}, and V_{TRX} in any order.

Functional Description: Processor Block

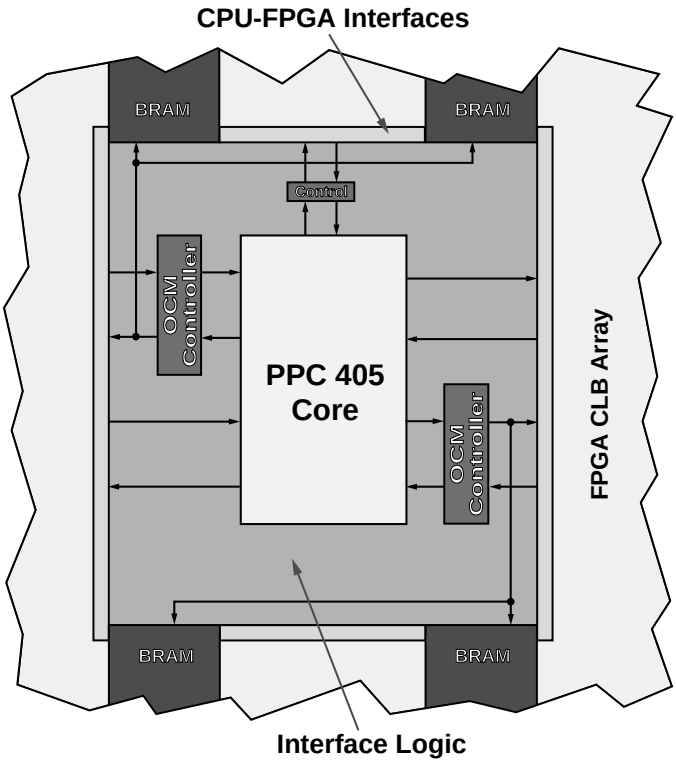


Figure 5: **Processor Block Architecture**

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. Therefore 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

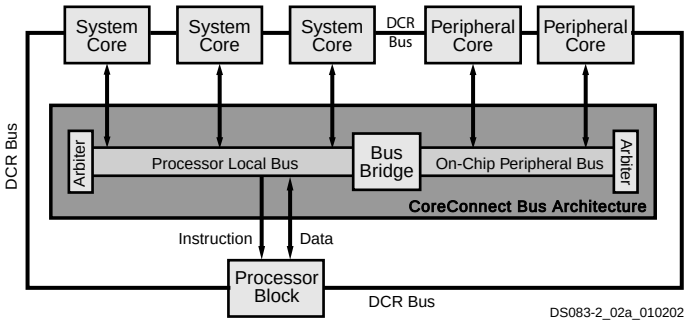


Figure 6: **CoreConnect Block Diagram**

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.nfs/productfamilies/CoreConnect_Bus_Architecture/

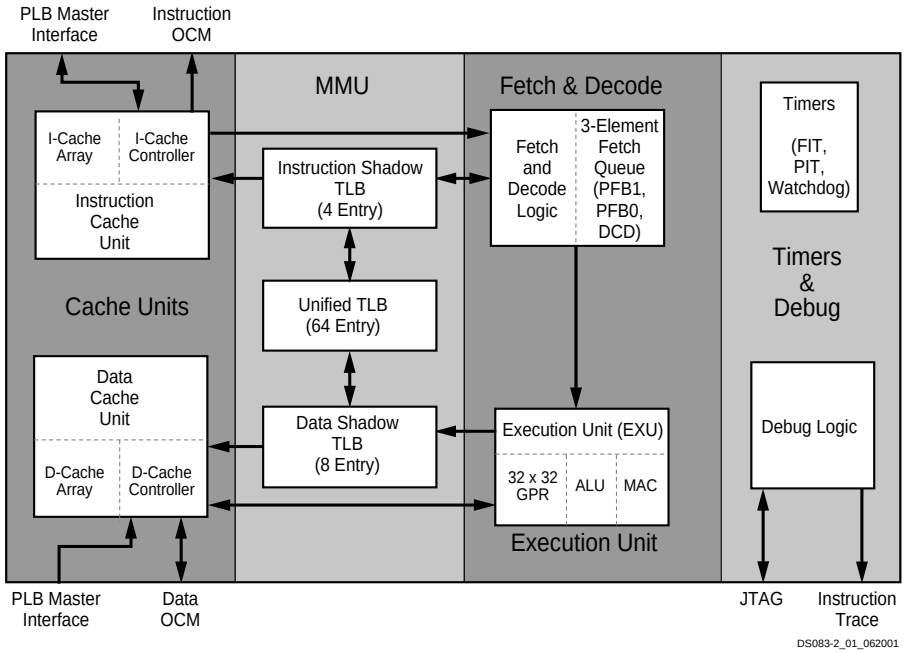


Figure 7: **PPC405 Core Block Diagram**

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.

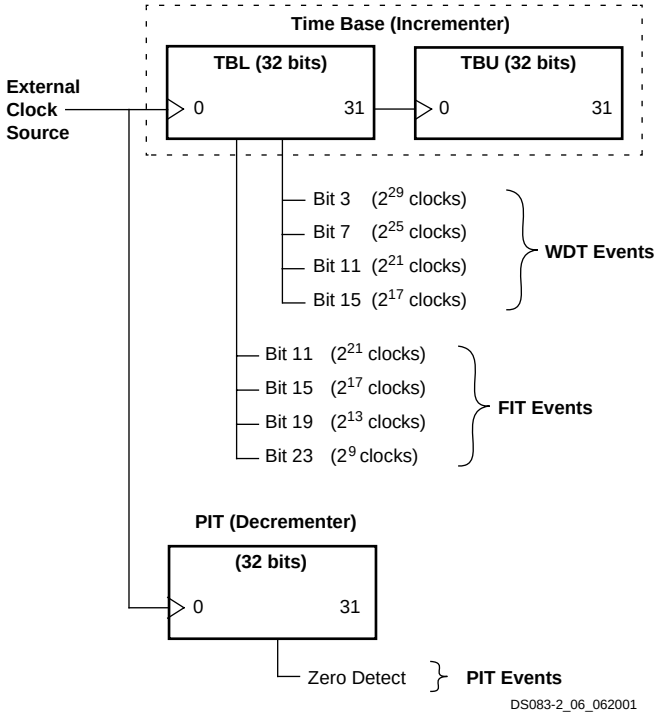


Figure 8: **Relationship of Timer Facilities to Base Clock**

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.

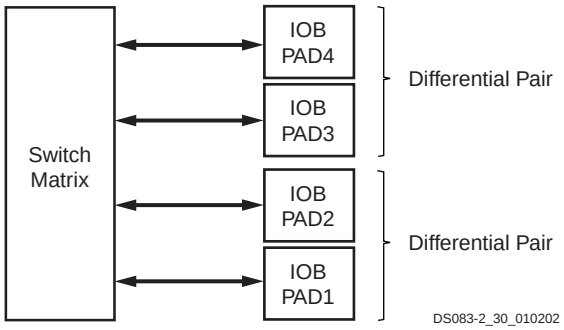


Figure 9: **Virtex-II Pro Input/Output Tile**

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})
LVTTL	3.3	3.3	N/A	N/A
LVCMOS33	3.3	3.3	N/A	N/A
LVCMOS25	2.5	2.5	N/A	N/A
LVCMOS18	1.8	1.8	N/A	N/A
LVCMOS15	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:
1. 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
LVDSEXT_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:
1. LVDCI_XX is LVCMOS controlled impedance buffers, matching the reference resistors or half of the reference resistors.
2. These are SSTL compatible.

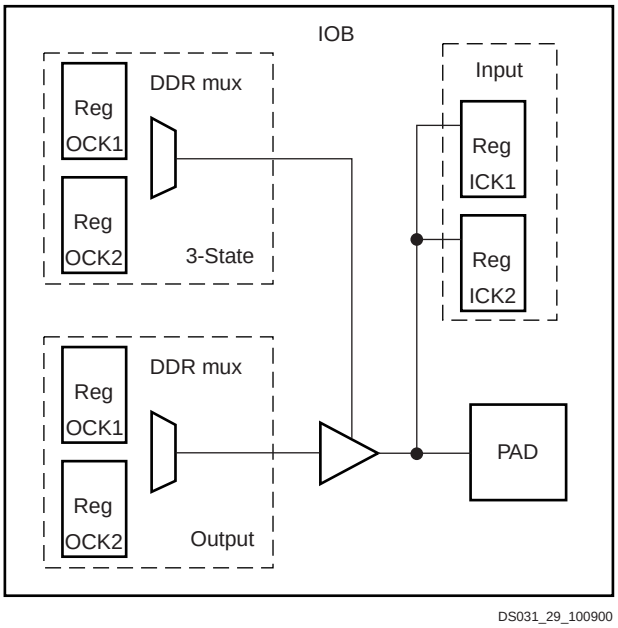


Figure 10: **Virtex-II Pro IOB Block**

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.

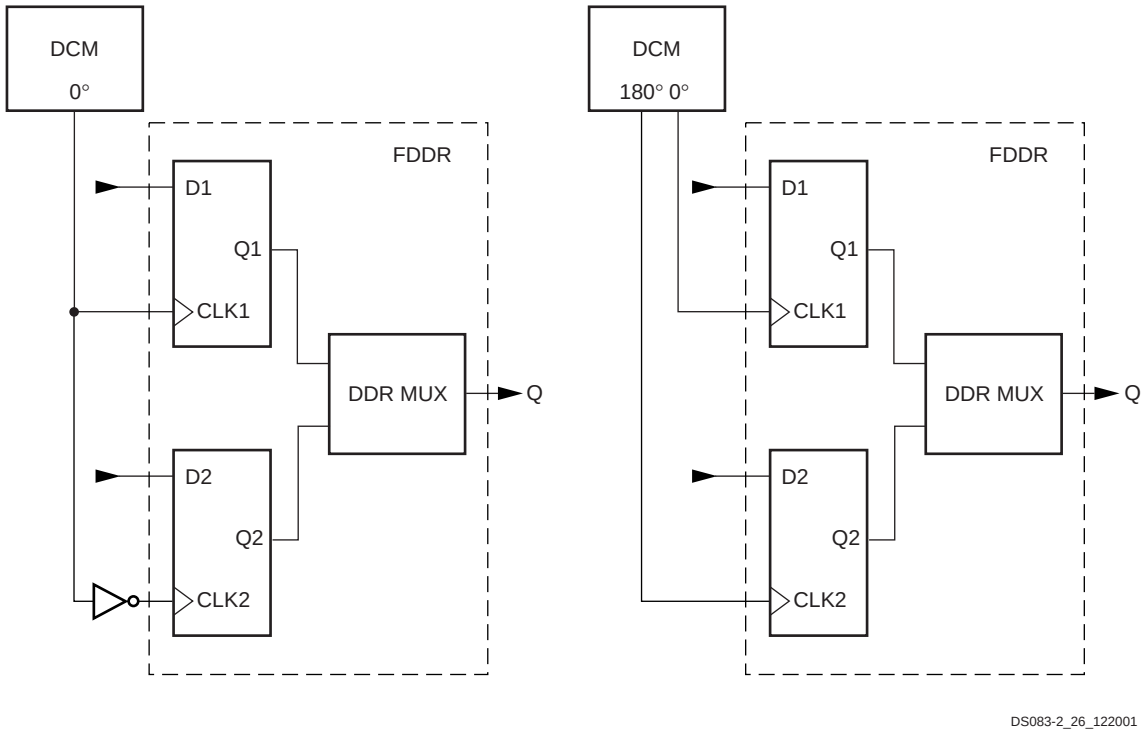


Figure 11: **Double Data Rate Registers**

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.

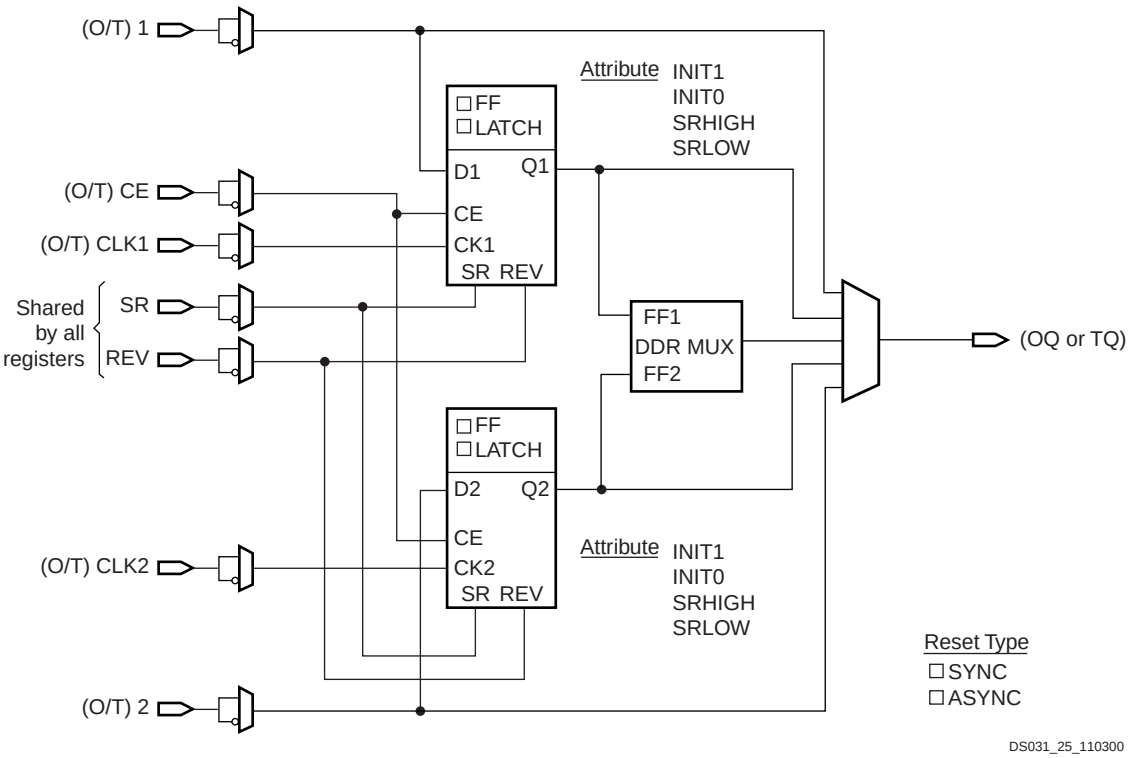


Figure 12: **Register / Latch Configuration in an IOB Block**

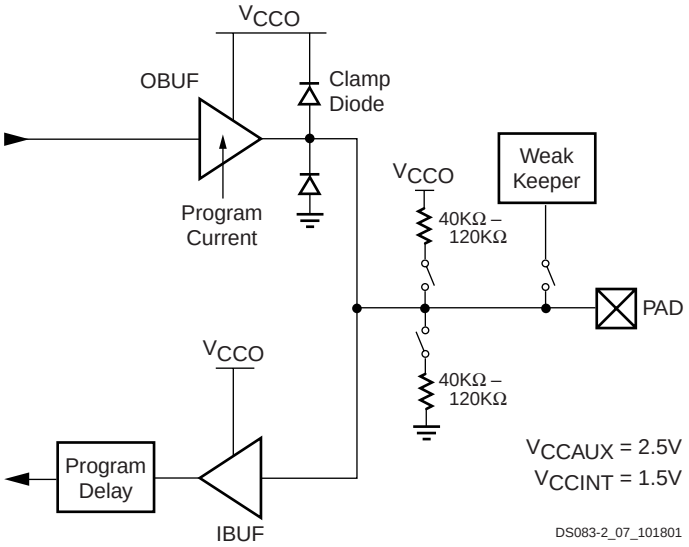


Figure 13: **LVCMOS SelectI/O Standard**

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 KΩ to 120 KΩ 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.

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)

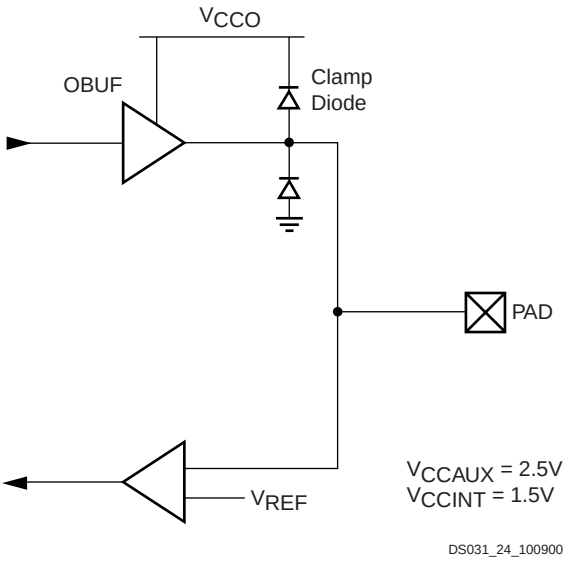


Figure 14: **SSTL or HSTL SelectI/O Standards**

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.

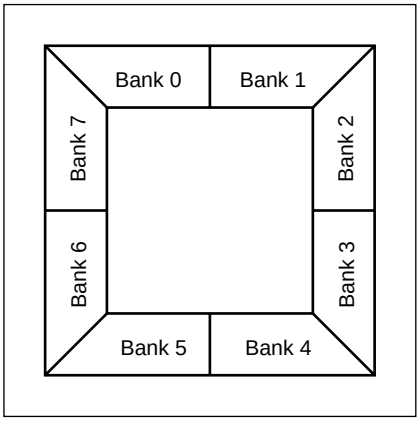


Figure 15: **Virtex-II Pro I/O Banks: Top View for Wire-Bond Packages (CS, FG, and BG)**

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.

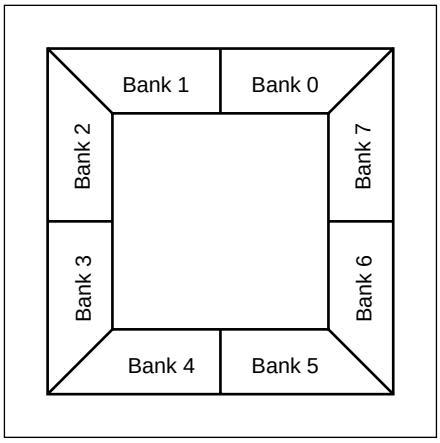


Figure 16: **Virtex-II Pro I/O Banks: Top View for Flip-Chip Packages (FF and BF)**

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), LVTTL, 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.

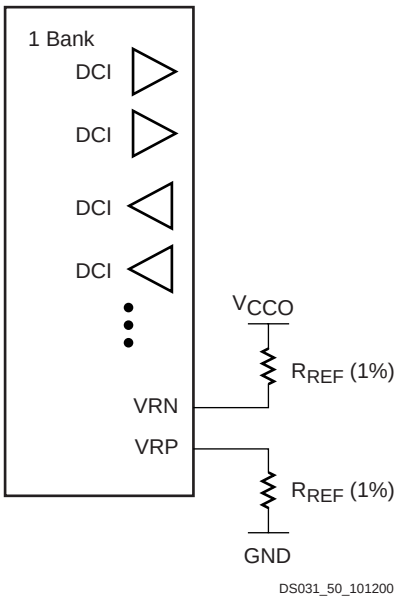


Figure 17: **DCI in a Virtex-II Pro Bank**

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.

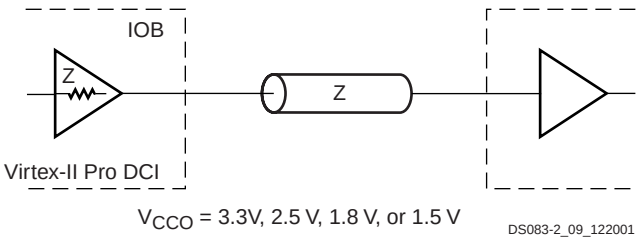


Figure 18: **Internal Series Termination**

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_DCI I/O standards.

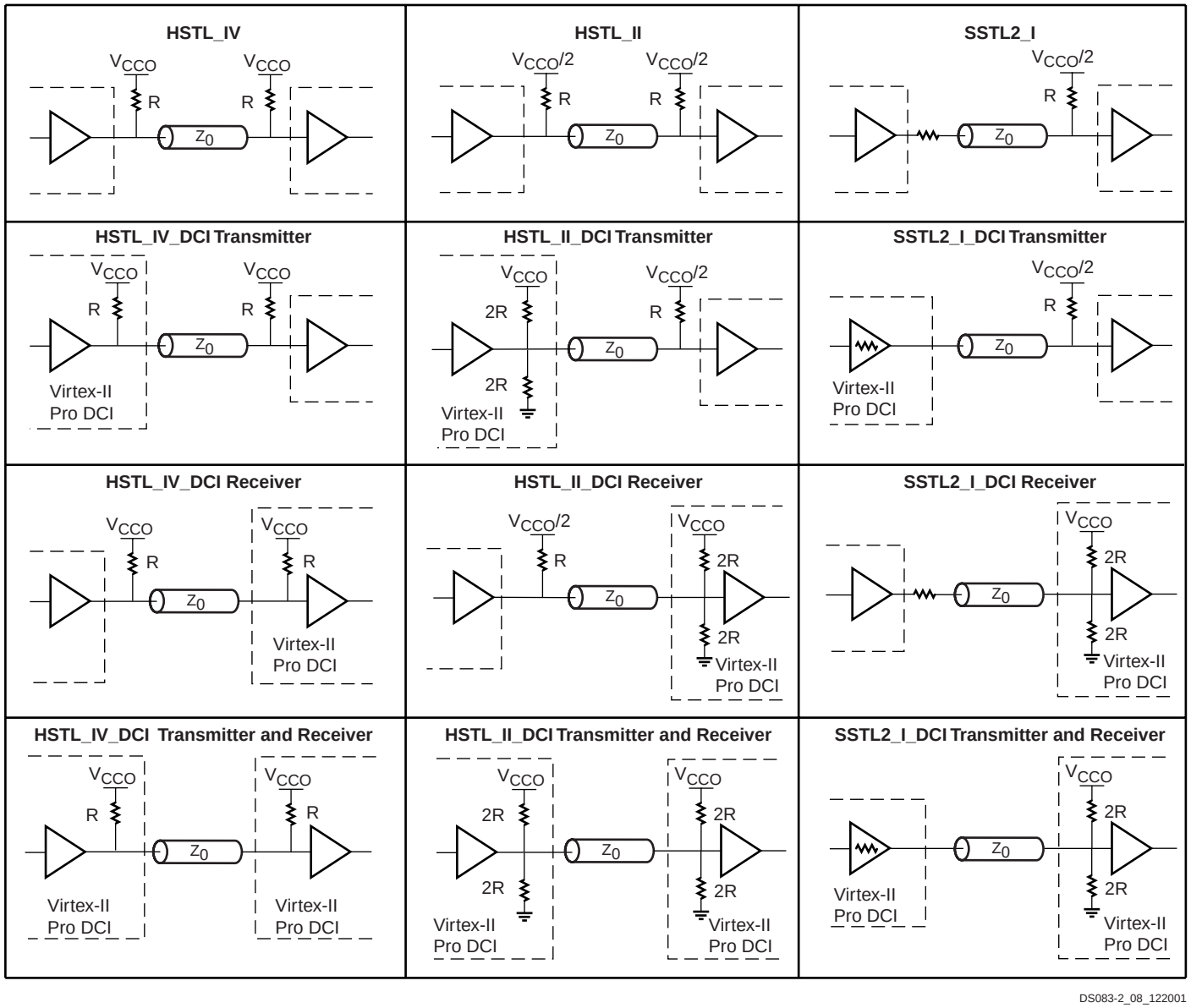


Figure 19: **DCI Usage Examples**

Configurable Logic Blocks (CLBs)

The Virtex-II Pro configurable logic blocks (CLB) are organized in an array and are used to build combinatorial 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.

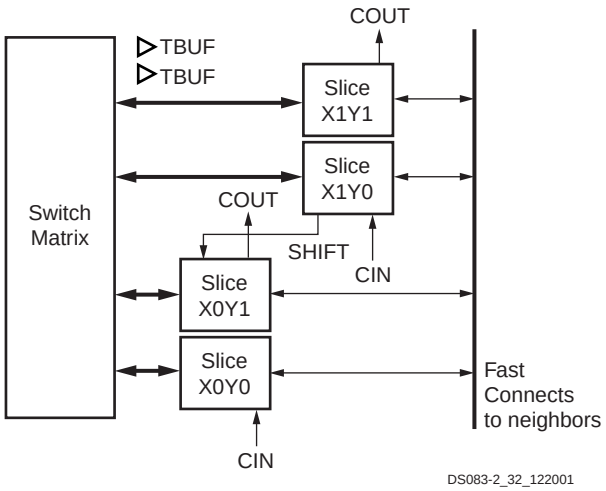


Figure 20: **Virtex-II Pro CLB Element**

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.

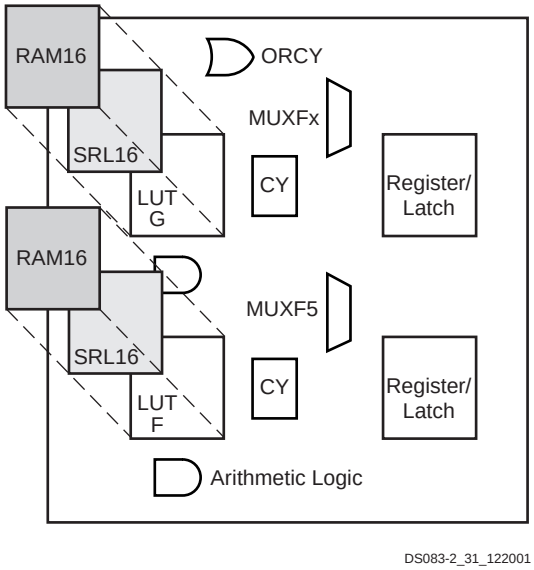


Figure 21: **Virtex-II Pro Slice Configuration**

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.

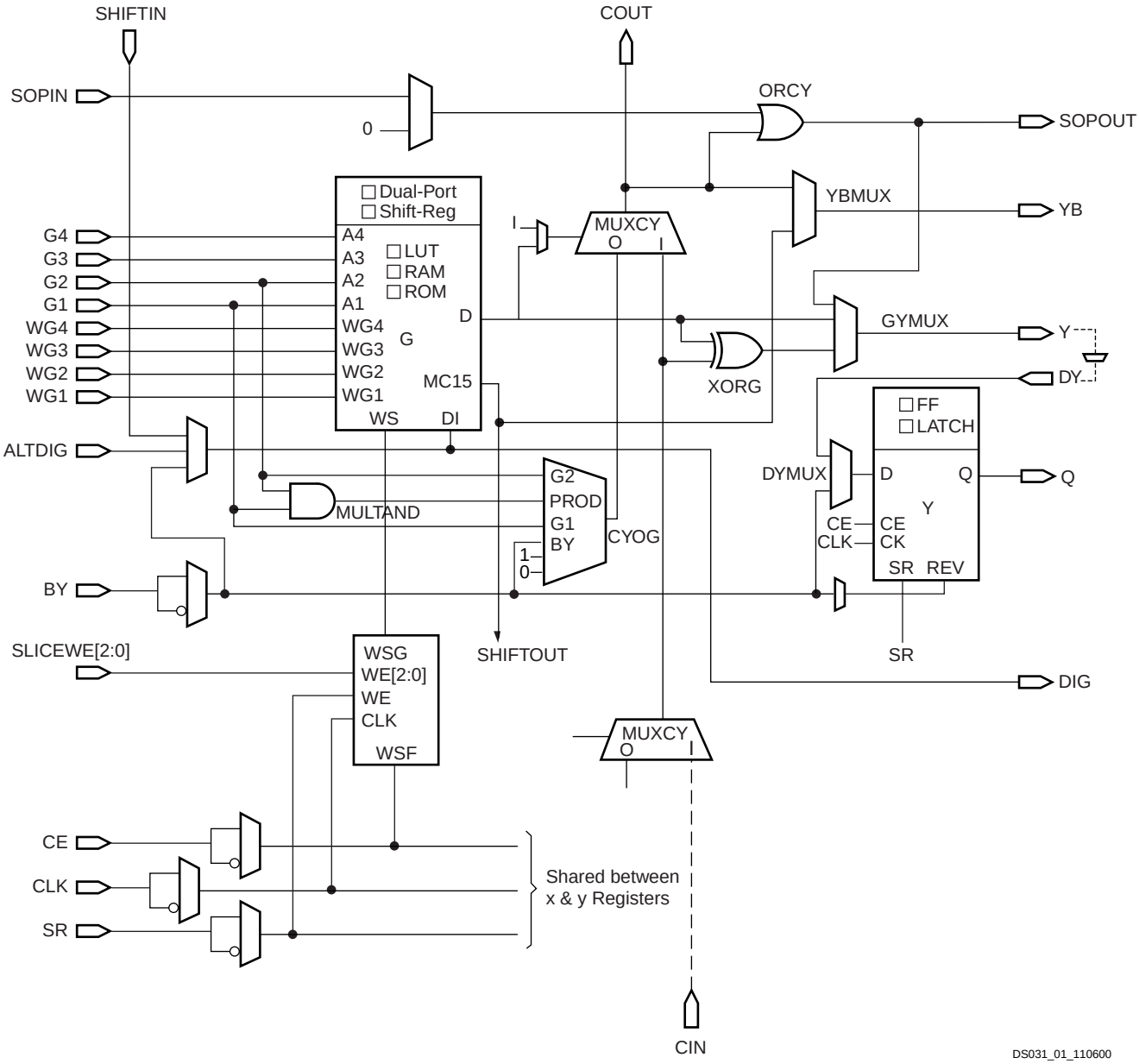


Figure 22: **Virtex-II Pro Slice (Top Half)**

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.

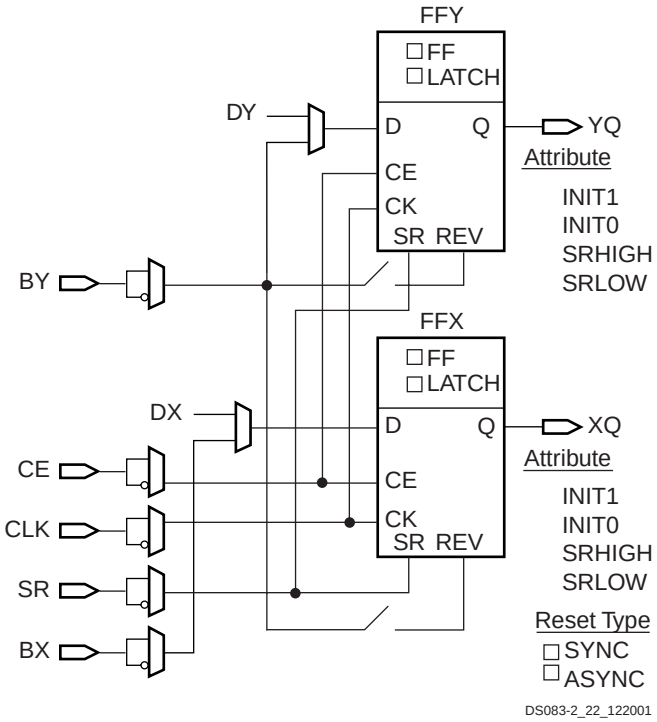


Figure 23: **Register / Latch Configuration in a Slice**

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

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.

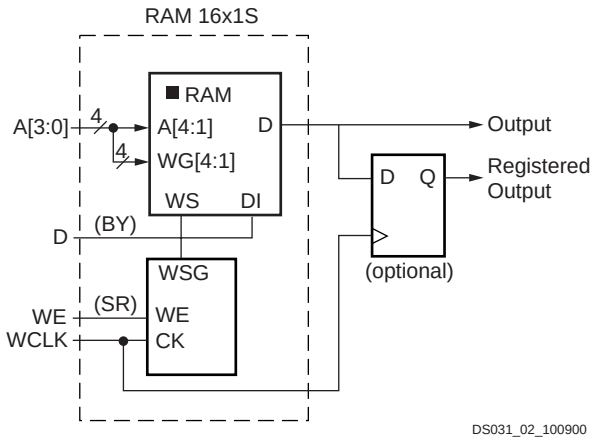


Figure 24: **Distributed SelectRAM (RAM16x1S)**

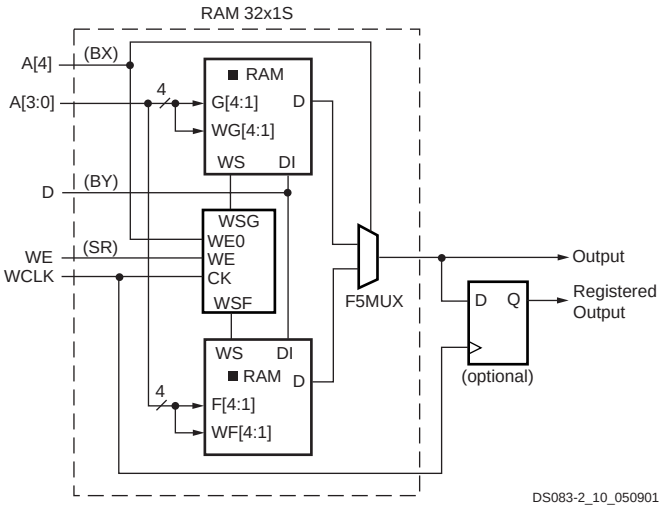


Figure 25: **Single-Port Distributed SelectRAM (RAM32x1S)**

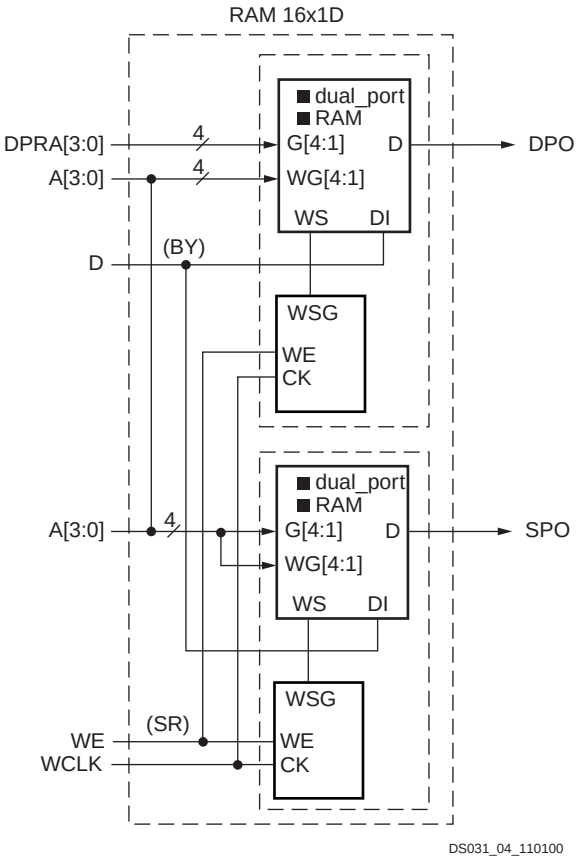


Figure 26: **Dual-Port Distributed SelectRAM (RAM16x1D)**

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.

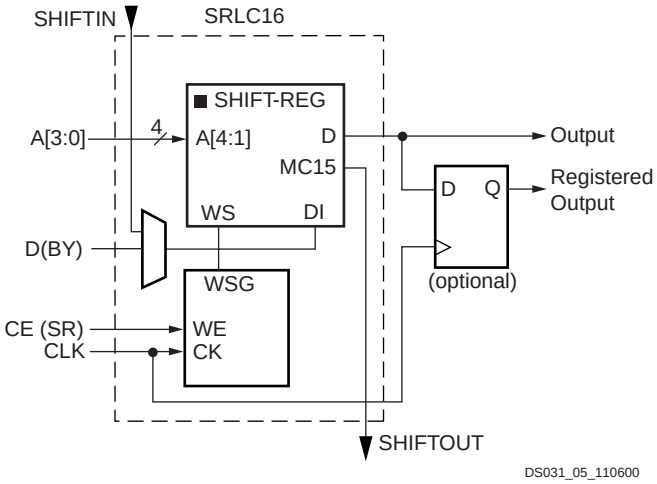


Figure 27: **Shift Register Configurations**

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.

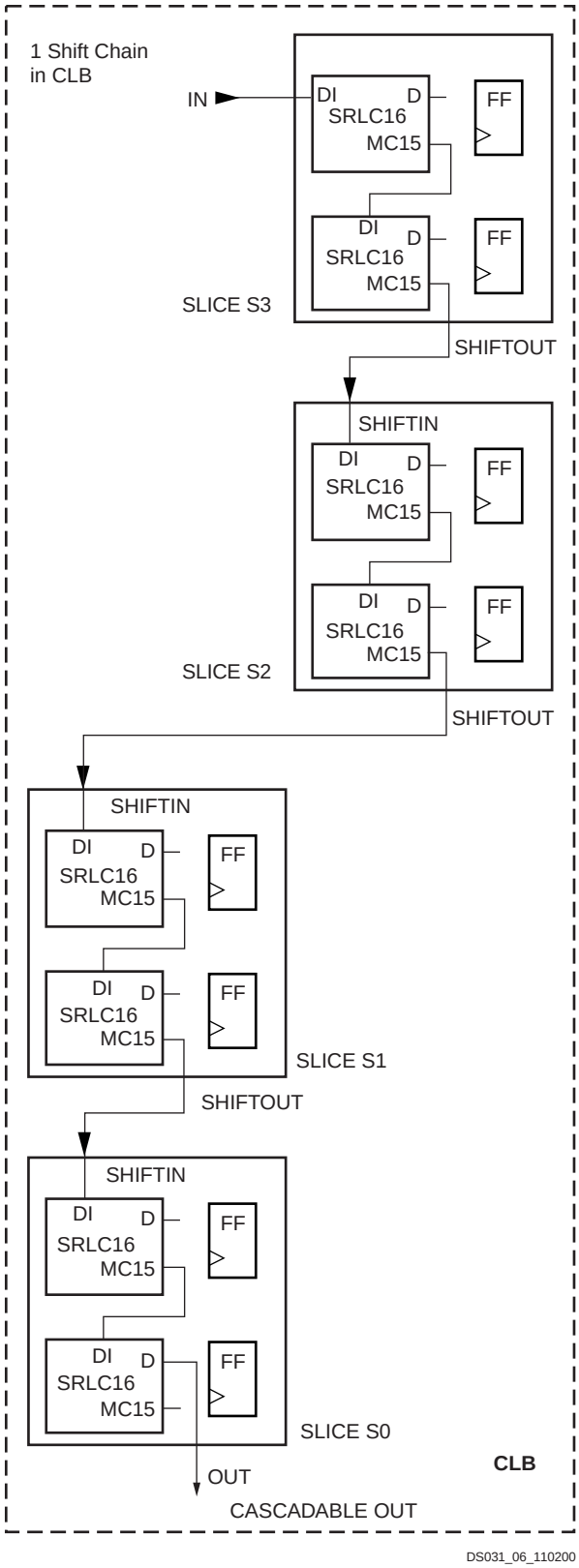


Figure 28: **Cascadable Shift Register**

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.

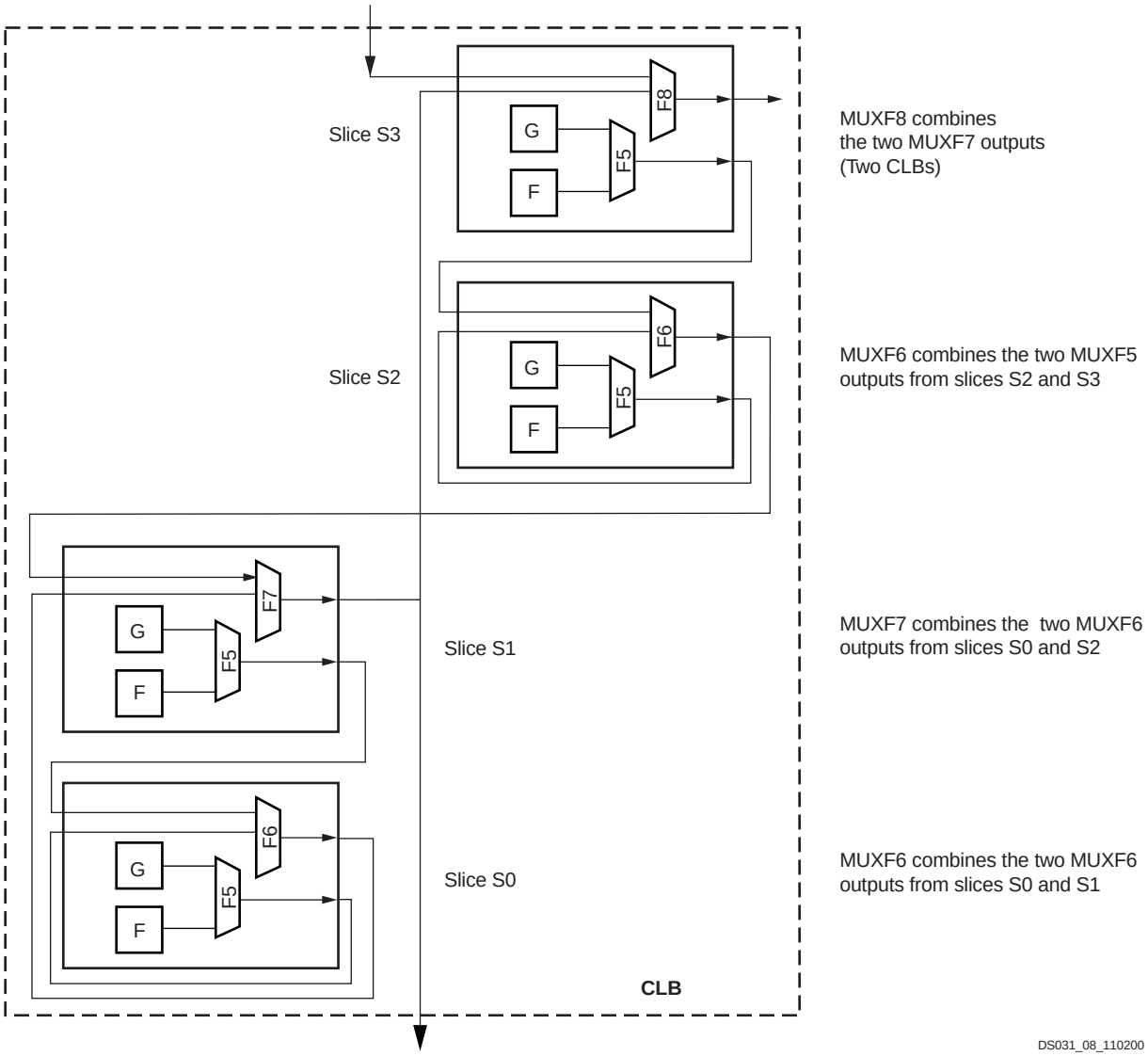


Figure 29: **MUXF5 and MUXFX multiplexers**

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.

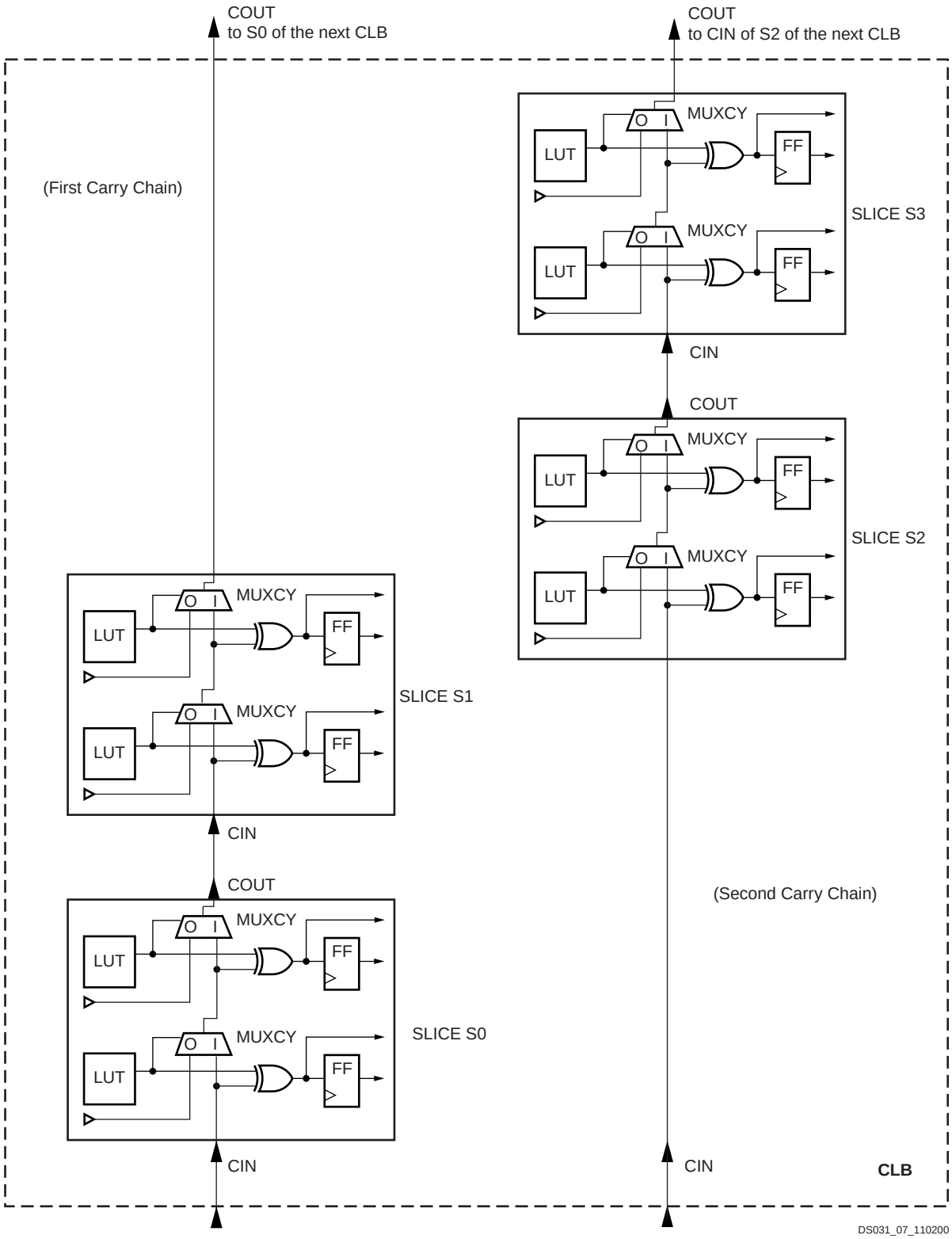


Figure 30: **Fast Carry Logic Path**

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.

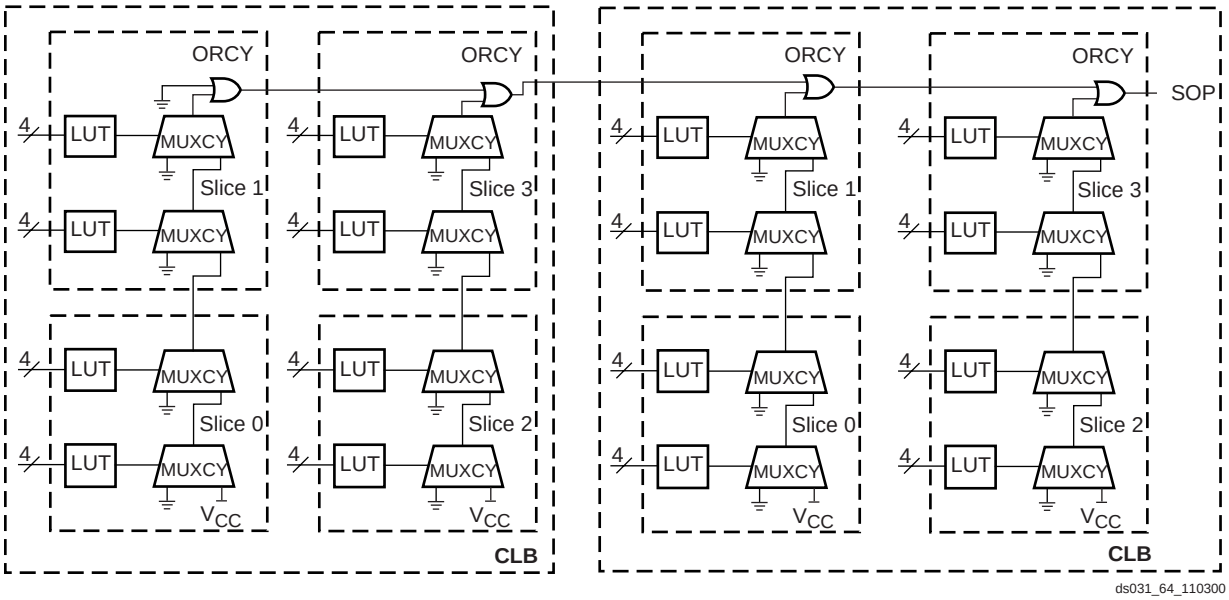


Figure 31: **Horizontal Cascade Chain**

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.

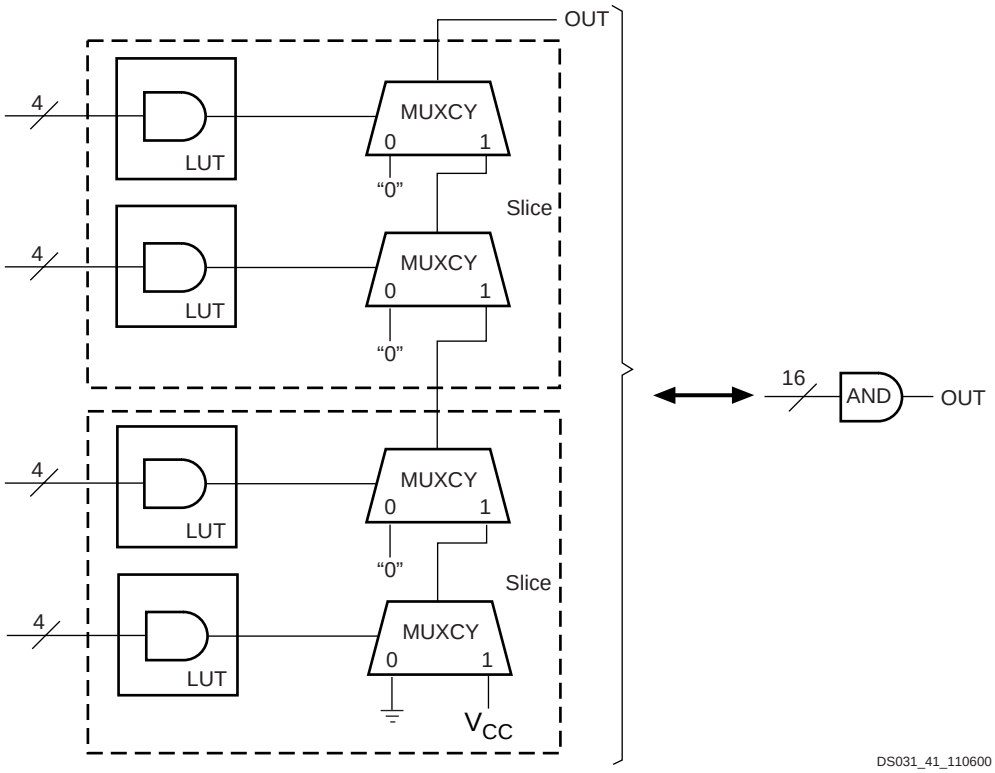


Figure 32: **Wide-Input AND Gate (16 Inputs)**

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.

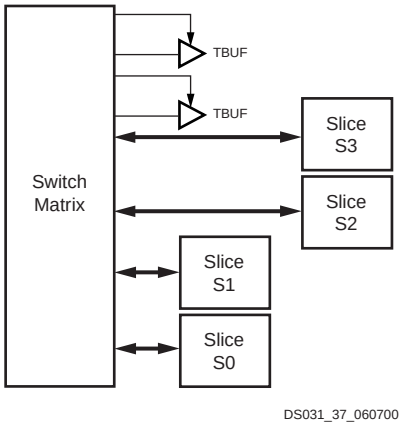


Figure 33: **Virtex-II Pro 3-State Buffers**

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

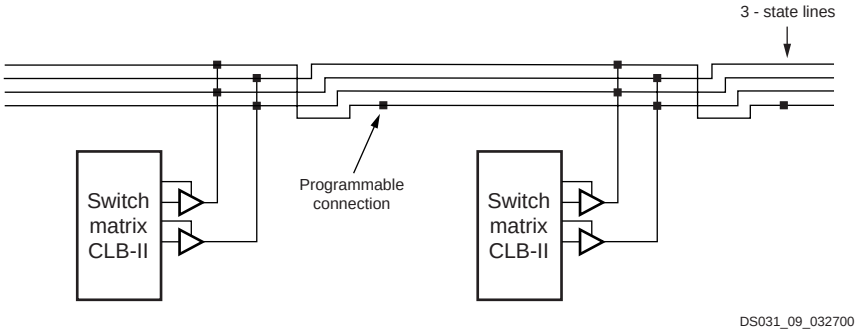


Figure 34: **3-State Buffer Connection to Horizontal Lines**

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 implemented 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.

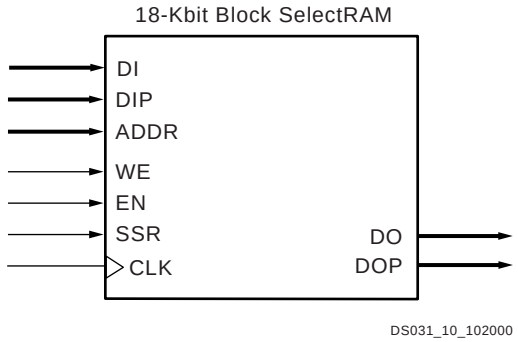


Figure 35: **18 Kb Block SelectRAM Memory in Single-Port Mode**

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.

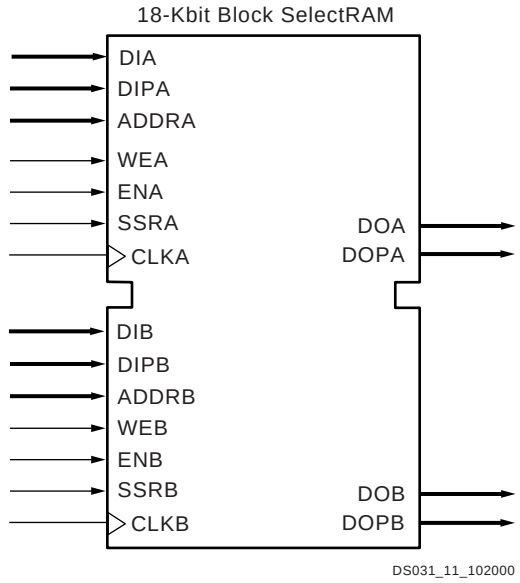


Figure 36: **18** Kb **Block SelectRAM in Dual-Port Mode**

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 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]

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.

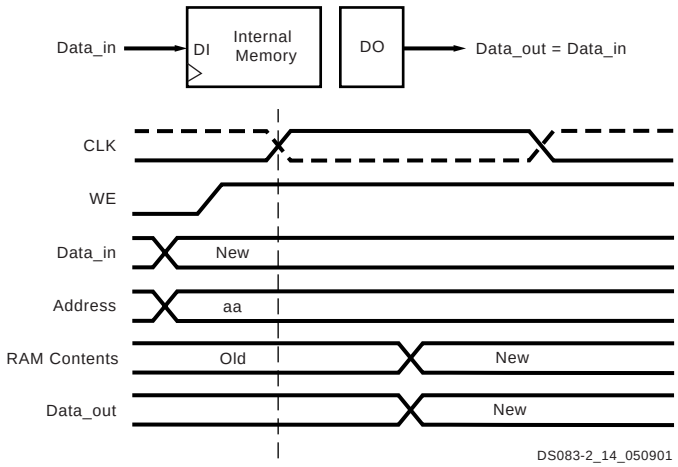


Figure 37: **WRITE_FIRST Mode**

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.

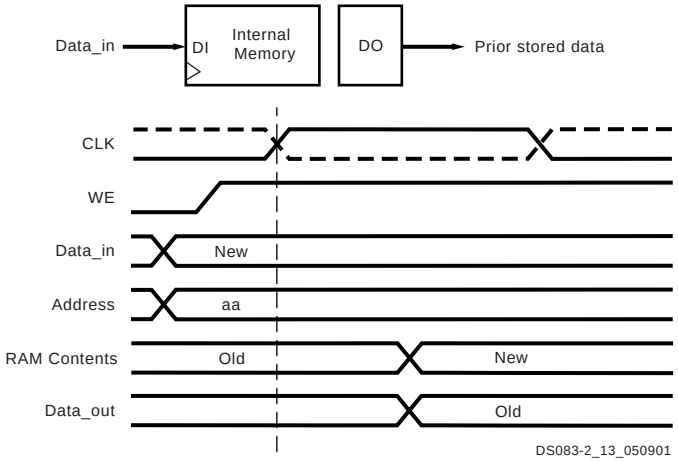


Figure 38: **READ_FIRST Mode**

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.

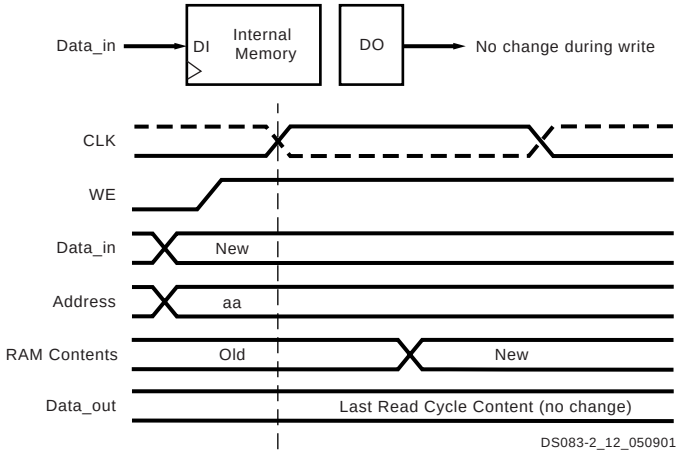


Figure 39: **NO_CHANGE Mode**

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.

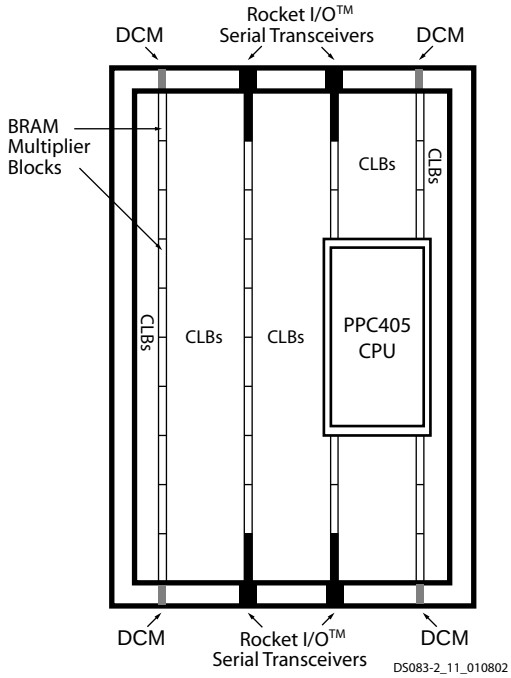


Figure 40: **XC2VP4 Block RAM Column Layout**

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.

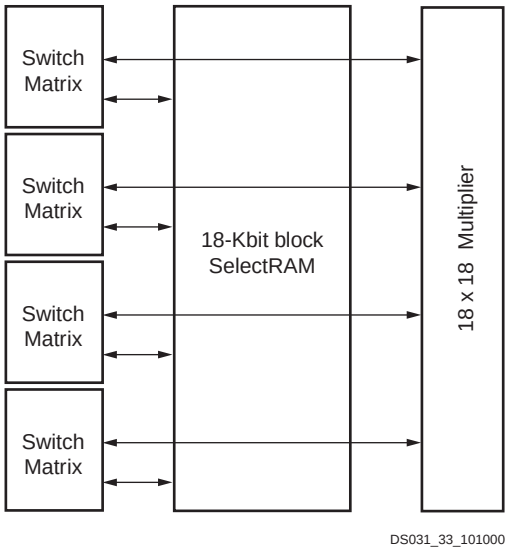


Figure 41: **SelectRAM and Multiplier Blocks**

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.

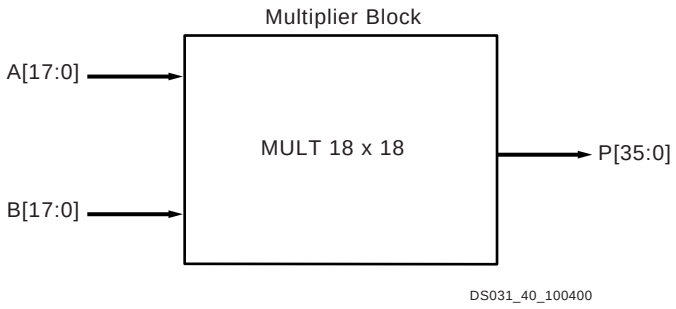


Figure 42: **Multiplier Block**

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.

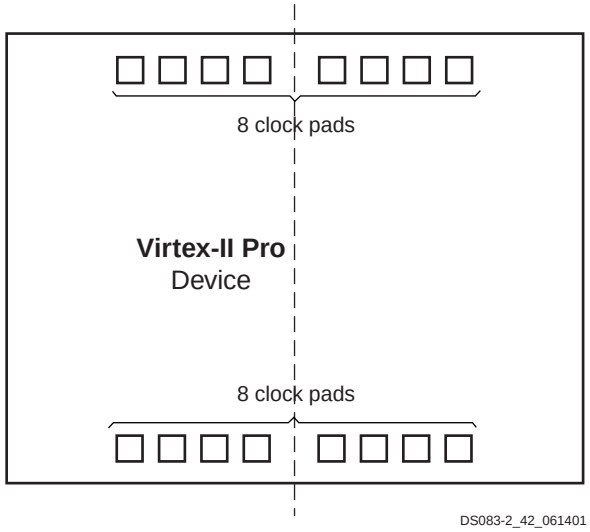


Figure 43: **Virtex-II Pro Clock Pads**

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.

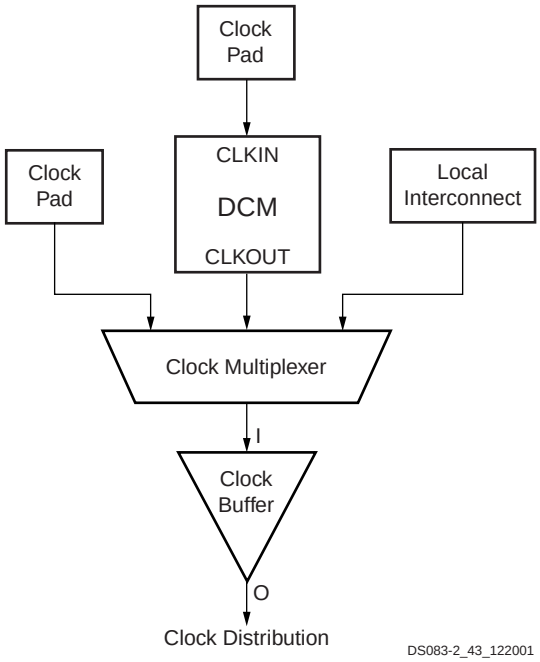


Figure 44: **Virtex-II Pro Clock Multiplexer Buffer Configuration**

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 floorplanning. (See the Virtex-II Pro *User Guide*.)

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

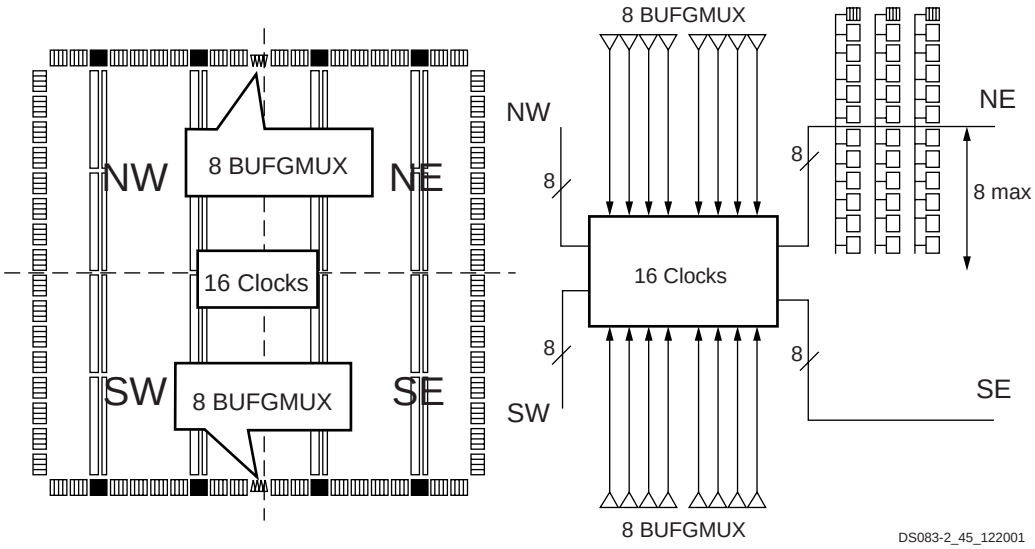


Figure 45: **Virtex-II Pro Clock Distribution**

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.

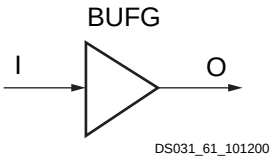


Figure 46: **Virtex-II Pro BUFG Function**

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 primitives. 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.

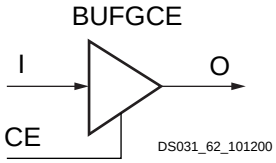


Figure 47: **Virtex-II Pro BUFGCE Function**

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 .

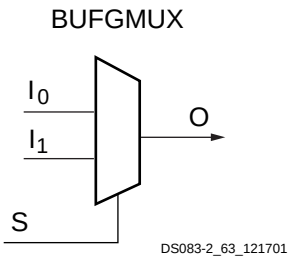


Figure 48: **Virtex-II Pro BUFGMUX Function**

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.

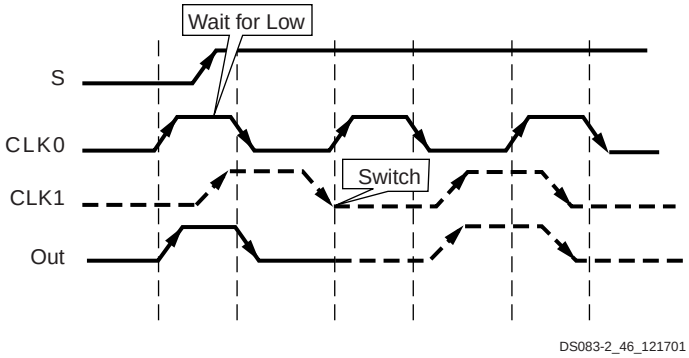


Figure 49: **Clock Multiplexer Waveform Diagram**

- 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.

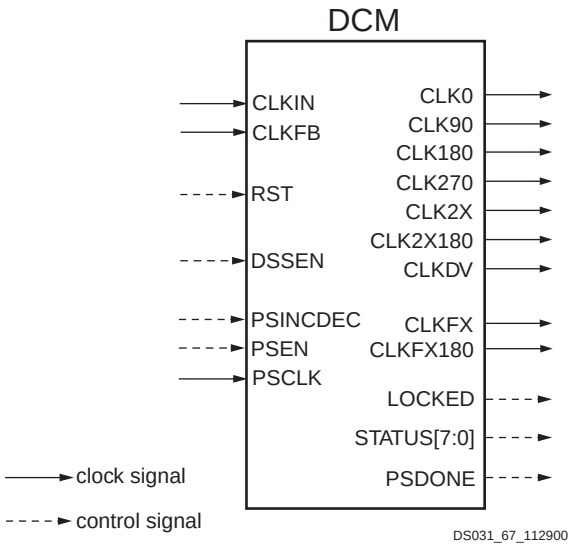


Figure 50: **Digital Clock Manager**

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:

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

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 ¼ 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 ± phase shift
PSCLK	In	Clock for phase shift
PSDONE	Out	Active when completed

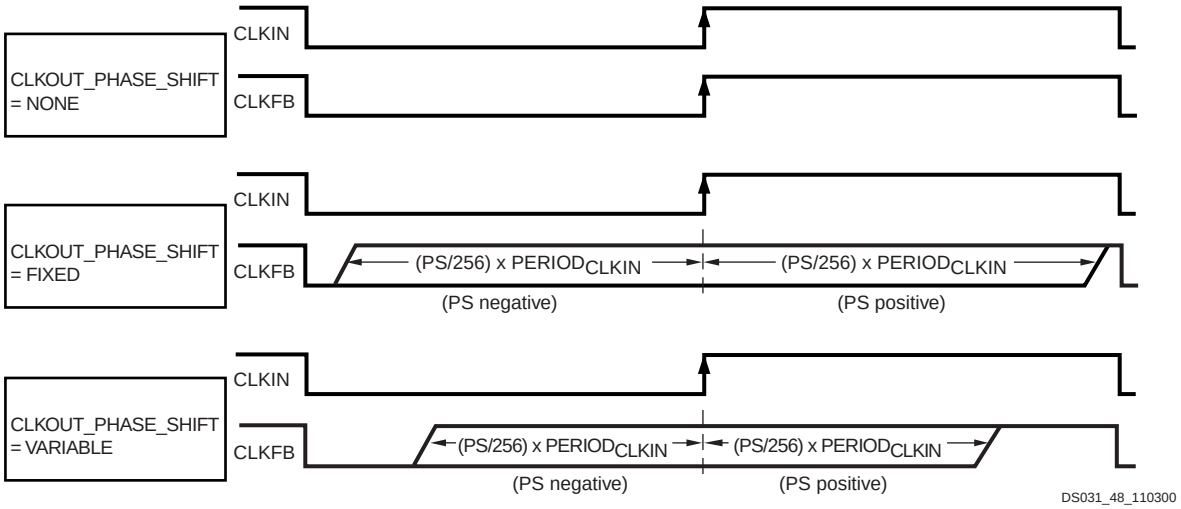


Figure 51: **Fine-Phase Shifting Effects**

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:

Phase Shift (ns) = (PHASE_SHIFT/256) * $PERIOD_{CLKIN}$

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) = ± FINE_SHIFT_RANGE

Absolute range (variable mode) = ± 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 $PERIOD_{CLKIN}$ = 2 * FINE_SHIFT_RANGE, then PHASE_SHIFT in fixed mode is limited to ± 128, and in variable mode it is limited to ± 64.
- If $PERIOD_{CLKIN}$ = FINE_SHIFT_RANGE, then PHASE_SHIFT in fixed mode is limited to ± 255, and in variable mode it is limited to ± 128.
- If $PERIOD_{CLKIN} \leq 0.5$ * 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.)

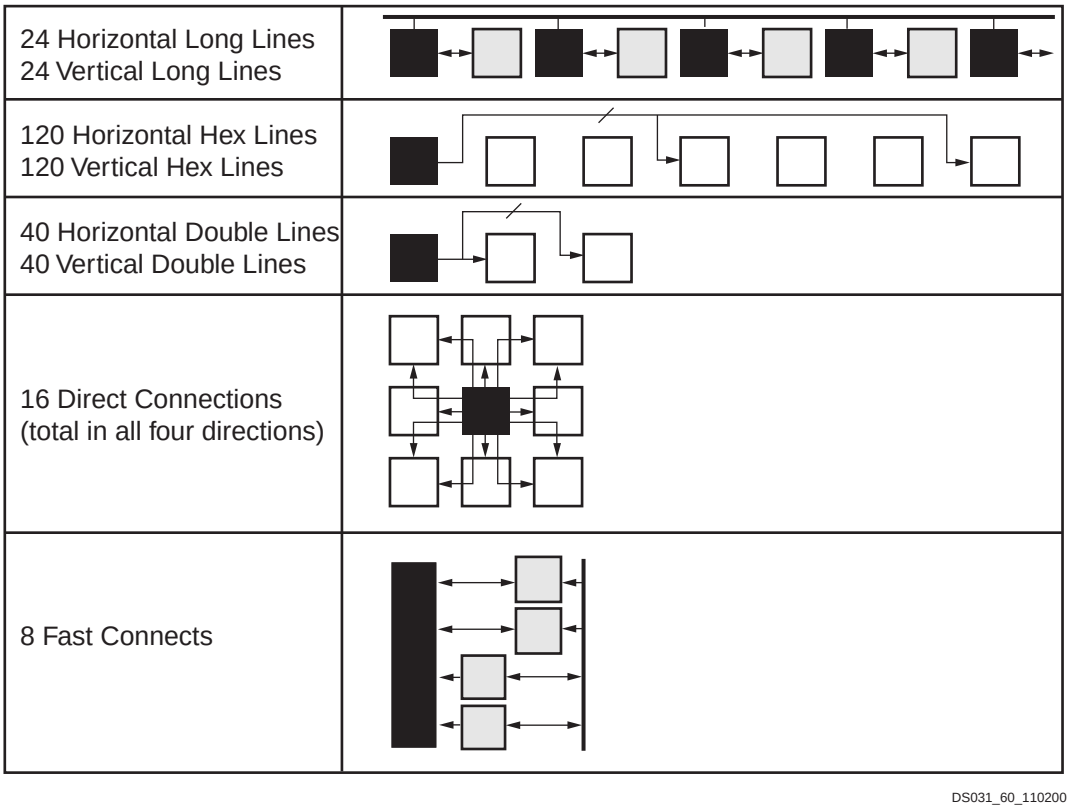


Figure 52: **Hierarchical Routing Resources**

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: Functional Description (Module 2)**
- **Virtex-II Pro™ Platform FPGAs: DC and Switching Characteristics (Module 3)**
- **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:
1. 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.
2. 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.
3. 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.
4. I/O input pin voltage, including overshoot, must not exceed 3.45V, either on a continuous or transient basis.
5. 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 °C to +85°C	Commercial	1.425	1.575	V
	Internal supply voltage relative to GND, T_J = –40°C to +100°C	Industrial	1.425	1.575	V
V_{CCAUX}(1)	Auxiliary supply voltage relative to GND, T_J = 0 °C to +85°C	Commercial	2.375	2.625	V
	Auxiliary supply voltage relative to GND, T_J = –40°C to +100°C	Industrial	2.375	2.625	V
V_{CCO}(2)	Supply voltage relative to GND, T_J = 0 °C to +85°C	Commercial	1.2	3.45(4)	V
	Supply voltage relative to GND, T_J = –40°C to +100°C	Industrial	1.2	3.45(4)	V
V_{BATT}(3)	Battery voltage relative to GND, T_J = 0 °C to +85°C	Commercial	1.0	2.63	V
	Battery voltage relative to GND, T_J = –40°C to +100°C	Industrial	1.0	2.63	V
$V_{CCAUXRX}$, $V_{CCAUXTX}$	Auxilliary supply voltage relative to GNDA	Commercial	2.375	2.625	V
	Auxilliary 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} = 0V, V_{CCO} = 3.3V (sample tested)	All	Note (1)			mA
I_{RPD}	Pad pull-down (when selected) @ V_{in} = 3.6V (sample tested)	All	Note (1)			mA
$I_{CCAUXTX}$	Operating $V_{CCAUXTX}$ 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 µs 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 chosen 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
LVCMOS33	0.0	0.8	2.0	V_{CCO}	0.4	V_{CCO} – 0.4	24	–24
LVCMOS25	–0.5	0.7	1.7	V_{CCO} + 0.4	0.4	V_{CCO} – 0.4	24	–24
LVCMOS18	–0.5	20% V_{CCO}	70% V_{CCO}	V_{CCO} + 0.4	0.4	V_{CCO} – 0.45	16	–16
LVCMOS15	–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:

1. 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.
2. 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			
Description	**Symbol**	**-8**	**-7**	**-6**	**Units**
Setup and Hold Relative to Clock (CPMC405CLOCK)					
Device Control Register Bus control inputs	$T_{PCCK\_}DCR/T_{PCKC\_}DCR$				ns, min
Device Control Register Bus data inputs	$T_{PDCK\_}DCR/T_{PCKD\_}DCR$				ns, min
Clock and Power Management control inputs	$T_{PCCK\_}CPM/T_{PCKC\_}CPM$				ns, min
Reset control inputs	$T_{PCCK\_}RST/T_{PCKC\_}RST$				ns, min
Debug control inputs	$T_{PCCK\_}DBG/T_{PCKC\_}DBG$				ns, min
Trace control inputs	$T_{PCCK\_}TRC/T_{PCKC\_}TRC$				ns, min
External Interrupt Controller control inputs	$T_{PCCK\_}EIC/T_{PCKC\_}EIC$				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)***

Description	Symbol	Speed Grade			Units
		-8	-7	-6	
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**

Description	Symbol	Speed Grade			Units
		-8	-7	-6	
Setup and Hold Relative to Clock (PLBCLK)					
Processor Local Bus(ICU/DCU) control inputs	$T_{PCCK\_}PLB/T_{PCKC\_}PLB$				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**

Description	Symbol	Speed Grade			Units
		-8	-7	-6	
Setup and Hold Relative to Clock (JTAGC405TCK)					
JTAG control inputs	$T_{PCCK\_}JTAG/T_{PCKC\_}JTAG$				ns, min
JTAG reset input	$T_{PCCK\_}JTAGRST/$ $T_{PCKC\_}JTAGRST$				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 (BRAMDSOCMCLK)					
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					
BRAMDSOCMCLK minimum pulse width, high	T_{DPWH}				ns, min
BRAMDSOCMCLK 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 (BRAMISOCMCLK)					
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**

			All Speed Grades			
Description	**Symbol**	**Conditions**	**Min**	**Typ**	**Max**	**Units**
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:
1. REFCLK frequency is typically 1/20 of serial data rate.

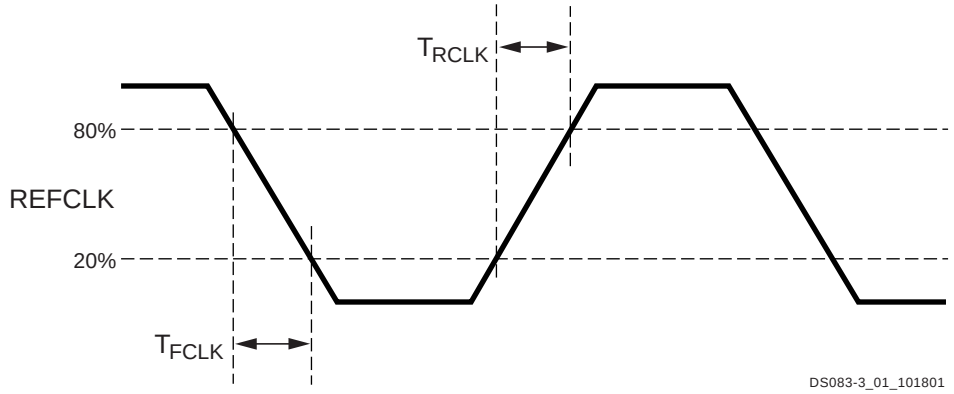


Figure 1: **Reference Clock (REFCLK) Timing Parameters**

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	RXUSRCLK 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

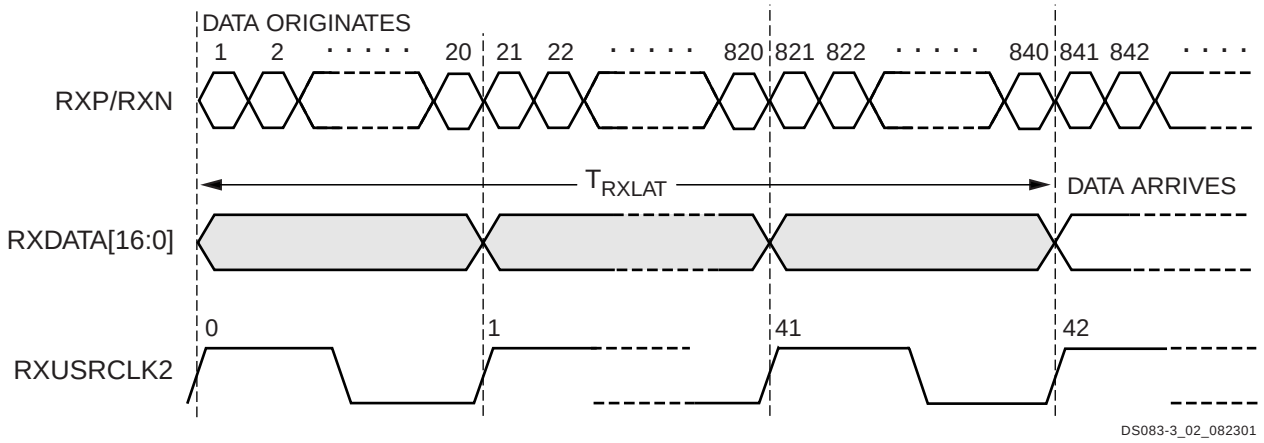


Figure 2: **Receive Latency (Maximum)**

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	TXUSRCLK 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

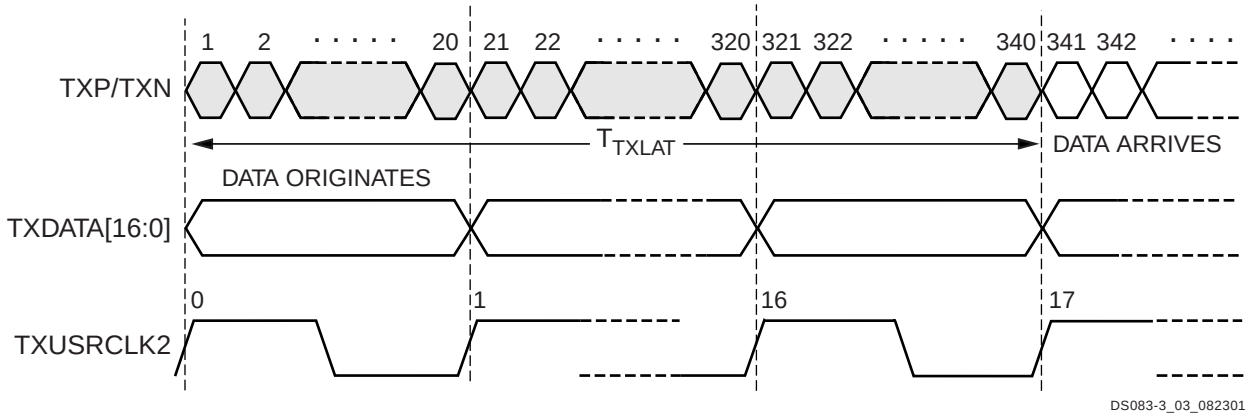


Figure 3: **Transmit Latency (Maximum, Including CRC)**

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	$T_{GCCK\_}RRST/T_{GCKC\_}RRST$				ns, min
RXPOLARITY control input	$T_{GCCK\_}RPOL/T_{GCKC\_}RPOL$				ns, min
ENCHANSYNC control input	$T_{GCCK\_}ECSY/T_{GCKC\_}ECSY$				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
RXCRCERR 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	$T_{GCCK\_}CFGEN/T_{GCKC\_}CFGEN$				ns, min
TXBYPASS8B10B control inputs	$T_{GCCK\_}TBYP/T_{GCKC\_}TBYP$				ns, min
TXFORCECRCERR control input	$T_{GCCK\_}TCRCE/T_{GCKC\_}TCRCE$				ns, min
TXPOLARITY control input	$T_{GCCK\_}TPOL/T_{GCKC\_}TPOL$				ns, min
TXINHIBIT control inputs	$T_{GCCK\_}TINH/T_{GCKC\_}TINH$				ns, min
LOOPBACK control inputs	$T_{GCCK\_}LBK/T_{GCKC\_}LBK$				ns, min
TXRESET control input	$T_{GCCK\_}TRST/T_{GCKC\_}TRST$				ns, min
TXCHARISK control inputs	$T_{GCCK\_}TKCH/T_{GCKC\_}TKCH$				ns, min
TXCHARDISPMODE control inputs	$T_{GCCK\_}TCDM/T_{GCKC\_}TCDM$				ns, min
TXCHARDISPVAL control inputs	$T_{GCCK\_}TCDV/T_{GCKC\_}TCDV$				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_{IOPICK}/T_{IOICKP}	All				ns, min
Pad, with delay	$T_{IOPICKD}/T_{IOICKPD}$	XC2VP2				ns, max
		XC2VP4				ns, max
		XC2VP7				ns, max
		XC2VP20				ns, max
		XC2VP50				ns, max
ICE input	$T_{IOICECK}/T_{IOCKICE}$	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
	$T_{ILVCMOS33}$	LVCMOS33				ns
	$T_{ILVCMOS25}$	LVCMOS25				ns
	$T_{ILVCMOS18}$	LVCMOS18				ns
	$T_{ILVCMOS15}$	LVCMOS15				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)***

			Speed Grade			
Description	**Symbol**	**Standard**	**–8**	**–7**	**–6**	**Units**
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 LVTTL 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)***

Description	Symbol	Speed Grade			Units
		–8	–7	–6	
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**

Description	Symbol	Standard	Speed Grade			Units
			–8	–7	–6	
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)***

			Speed Grade			
Description	**Symbol**	**Standard**	**–8**	**–7**	**–6**	**Units**
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_IIII}	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
	$T_{OHSTL\_IIII\_18}$	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
	$T_{OLVCMOS33\_S2}$	LVCMOS33, 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
	$T_{OLVCMOS33\_F2}$	LVCMOS33, 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)***

			Speed Grade			
Description	**Symbol**	**Standard**	**–8**	**–7**	**–6**	**Units**
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
	$T_{OLVCMOS25\_S2}$	LVCMOS25, 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
	$T_{OLVCMOS25\_F2}$	LVCMOS25, 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
	$T_{OLVCMOS18\_S2}$	LVCMOS18, 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
	$T_{OLVCMOS18\_F2}$	LVCMOS18, 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
	$T_{OLVCMOS15\_S2}$	LVCMOS15, 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)***

			Speed Grade			
Description	**Symbol**	**Standard**	**–8**	**–7**	**–6**	**Units**
Standard-specific adjustments for output delays terminating at pads (based on standard capacitive load, Csl) *(continued)*	$T_{OLVCMOS15\_F2}$	LVCMOS15, 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}	GTLP_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} (Typ) (2)
LVTTL	0	3	1.4	–
LVCMOS33	0	3.3	1.65	–
LVCMOS25	0	2.5	1.25	–
LVCMOS18	0	1.8	0.9	–
LVCMOS15	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} (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
LVCMOS33	35
LVCMOS25	35
LVCMOS18	35
LVCMOS15	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 (1.5Vand 1.8V	20
HSTL Class III (1.5Vand 1.8V	20
HSTL Class IV1.5Vand 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**

		Speed Grade			
Description	**Symbol**	**–8**	**–7**	**–6**	**Units**
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**

Description	Symbol	Speed Grade			Units
		–8	–7	–6	
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**

Description	Symbol	Speed Grade			Units
		–8	–7	–6	
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**
LVCMOS25 Global Clock Input to Output Delay using Output Flip-flop, 12 mA, Fast Slew Rate, *with* DCM. For data *output* 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	$T_{ICKOFDCM}$	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**
LVCMOS25 Global Clock Input to Output Delay using Output Flip-flop, 12 mA, Fast Slew Rate, *without* DCM. For data *output* 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 following 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			
Description	**Symbol**	**Constraints**	**-8**	**-7**	**-6**	**Units**
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:
1. “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)**

Virtex-II Pro™ Platform FPGAs: Pinout Information

DS083-4 (v1.0) January 31, 2002

Advance Product Specification

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).
AVCCAUXTX#	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	Negitive 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	Negitive 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	GNDA6	C6
N/A	GNDA6	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	GNDA7	C11
N/A	GNDA7	C11
N/A	RXPPAD7	A12

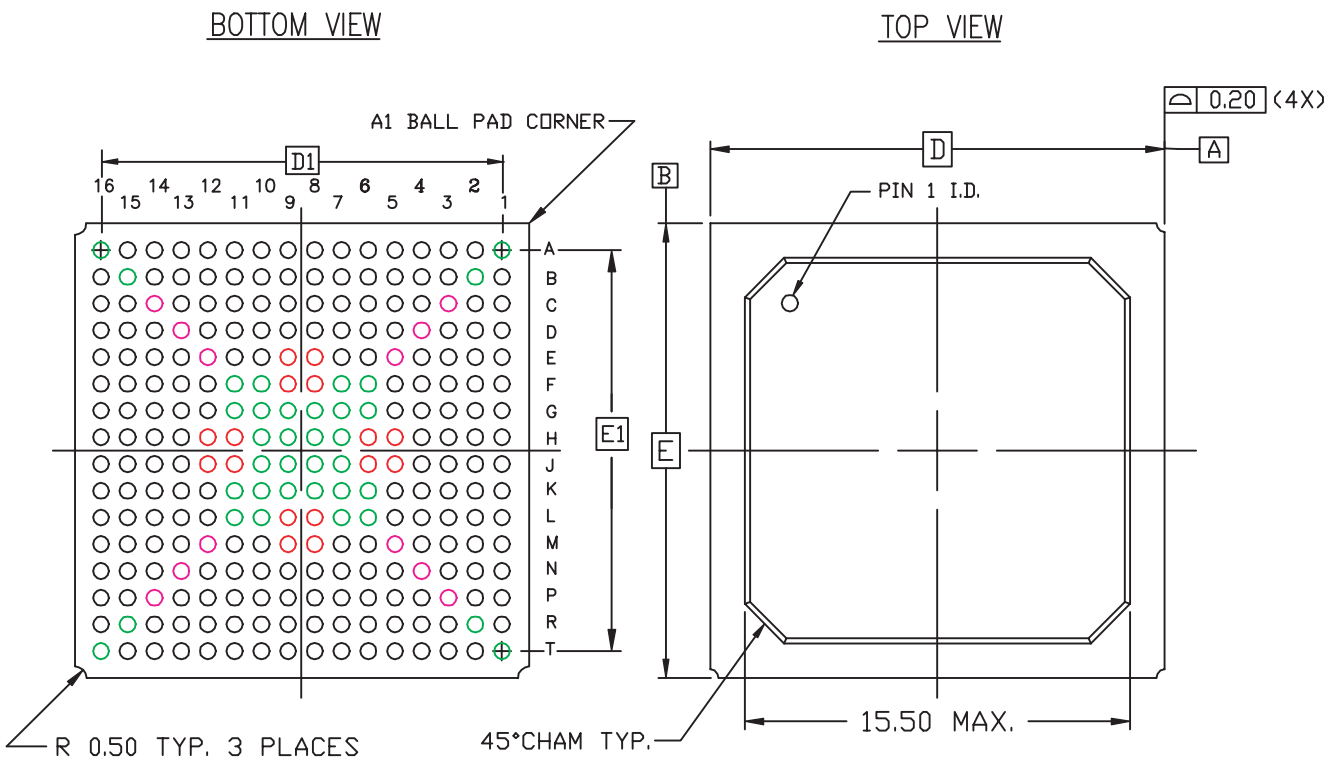
Table 6: **FG256 — XC2VP2 and XC2VP4**

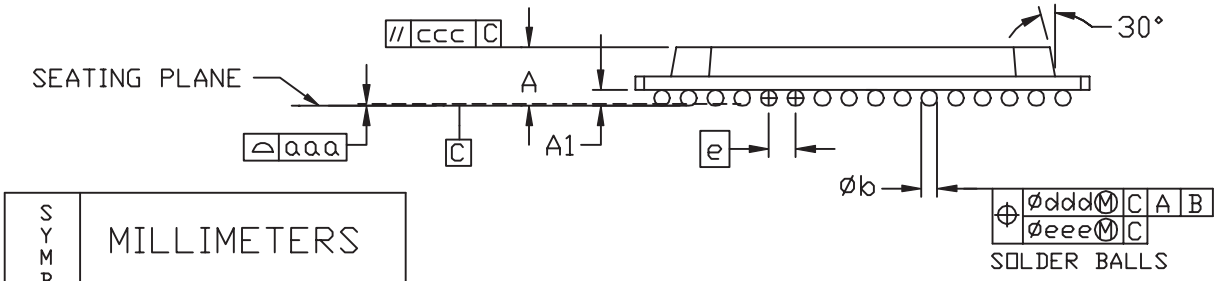
Bank	Pin Description	Pin Number
N/A	RXNPAD7	A13
N/A	VTRXPAD7	B12
N/A	AVCCAUXRX7	B13
N/A	AVCCAUXRX18	R13
N/A	VTRXPAD18	R12
N/A	RXNPAD18	T13
N/A	RXPPAD18	T12
N/A	GNDA18	P11
N/A	GNDA18	P11
N/A	TXPPAD18	T11
N/A	TXNPAD18	T10
N/A	VTTXPAD18	R10
N/A	AVCCAUXTX18	R11
N/A	AVCCAUXRX19	R7
N/A	VTRXPAD19	R6
N/A	RXNPAD19	T7
N/A	RXPPAD19	T6
N/A	GNDA19	P6
N/A	GNDA19	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		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			0.20
ccc			0.35
ddd			0.30
eee			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	GNDA4	C6	NC	NC	
N/A	GNDA4	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	GNDA6	C9			
N/A	GNDA6	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	GNDA7	C14			
N/A	GNDA7	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	GNDA9	C17	NC	NC	
N/A	GNDA9	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	GNDA16	Y17	NC	NC	
N/A	GNDA16	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	GNDA18	Y14			
N/A	GNDA18	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	GNDA19	Y9			
N/A	GNDA19	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	GNDA21	Y6	NC	NC	
N/A	GNDA21	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			

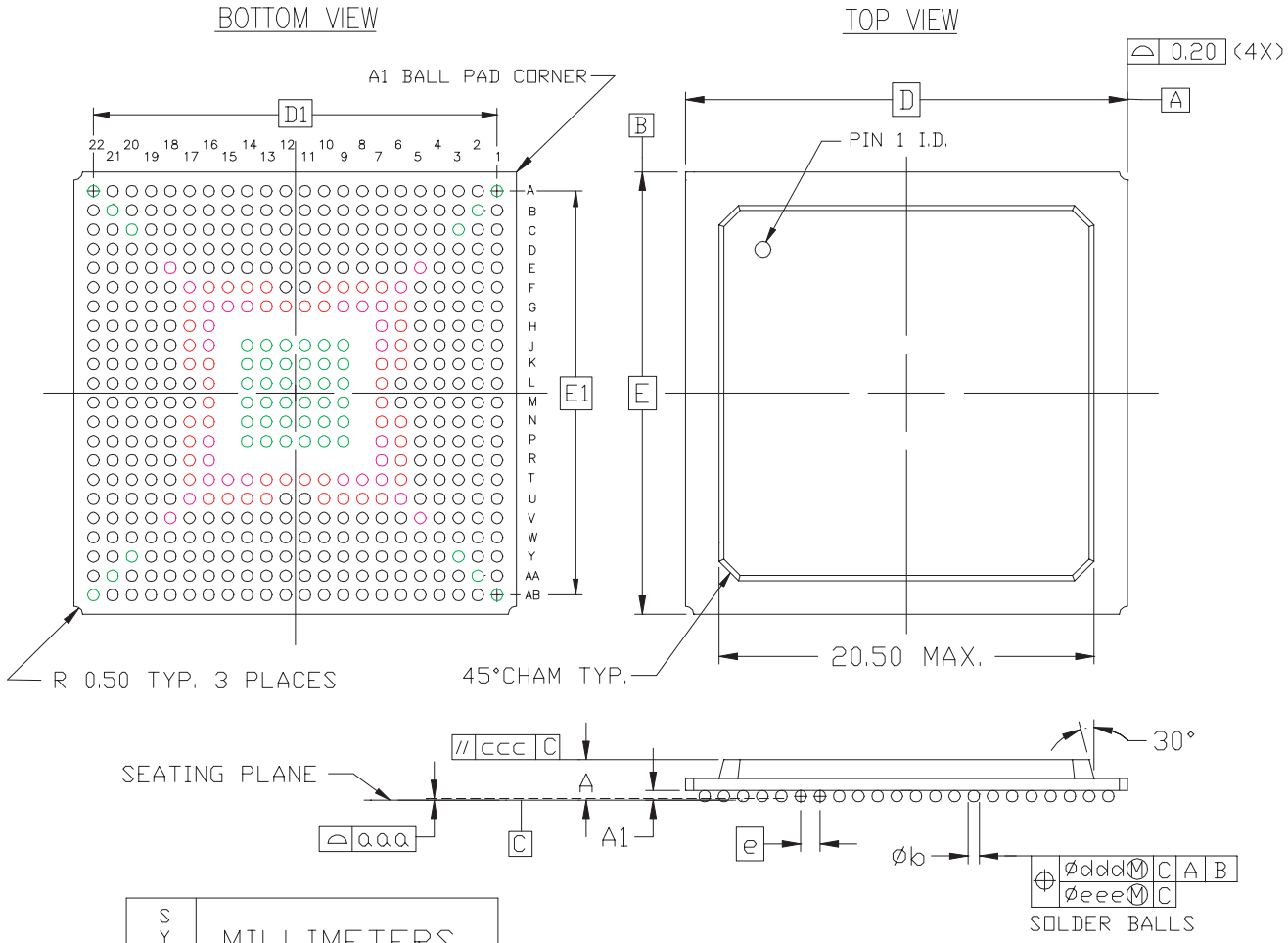
Table 7: **FG456 — XC2VP2, XC2VP4, and XC2VP7**

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)



SYMBOL	MILLIMETERS		
	MIN.	NOM.	MAX.
A		2.20	2.60
A_1	0.40	0.50	0.60
D/E	23.00 BSC		
D_1/E_1	21.00 REF		
e	1.00 BSC		
øb	0.50	0.60	0.70
aaa			0.20
ccc			0.35
ddd			0.30
eee			0.10
M	22		

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 AAJ–1 (DEPOPULATED)

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	GNDA7	C11			
N/A	GNDA7	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	GNDA9	C5	NC	NC	
N/A	GNDA9	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	GNDA16	AD5	NC	NC	
N/A	GNDA16	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	GNDA18	AD11			
N/A	GNDA18	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	GNDA4	C22	NC	NC	
N/A	GNDA4	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	GNDA6	C16			
N/A	GNDA6	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	GNDA19	AD16			
N/A	GNDA19	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	GNDA21	AD22	NC	NC	
N/A	GNDA21	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			

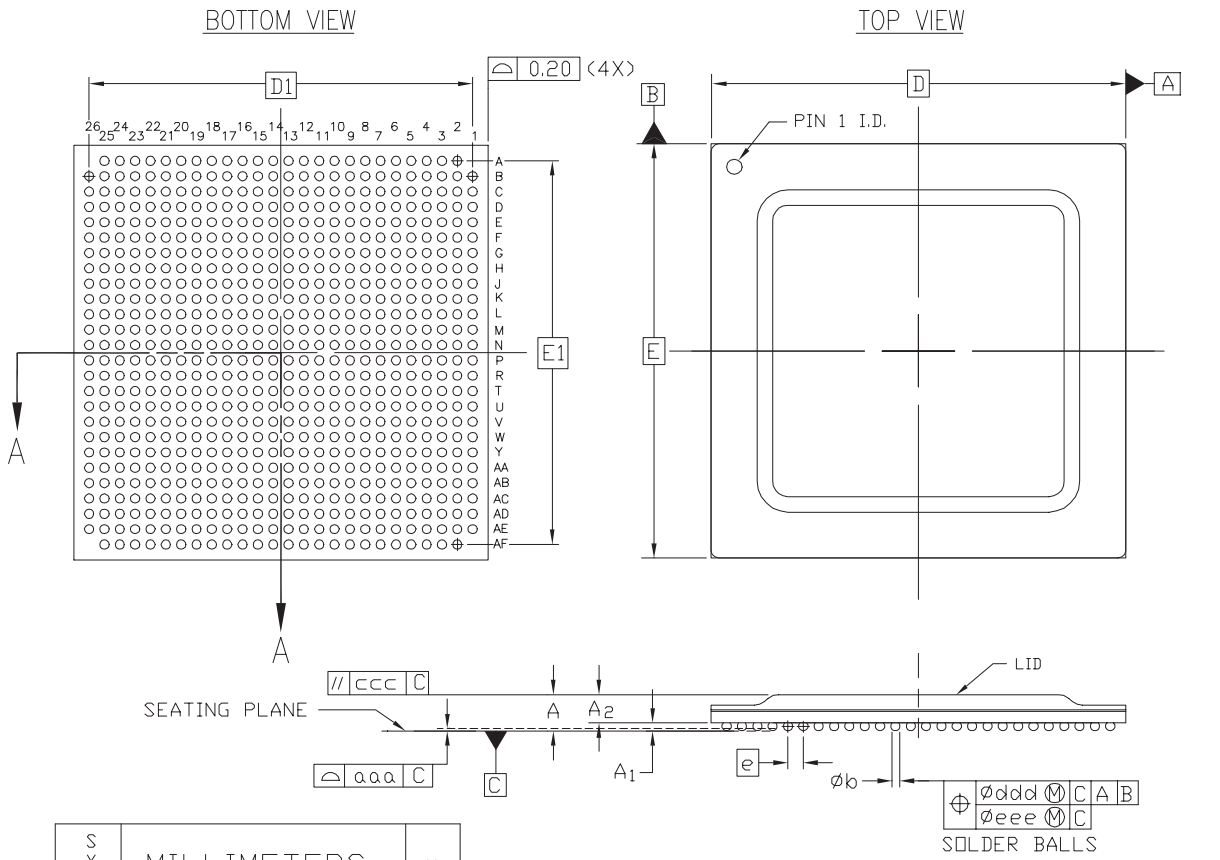
Table 8: **FF672 — XC2VP2, XC2VP4, and XC2VP7**

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)



SYMBOL	MILLIMETERS			NOTE
	MIN.	NOM.	MAX.	
A		2.45	2.65	
A_1	0.40	0.50	0.60	
A_2			2.15	
D/E	27.00 BASIC			
D_1/E_1	25.00 REF			
e	1.00 BASIC			
øb	0.50	0.60	0.70	
aaa			0.20	
ccc			0.35	
ddd			0.30	
eee			0.10	
M	26			2

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–AAL–1 (DEPOPULATED)

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	GNDA4	C25		
N/A	GNDA4	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	GNDA6	C19		
N/A	GNDA6	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	GNDA7	C12		
N/A	GNDA7	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	GNDA9	C6		
N/A	GNDA9	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		

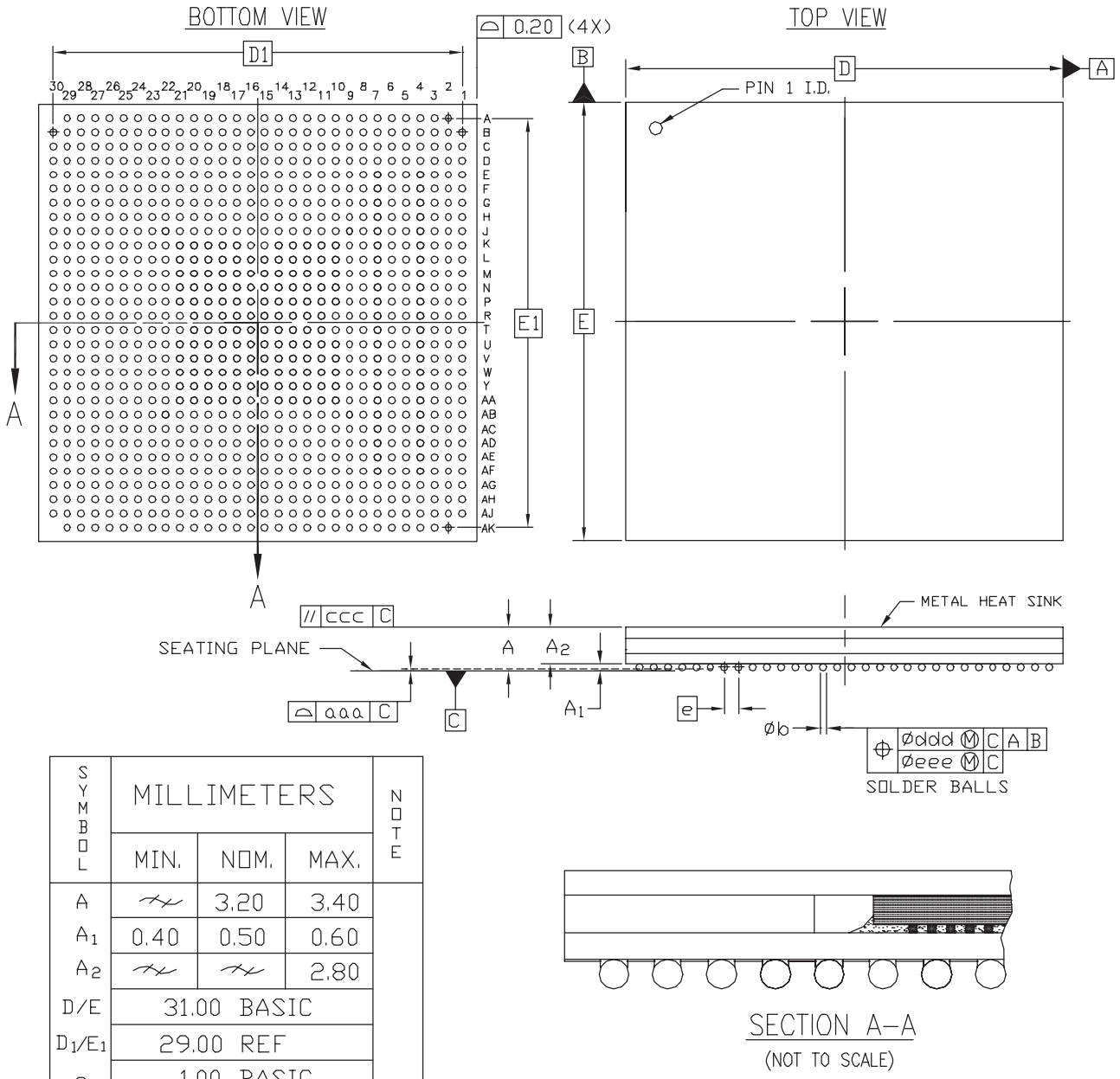
Table 9: **FF896 — XC2VP7 and XC2VP20**

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)



SYMBOL	MILLIMETERS			NOTE
	MIN.	NOM.	MAX.	
A		3.20	3.40	
A_1	0.40	0.50	0.60	
A_2			2.80	
D/E	31.00 BASIC			
D_1/E_1	29.00 REF			
e	1.00 BASIC			
øb	0.50	0.60	0.70	
aaa			0.20	
ccc			0.35	
ddd			0.30	
eee			0.10	
M	30			2

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	GNDA2	C30	NC	
N/A	GNDA2	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	GNDA4	C27		
N/A	GNDA4	C27		
N/A	RXPPAD4	A27		
N/A	RXNPAD4	A26		
N/A	VTRXPAD4	B27		
N/A	AVCCAUXRX4	B26		
N/A	AVCCAUXTX5	B24	NC	
N/A	VTTXPAD5	B25	NC	
N/A	TXNPAD5	A25	NC	
N/A	TXPPAD5	A24	NC	
N/A	GNDA5	C23	NC	
N/A	GNDA5	C23	NC	
N/A	RXPPAD5	A23	NC	
N/A	RXNPAD5	A22	NC	
N/A	VTRXPAD5	B23	NC	
N/A	AVCCAUXRX5	B22	NC	
N/A	AVCCAUXTX6	B20		
N/A	VTTXPAD6	B21		
N/A	TXNPAD6	A21		
N/A	TXPPAD6	A20		
N/A	GNDA6	C20		
N/A	GNDA6	C20		
N/A	RXPPAD6	A19		
N/A	RXNPAD6	A18		
N/A	VTRXPAD6	B19		
N/A	AVCCAUXRX6	B18		
N/A	AVCCAUXTX7	B16		
N/A	VTTXPAD7	B17		
N/A	TXNPAD7	A17		
N/A	TXPPAD7	A16		
N/A	GNDA7	C15		
N/A	GNDA7	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	GNDA8	C12	NC	
N/A	GNDA8	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	GNDA9	C8		
N/A	GNDA9	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	GNDA11	C5	NC	
N/A	GNDA11	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	GNDA14	AM5	NC	
N/A	GNDA14	AM5	NC	
N/A	TXPPAD14	AP4	NC	
N/A	TXNPAD14	AP5	NC	
N/A	VTTXPAD14	AN5	NC	
N/A	AVCCAUXTX14	AN4	NC	
N/A	AVCCAUXRX16	AN6		
N/A	VTRXPAD16	AN7		
N/A	RXNPAD16	AP6		
N/A	RXPPAD16	AP7		
N/A	GNDA16	AM8		
N/A	GNDA16	AM8		
N/A	TXPPAD16	AP8		
N/A	TXNPAD16	AP9		
N/A	VTTXPAD16	AN9		
N/A	AVCCAUXTX16	AN8		
N/A	AVCCAUXRX17	AN10	NC	
N/A	VTRXPAD17	AN11	NC	
N/A	RXNPAD17	AP10	NC	
N/A	RXPPAD17	AP11	NC	
N/A	GNDA17	AM12	NC	
N/A	GNDA17	AM12	NC	
N/A	TXPPAD17	AP12	NC	
N/A	TXNPAD17	AP13	NC	
N/A	VTTXPAD17	AN13	NC	
N/A	AVCCAUXTX17	AN12	NC	
N/A	AVCCAUXRX18	AN14		
N/A	VTRXPAD18	AN15		
N/A	RXNPAD18	AP14		
N/A	RXPPAD18	AP15		
N/A	GNDA18	AM15		
N/A	GNDA18	AM15		
N/A	TXPPAD18	AP16		
N/A	TXNPAD18	AP17		
N/A	VTTXPAD18	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	GNDA19	AM20		
N/A	GNDA19	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	GNDA20	AM23	NC	
N/A	GNDA20	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	GNDA21	AM27		
N/A	GNDA21	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	GNDA23	AM30	NC	

Table 10: **FF1152 — XC2VP20 and XC2VP50**

Bank	Pin Description	Pin Number	No Connects	
			XC2V P20	XC2V P50
N/A	GNDA23	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		

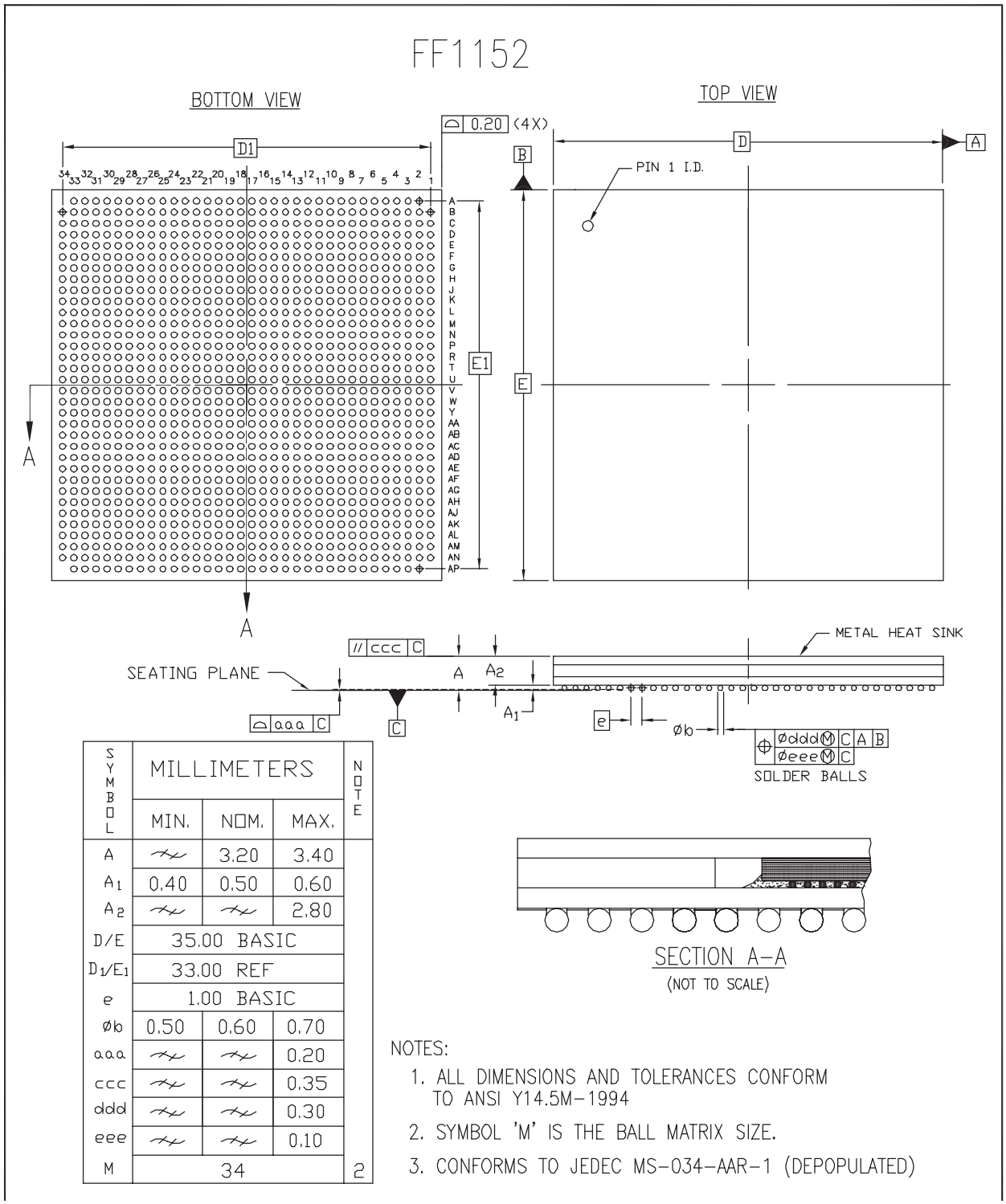
Table 10: **FF1152 — XC2VP20 and XC2VP50**

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)



SYMBOL	MILLIMETERS			NOTE
	MIN.	NOM.	MAX.	
A		3.20	3.40	
A_1	0.40	0.50	0.60	
A_2			2.80	
D/E	35.00 BASIC			
D_1/E_1	33.00 REF			
e	1.00 BASIC			
øb	0.50	0.60	0.70	
aaa			0.20	
ccc			0.35	
ddd			0.30	
eee			0.10	
M	34			2

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–AAR–1 (DEPOPULATED)

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	GNDA2	C34
N/A	GNDA2	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	GNDA4	C31
N/A	GNDA4	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	GNDA5	C27
N/A	GNDA5	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	GNDA6	C24
N/A	GNDA6	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	GNDA7	C16
N/A	GNDA7	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	GNDA8	C13
N/A	GNDA8	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	GNDA21	AU31
N/A	GNDA21	AU31
N/A	TXPPAD21	AW31
N/A	TXNPAD21	AW32
N/A	VTTXPAD21	AV32
N/A	AVCCAUXTX21	AV31
N/A	AVCCAUXRX23	AV33
N/A	VTRXPAD23	AV34
N/A	RXNPAD23	AW33
N/A	RXPPAD23	AW34
N/A	GNDA23	AU34
N/A	GNDA23	AU34
N/A	TXPPAD23	AW35
N/A	TXNPAD23	AW36
N/A	VTTXPAD23	AV36
N/A	AVCCAUXTX23	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

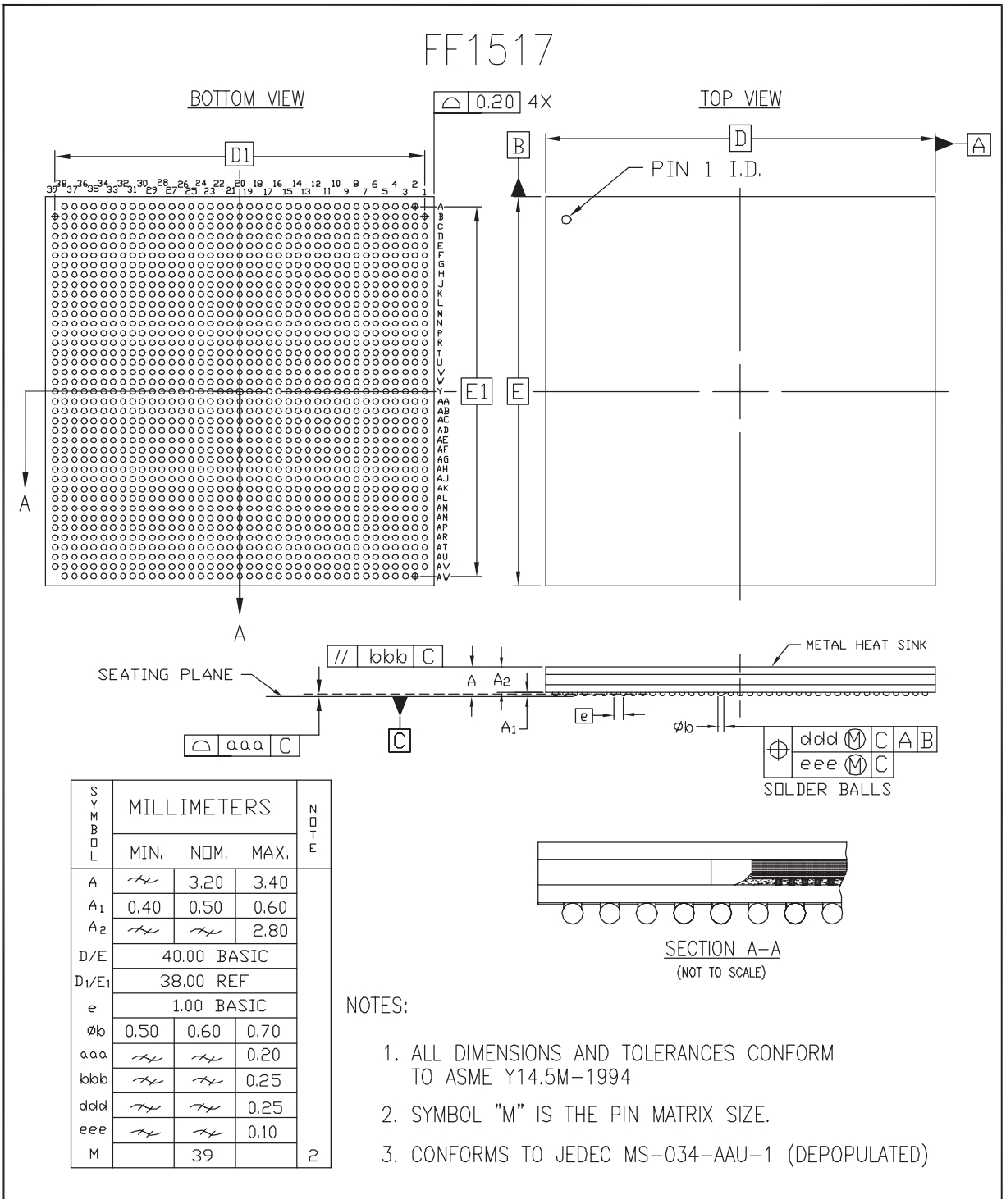
Table 11: **FF1517 — XC2VP50**

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)



SYMBOL	MILLIMETERS MIN.	NOM.	MAX.	NOTE
A		3.20	3.40	
A_1	0.40	0.50	0.60	
A_2			2.80	
D/E	40.00 BASIC			
D_1/E_1	38.00 REF			
e	1.00 BASIC			
øb	0.50	0.60	0.70	
aaa			0.20	
bbb			0.25	
ddd			0.25	
eee			0.10	
M		39		2

NOTES:

1. ALL DIMENSIONS AND TOLERANCES CONFORM TO ASME Y14.5M–1994
2. SYMBOL "M" IS THE PIN MATRIX SIZE.
3. CONFORMS TO JEDEC MS–034–AAU–1 (DEPOPULATED)

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	GNDA2	C27	NC	
N/A	GNDA2	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	GNDA4	C24		
N/A	GNDA4	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	GNDA6	C20		
N/A	GNDA6	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	GNDA7	C12		
N/A	GNDA7	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	GNDA9	C8		
N/A	GNDA9	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	GNDA11	C5	NC	
N/A	GNDA11	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	GNDA14	AJ5	NC	
N/A	GNDA14	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	GNDA16	AJ8		
N/A	GNDA16	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	GNDA18	AJ12		
N/A	GNDA18	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	GNDA19	AJ20		
N/A	GNDA19	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	GNDA21	AJ24		
N/A	GNDA21	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	GNDA23	AJ27	NC	
N/A	GNDA23	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		

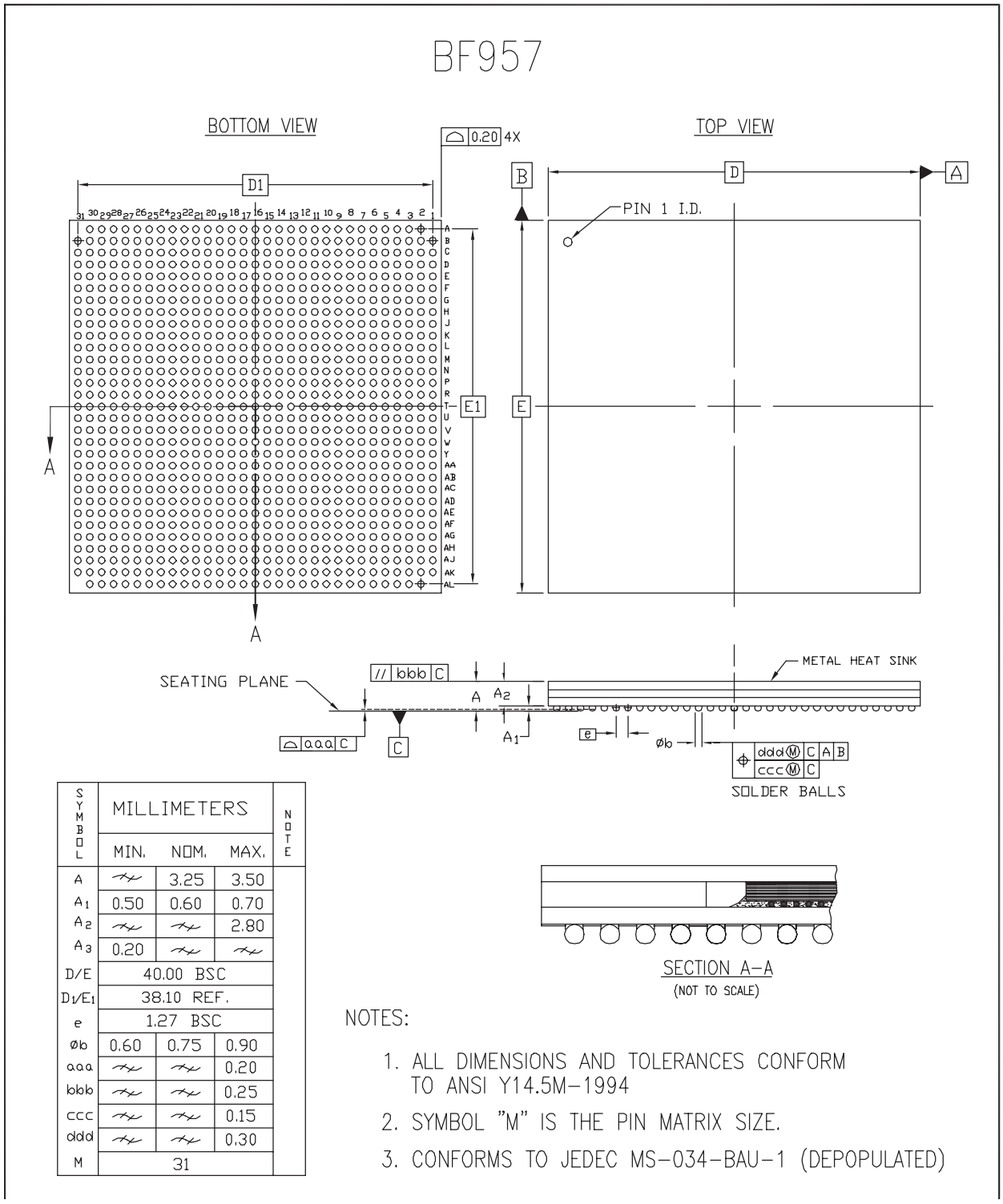
Table 12: **BF957 — XC2VP20 and XC2VP50**

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)



SYMBOL	MILLIMETERS MIN.	NOM.	MAX.	NOTE
A	---	3.25	3.50	
A_1	0.50	0.60	0.70	
A_2	---	---	2.80	
A_3	0.20	---	---	
D/E	40.00 BSC			
D_1/E_1	38.10 REF.			
e	1.27 BSC			
øb	0.60	0.75	0.90	
aaa	---	---	0.20	
bbb	---	---	0.25	
ccc	---	---	0.15	
ddd	---	---	0.30	
M	31			

NOTES:

1. ALL DIMENSIONS AND TOLERANCES CONFORM TO ANSI Y14.5M–1994
2. SYMBOL "M" IS THE PIN MATRIX SIZE.
3. CONFORMS TO JEDEC MS–034–BAU–1 (DEPOPULATED)

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

Volume 2: Virtex-II Pro™ Processor

Virtex-II Pro™ Platform FPGA Documentation

March 2002 Release

XILINX®

Volume 2(a): PPC405 User Manual

Virtex-II Pro™ Platform FPGA Documentation

March 2002 Release

XILINX®

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.
variable	Variable items are shown in italic.
<optional>	Optional items are shown in angle brackets.
$\overline{\text{ActiveLow}}$	An overbar indicates an active-low signal.
n	A decimal number.
0x*n*	A hexadecimal number.
0b*n*	A binary number.
(**r***n*)	The contents of GPR **r***n*.
(**r**A | 0)	The contents of the register **r**A, or 0 if the **r**A 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).
$OBJECT_{b,b,\ldots}$	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 *relative* 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 *absolute* 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 367
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 **mtcrf** 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 **wrteei** 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 OPCD 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 361 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 *without* 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 *with* 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.
$\|\|$	Concatenation
=, $\neq$	Equal, not-equal relations
<, >	Signed comparison relations
$\overset{u}{<}$, $\overset{u}{>}$	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 ≥ *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 **mtdcr** 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 *condition* 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.
instruction(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(*addr*, *n*)	The number of bytes represented by *n* at the location in main storage represented by *addr*.
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 **mfspr** or **mtspr** 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
$\neg$, – (unary minus)	Right to left
$\times$, $\div$	Left to right
+, –	Left to right
$\|\|$	Left to right
=, $\neq$, <, >, $\overset{u}{<}$, $\overset{u}{>}$	Left to right
$\wedge$, $\oplus$	Left to right
$\vee$	Left to right
$\leftarrow$	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 *n*
DBCR*n*	Debug-control register *n*
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 **r***n* (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
r*n*	Specifies GPR *n* (**r**15, 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.

Book-E	An version of the PowerPC architecture designed specifically for embedded applications.
cache block	Synonym for *cacheline*.
cacheline	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.
clear	To write a bit value of 0.
cache set	Synonym for *congruence class*.
congruence class	A collection of cachelines with the same index.
dirty	An indication that cache information is more recent than the copy in memory.
doubleword	Eight bytes, or 64 bits.
effective address	The untranslated memory address as seen by a program.
exception	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 *interrupt* to occur.
fill buffer	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.
flush	A cache or TLB operation that involves writing back a modified entry to memory, followed by an invalidation of the entry.
GB	Gigabyte, or one-billion bytes.
halfword	Two bytes, or 16 bits.
hit	For cache arrays and TLB arrays, an indication that requested information exists in the accessed array.
interrupt	The process of stopping the currently executing program so that an exception can be handled.
invalidate	A cache or TLB operation that causes an entry to be marked as invalid. An invalid entry can be subsequently replaced.
KB	Kilobyte, or one-thousand bytes.
line buffer	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.
little endian	A memory byte ordering where the address of an item corresponds to the least-significant byte.
logical address	Synonym for *effective address*.
MB	Megabyte, or one-million bytes.
memory	Collectively, cache memory and system memory.
miss	For cache arrays and TLB arrays, an indication that requested information does not exist in the accessed array.

OEA	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.
on chip	In system-on-chip implementations, this indicates on the same chip as the processor core, but external to the processor core.
pending	As applied to interrupts, this indicates that an exception occurred, but the interrupt is disabled. The interrupt occurs when it is later enabled.
physical address	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.
PLB	Processor local bus.
privileged mode	The operating mode typically used by system software. Privileged operations are allowed and software can access all registers and memory.
process	A program (or portion of a program) and any data required for the program to run.
problem state	Synonym for *user mode*.
real address	Synonym for *physical address*.
scalar	Individual data objects and instructions. Scalars are of arbitrary size.
set	To write a bit value of 1.
sticky	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.
string	A sequence of consecutive bytes.
supervisor state	Synonym for *privileged mode*.
system memory	Physical memory installed in a computer system external to the processor core, such RAM, ROM, and flash.
tag	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.
UISA	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.
user mode	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.

VEA	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.
virtual address	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.
word	Four bytes, or 32 bits.

Additional Reading

In addition to the source documents listed on page 311, 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.
-

XILINX®

Chapter 1

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.

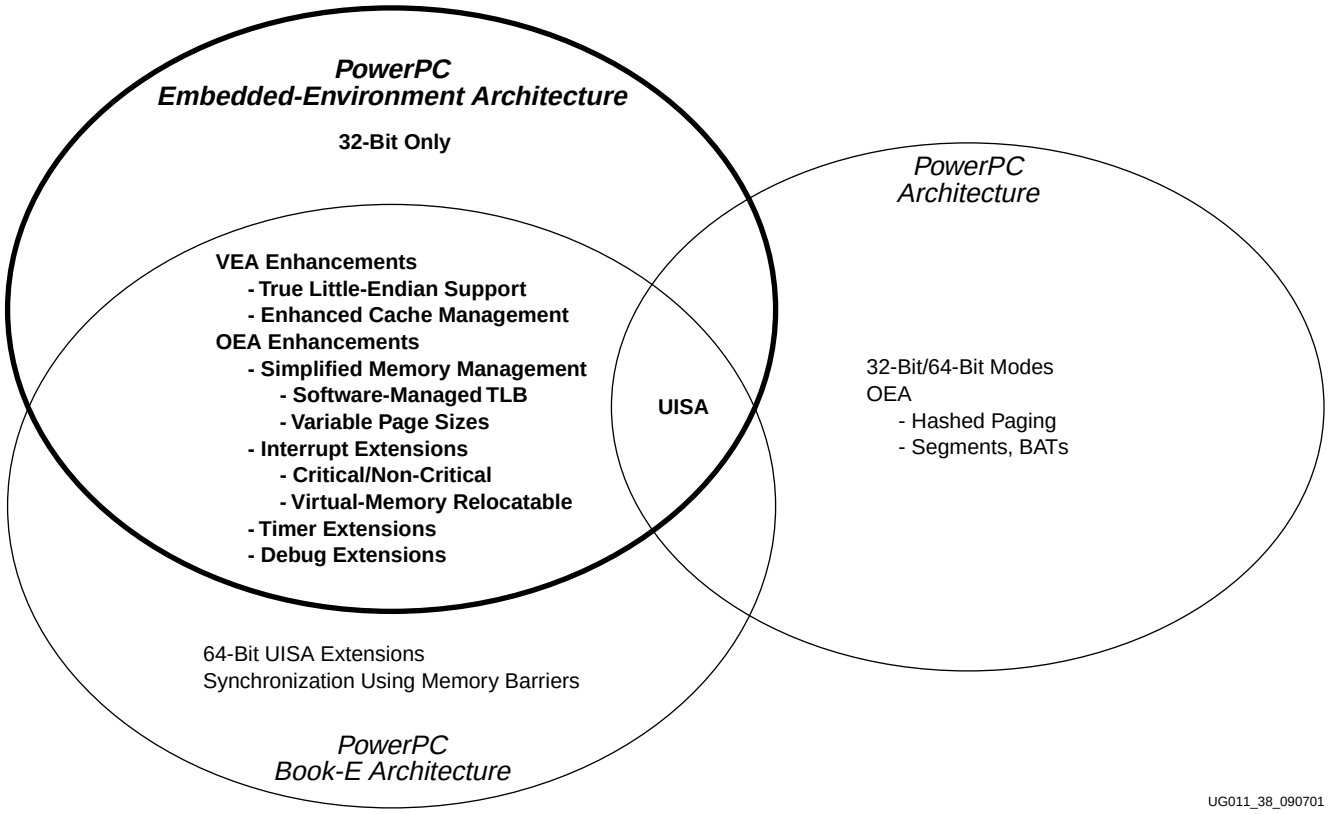


Figure 1-1: **Relationship of PowerPC Architectures**

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 decrementer (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 UISA, 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 UISA. 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 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 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: privilieged and user. The privilege modes supported by the PPC405 are described in **Processor Operating Modes**, page 343.

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 378.

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 372.

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 347, for more information on the supported data types and byte ordering.

Register Set Summary

Figure 1-2, page 333 shows the registers contained in the PPC405. Descriptions of the registers are in the following sections.

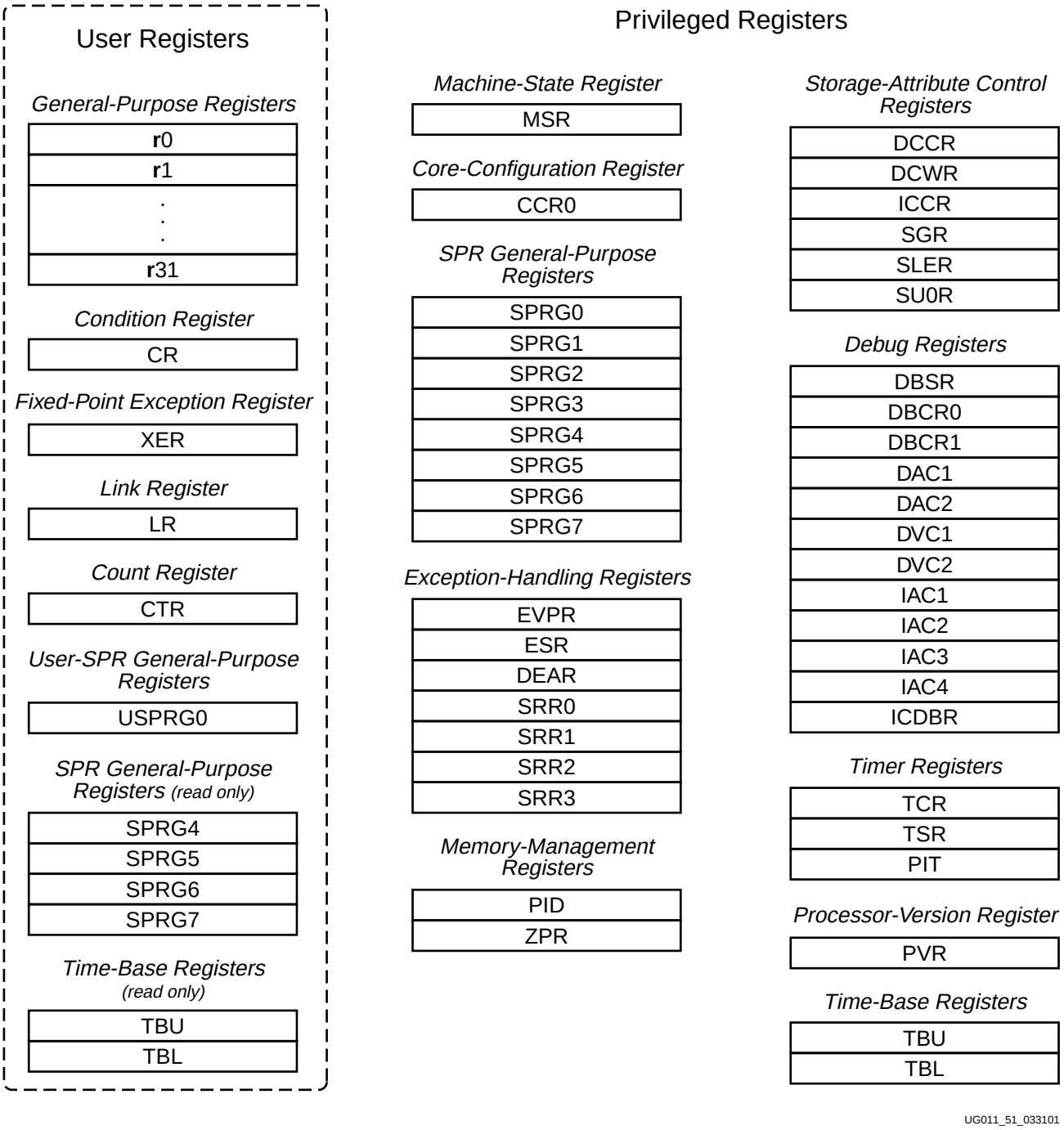


Figure 1-2: **PPC405 Registers**

General-Purpose Registers

The processor contains thirty-two 32-bit *general-purpose registers* (GPRs), identified as **r**0 through **r**31. 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 360, 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 359, and **Privileged Registers**, page 429 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 431, 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 361, 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.

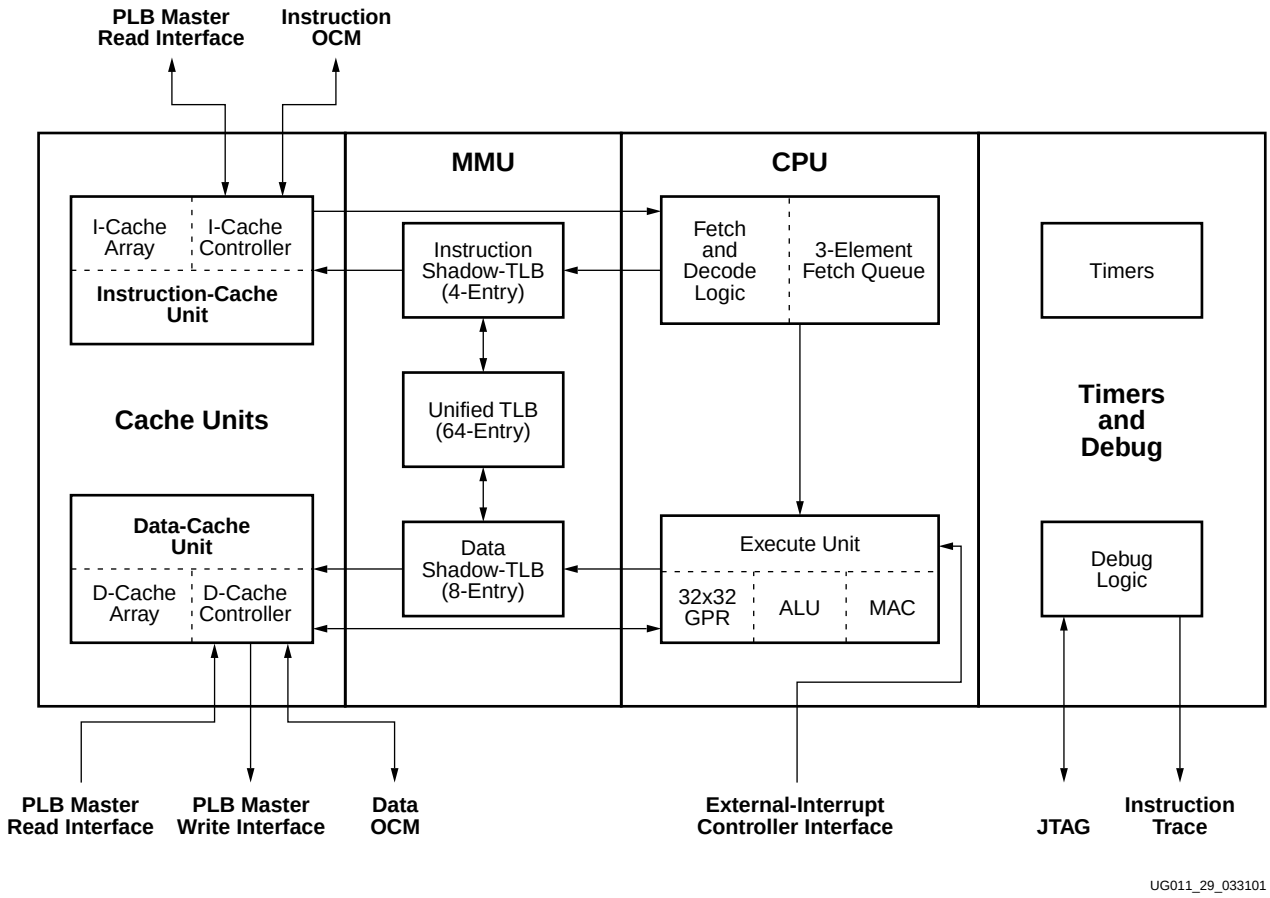


Figure 1-3: **PPC405 Organization**

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 370.

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 405 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 459 . 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.

Chapter 2

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 448, 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 424. 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 424.

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 431 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 511).

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 430. 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 434. 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 360. 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 434.

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 = (**r**A | 0)—this is referred to as *register-indirect* addressing.
- EA = (**r**A | 0) + offset—this is referred to as *register-indirect with immediate-index* addressing.
- EA = (**r**A | 0) + (**r**B)—this is referred to as *register-indirect with index* addressing.

  Note: In the above, the notation **(r**A | 0) specifies the following:

  If the **r**A instruction field is 0, the base address is 0.

  If the **r**A instruction field is not 0, the contents of register **r**A 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 + displacement—this is referred to as *branch-to-relative* addressing.
- NIA = 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**.

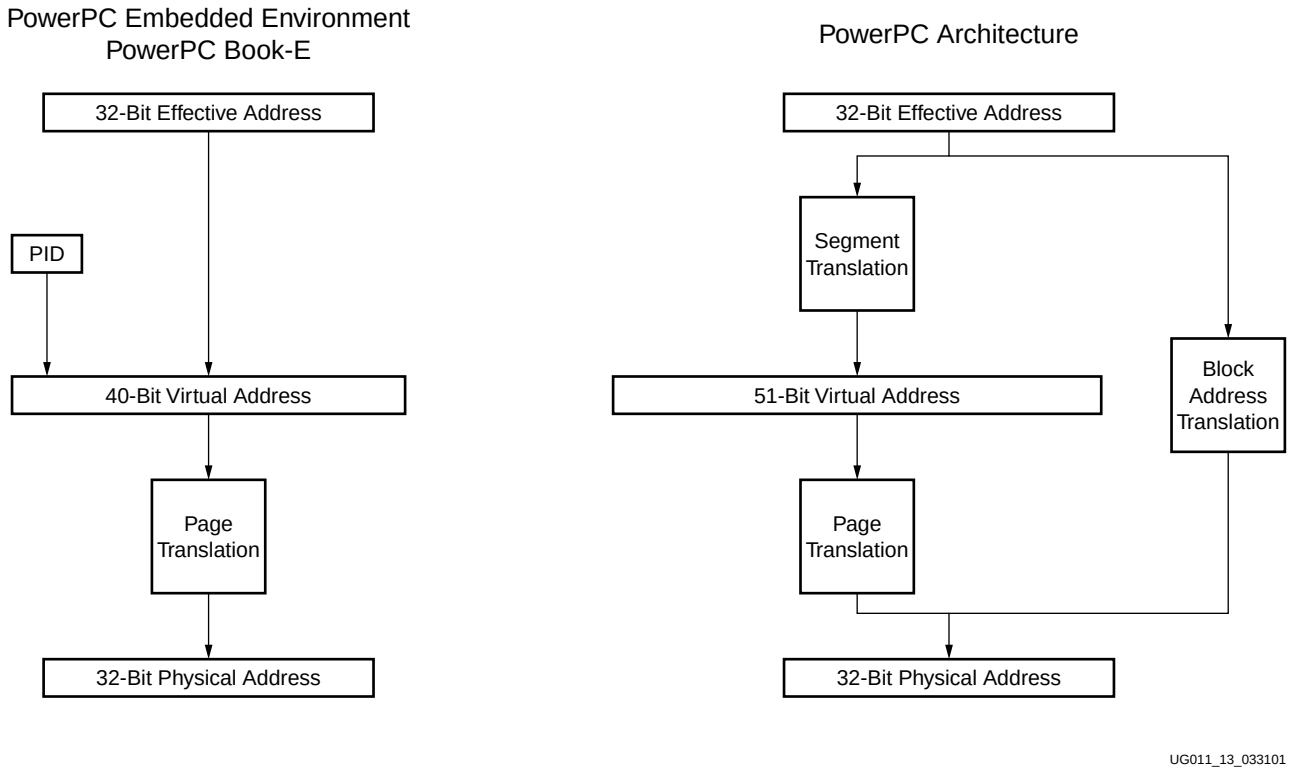


Figure 2-1: **PowerPC 32-Bit Memory Management**

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 472 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 346. 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.

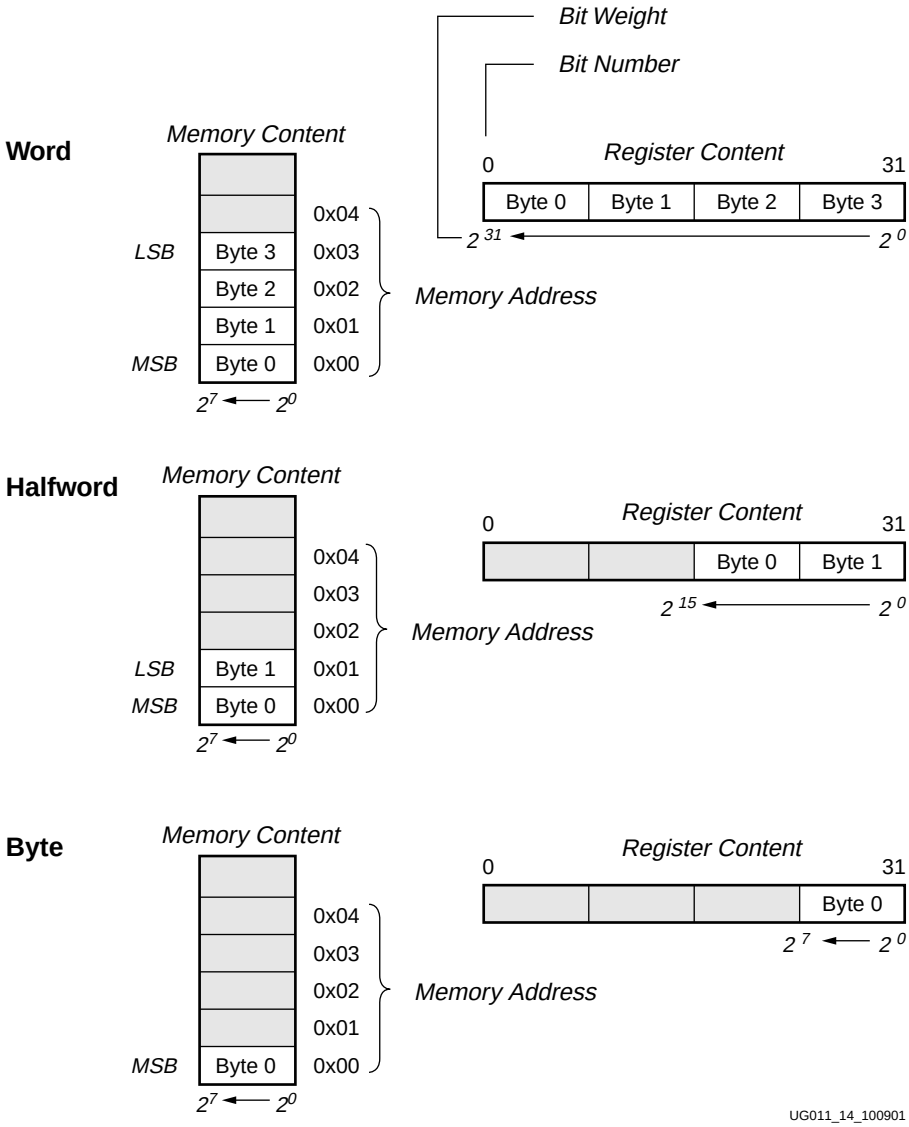


Figure 2-2: **Operand Data Types**

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	**12**	**13**	**14**				
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
21	**22**	**23**	**24**	**25**	**26**	**27**	**28**
0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
31	**32**	**33**	**34**	**'A'**	**'B'**	**'C'**	**'D'**
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
'E'	**'F'**	**'G'**		**51**	**52**		
0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
61	**62**	**63**	**64**				
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27

Little-Endian Mapping

The little-endian mapping of structure *s* follows.

14	**13**	**12**	**11**				
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
28	**27**	**26**	**25**	**24**	**23**	**22**	**21**
0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
34	**33**	**32**	**31**	**'A'**	**'B'**	**'C'**	**'D'**
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
'E'	**'F'**	**'G'**		**52**	**51**		
0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
64	**63**	**62**	**61**				
0x20	0x21	0x22	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 455 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 456 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 349, 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 385, 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	*xxxx*
Halfword	2 Bytes	*xxx*0
Word	4 Bytes	*xx*00
Doubleword	8 Bytes	*x*000

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 510 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 792. The **Instruction Set Information**, page 797 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 781. 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)		
mfdcr	**tlbre**	**dccci**	**macchw[o][.]**	**nmacchw[o][.]**
mtdcr	**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.

XILINX®

Chapter 3

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 524. Floating-point registers are not supported by the PPC405.

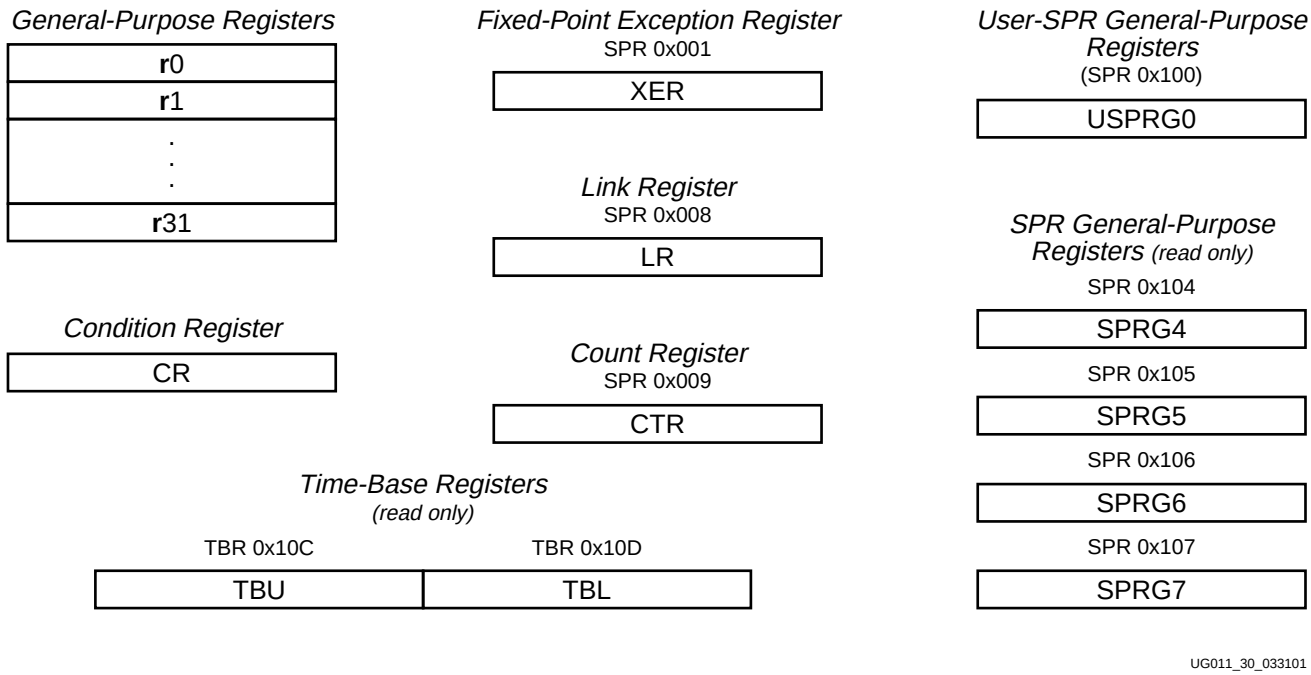


Figure 3-1: **PPC405 User Registers**

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 424 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 770 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 830 for more information.

General-Purpose Registers (GPRs)

The PPC405 contains thirty-two 32-bit general-purpose registers (GPRs), numbered **r**0 through **r**31, 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.

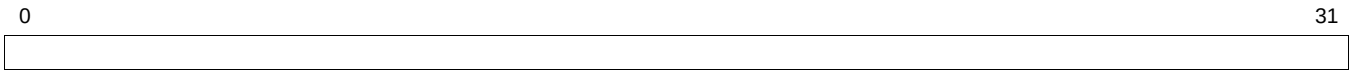


Figure 3-2: **General Purpose Registers (R0-R31)**

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.

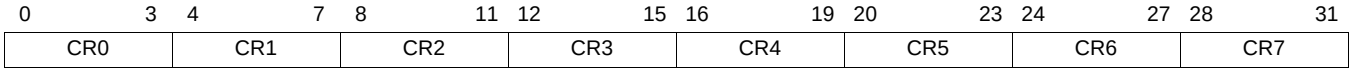


Figure 3-3: **Condition Register (CR)**

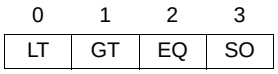


Figure 3-4: **CR*n* Field**

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—**r**A is not less than. 1—**r**A is less than.	This bit is set when **r**A < SIMM or **r**B (signed comparison), or **r**A < UIMM or **r**B (unsigned comparison), otherwise it is cleared.
1	GT	Greater than 0—**r**A is not greater than. 1—**r**A is greater than.	This bit is set when **r**A > SIMM or **r**B (signed comparison), or **r**A > UIMM or **r**B (unsigned comparison), otherwise it is cleared.
2	EQ	Equal to 0—**r**A is not equal. 1—**r**A is equal.	This bit is set when **r**A = SIMM or **r**B (signed comparison), or **r**A = UIMM or **r**B (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.

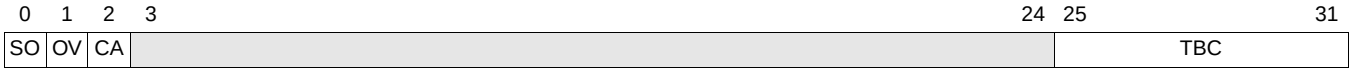


Figure 3-5: **Fixed Point Exception Register (XER)**

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 *add-carrying*, *subtract-from-carrying*, *add-extended*, and *subtract-from-extended* 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 *load-string word indexed* (**lswx**) or *store-string word indexed* (**stswx**) instruction.

The XER is an SPR with an address of 1 (0x001) and can be read and written using the **mfspr** and **mtspr** 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* (**bclr***x*) 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.

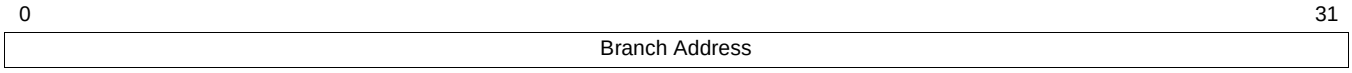


Figure 3-6: **Link Register (LR)**

The LR is an SPR with an address of 8 (0x008) and can be read and written using the **mfspr** and **mtspr** 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 371 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 367 for information on encoding the BO opcode field.
- The CTR can hold the branch-target address used by *branch-conditional to count-register* (**bcctr***x*) instructions.

The format of CTR is shown in Figure 3-7.

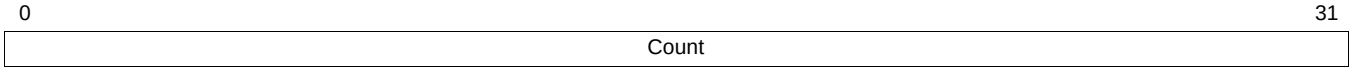


Figure 3-7: **Count Register (CTR)**

The CTR is an SPR with an address of 9 (0x009) and can be read and written using the **mfspr** and **mtspr** 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.

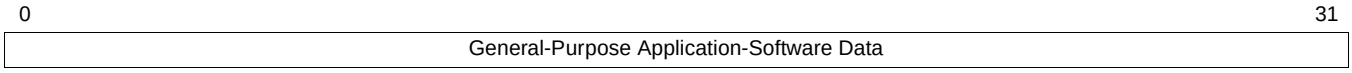


Figure 3-8: **User SPR General-Purpose Register (USPRG0)**

The USPRG0 is an SPR with an address of 256 (0x100) and can be read and written using the **mfspr** and **mtspr** 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.

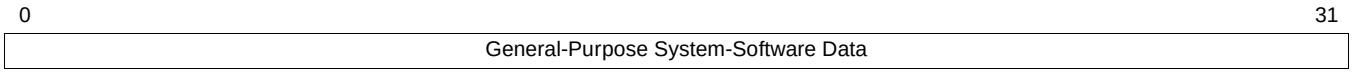


Figure 3-9: **SPR General-Purpose Registers (SPRG4–SPRG7)**

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 **mfspr** instruction. In privileged mode, system software accesses these registers using different SPR numbers (see page 432).

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.

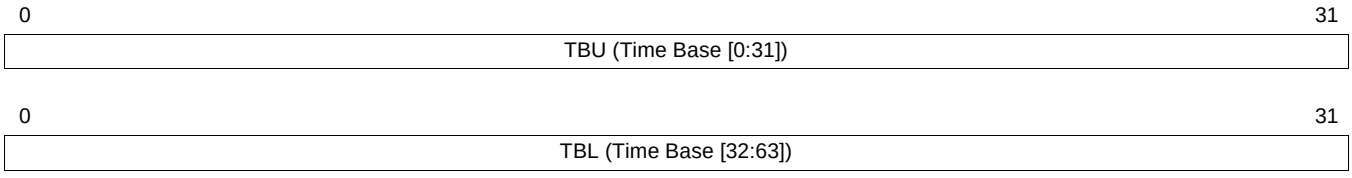


Figure 3-10: **Time-Base Register**

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 524, 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, CR_{10} 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 370.

Table 3-5: **Valid BO Opcode-Field Encoding**

BO[0:4]	Description
0000*y*	Decrement the CTR. Branch if the decremented CTR ≠ 0 and CR[BI]=0.
0001*y*	Decrement the CTR. Branch if the decremented CTR = 0 and CR[BI]=0.
001*zy*	Branch if CR[BI]=0.
0100*y*	Decrement the CTR. Branch if the decremented CTR ≠ 0 and CR[BI]=1.
0101*y*	Decrement the CTR. Branch if the decremented CTR=0 and CR[BI]=1.
011*zy*	Branch if CR[BI]=1.
1*z*00*y*	Decrement the CTR. Branch if the decremented CTR ≠ 0.
1*z*01*y*	Decrement the CTR. Branch if the decremented CTR = 0.
1*z*1*zz*	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 821 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 368. 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 368. 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 368. 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 **bc***x* instruction with a negative value in the displacement operand, the branch is predicted taken.
- For all other branch-conditional instructions (**bc***x* with a non-negative value in the displacement operand, **bclr***x*, or **bcctr***x*), 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] ∧ 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] ∧ BO[2]=0), *s* controls the prediction. In the **bclr***x* and **bcctr***x* instructions, bit 16 (*s*) is reserved and always 0. In this case those instructions are predicted not-taken.

Only the **bc***x* instructions can specify a displacement value. The **bc***x* 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.

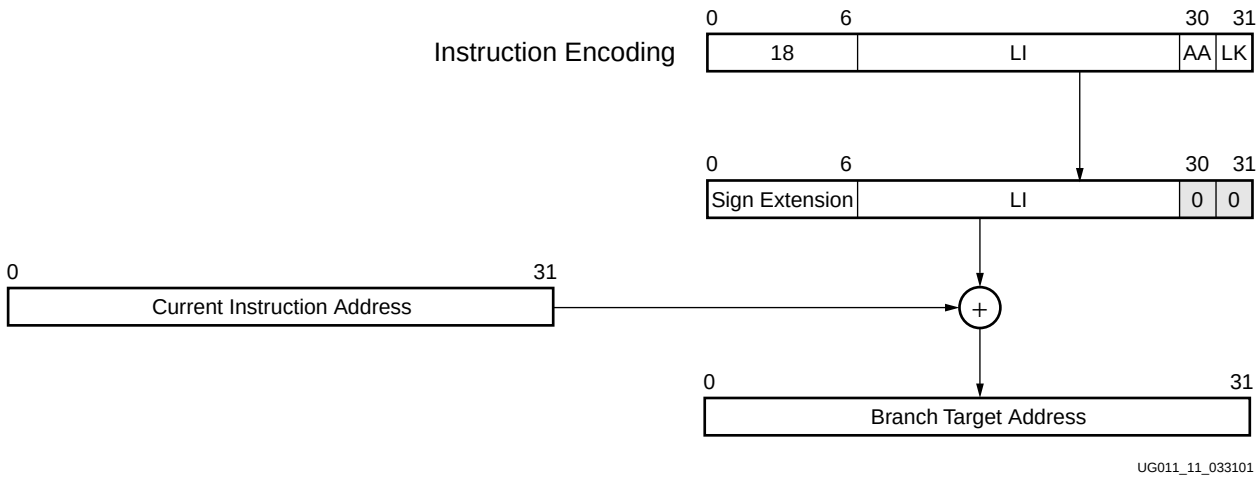


Figure 3-11: **Branch-to-Relative Addressing**

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.

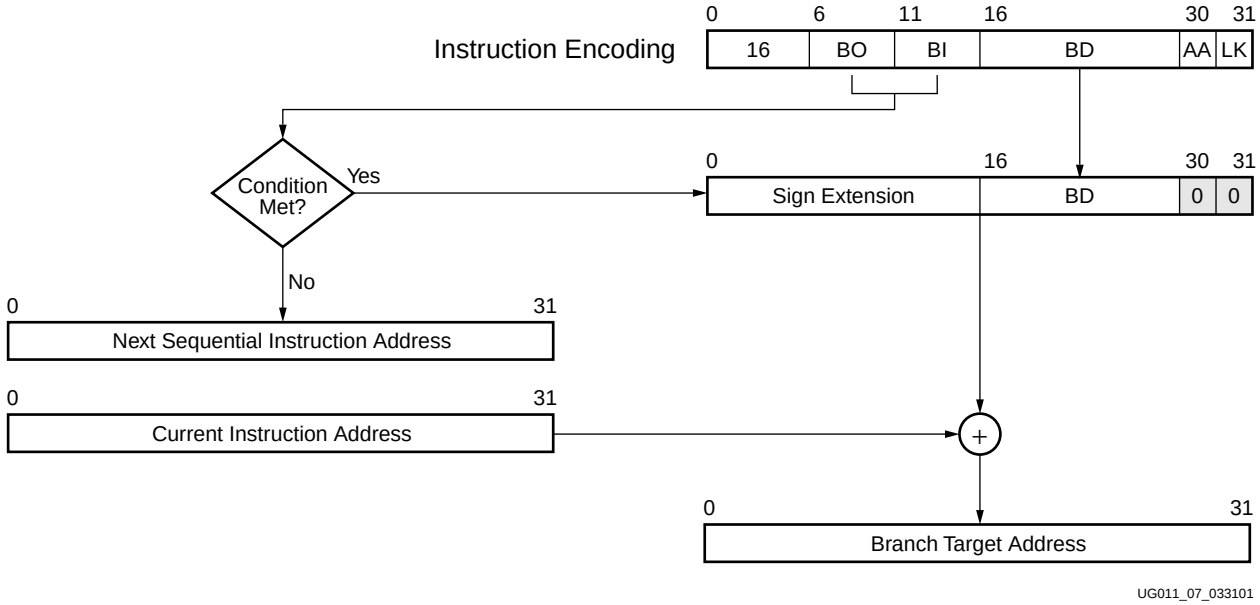


Figure 3-12: **Branch-Conditional to Relative Addressing**

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.

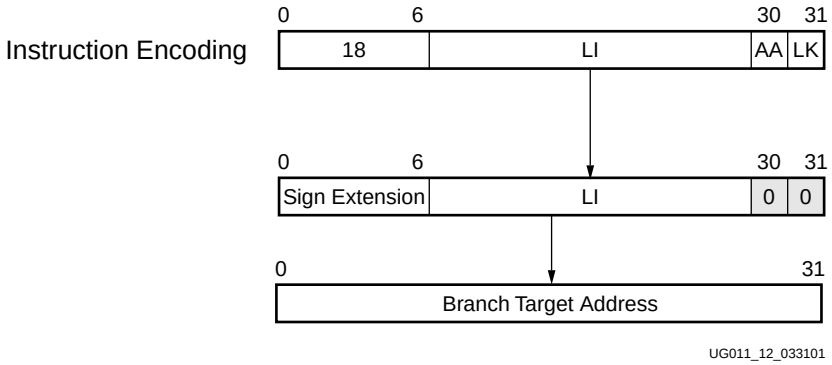


Figure 3-13: **Branch-to-Absolute Addressing**

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.

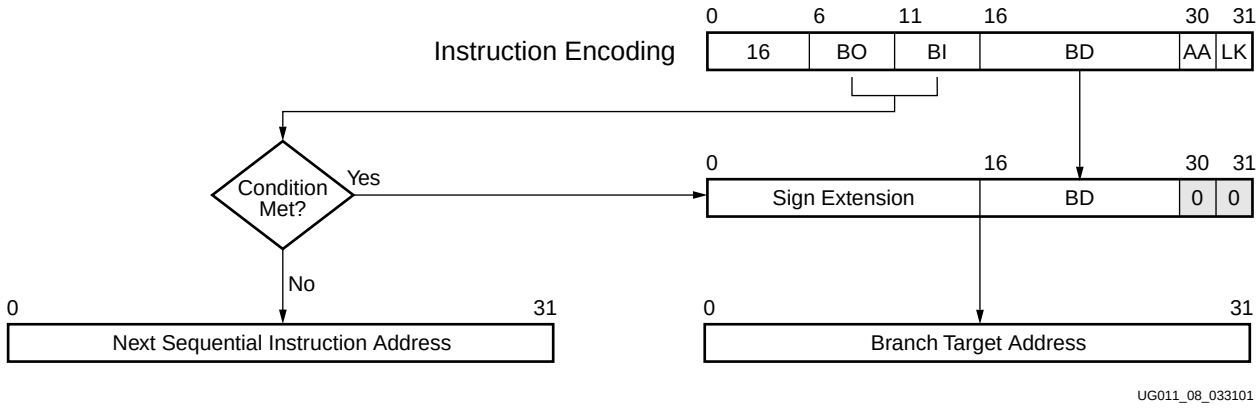


Figure 3-14: **Branch-Conditional to Absolute Addressing**

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.

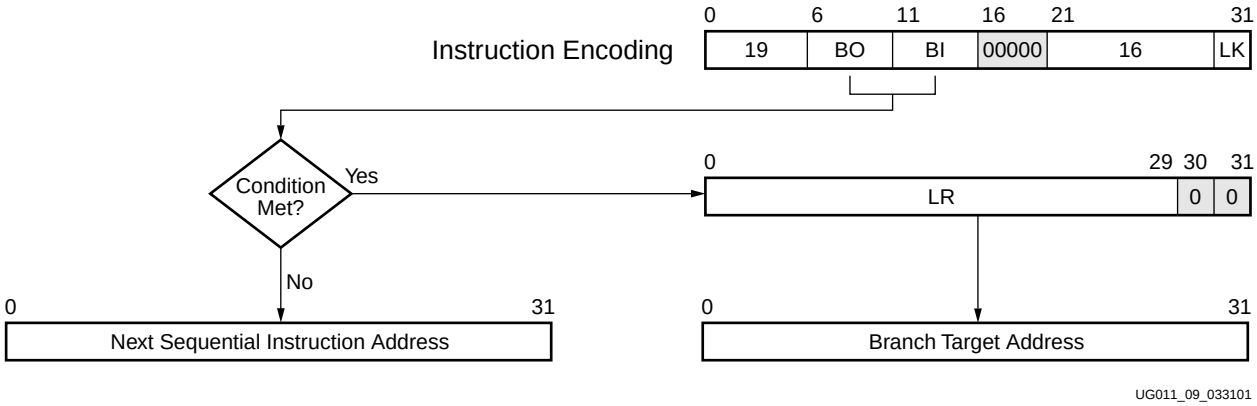


Figure 3-15: **Branch-Conditional to Link-Register Addressing**

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.

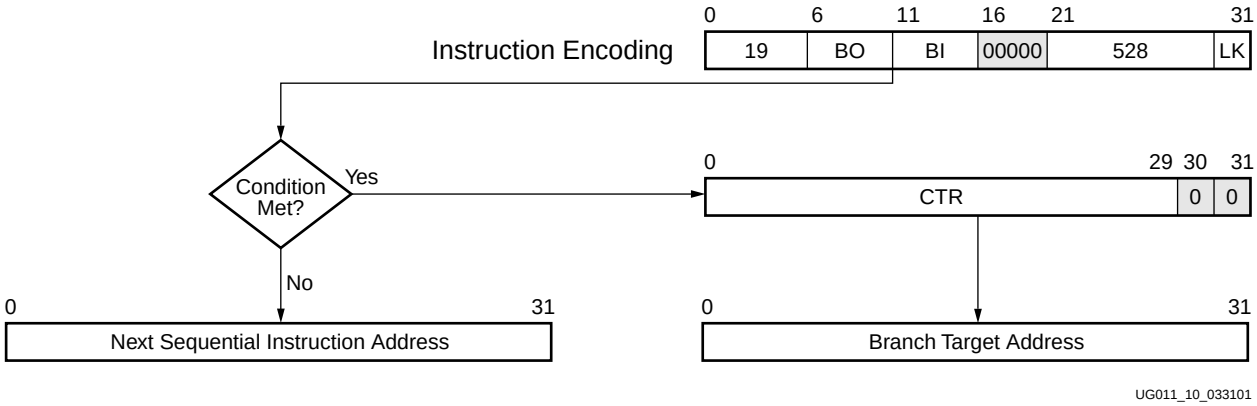


Figure 3-16: **Branch-Conditional to Count-Register Addressing**

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 828 for more information.

In Table 3-10, the instruction-operand fields **crb**A, **crb**B, **and crb**D all specify a single *bit* within the CR. The instruction-operand fields **crf**D and **crf**S 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 **crb**A is ANDed with CR-bit **crb**B and the result is stored in CR-bit **crb**D.	**crb**D,**crb**A,**crb**B
crandc	Condition Register AND with Complement	CR-bit **crb**A is ANDed with the *complement* of CR-bit **crb**B and the result is stored in CR-bit **crb**D.	
creqv	Condition Register Equivalent	CR-bit **crb**A is XORed with CR-bit **crb**B and the *complemented* result is stored in CR-bit **crb**D.	
crnand	Condition Register NAND	CR-bit **crb**A is ANDed with CR-bit **crb**B and the *complemented* result is stored in CR-bit **crb**D.	
crnor	Condition Register NOR	CR-bit **crb**A is ORed with CR-bit **crb**B and the *complemented* result is stored in CR-bit **crb**D.	
cror	Condition Register OR	CR-bit **crb**A is ORed with CR-bit **crb**B and the result is stored in CR-bit **crb**D.	
crorc	Condition Register OR with Complement	CR-bit **crb**A is ORed with the *complement* of CR-bit **crb**B and the result is stored in CR-bit **crb**D.	
crxor	Condition Register XOR	CR-bit **crb**A is XORed with CR-bit **crb**B and the result is stored in CR-bit **crb**D.	
mcrf	Move Condition Register Field	CR-field **crf**S is copied into CR-field **crf**D. No other CR fields are modified.	**crf**D,**crf**S

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 514 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 511 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 546 for more information.

Simplified mnemonics are defined for the system-trap instructions. See **Trap Instructions**, page 832 for more information.

Table 3-12: **System-Trap Instructions**

Mnemonic	Name	Operation	Operand Syntax
tw	Trap Word	The contents of **r**A are compared with **r**B. A program exception occurs if the comparison meets any test condition enabled by the TO operand.	TO,**r**A,**r**B
twi	Trap Word Immediate	The contents of **r**A 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,**r**A,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 424 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 467 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 **r**A 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 353 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, **r**A. The index is sign-extended to 32 bits and added to the contents of **r**A to generate the effective address. If the **r**A instruction field is 0 (specifying **r**0), a value of zero—rather than the contents of **r**0—is added to the sign-extended immediate index. The option to specify **r**A or 0 is shown in the instruction description as (**r**A | 0).

Figure 3-17 shows how an effective address is generated when using register-indirect with immediate-index addressing.

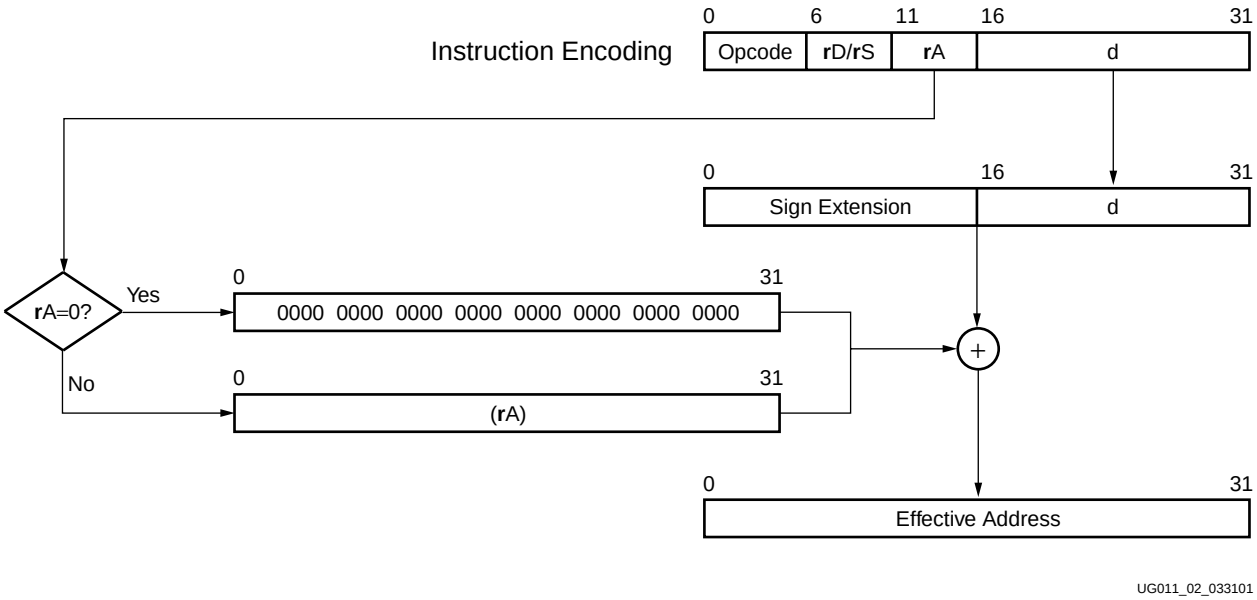


Figure 3-17: **Register-Indirect with Immediate-Index Addressing**

Register-Indirect with Index

Load and store instructions using this addressing mode contain two general-purpose register operands, **r**A and **r**B. The contents of these two registers are added to generate the effective address. If the **r**A instruction field is 0 (specifying **r**0), a value of zero—rather than the contents of **r**0—is added to **r**B. The option to specify **r**A or 0 is shown in the instruction description as (**r**A | 0).

Figure 3-18 shows how an effective address is generated when using register-indirect with index addressing.

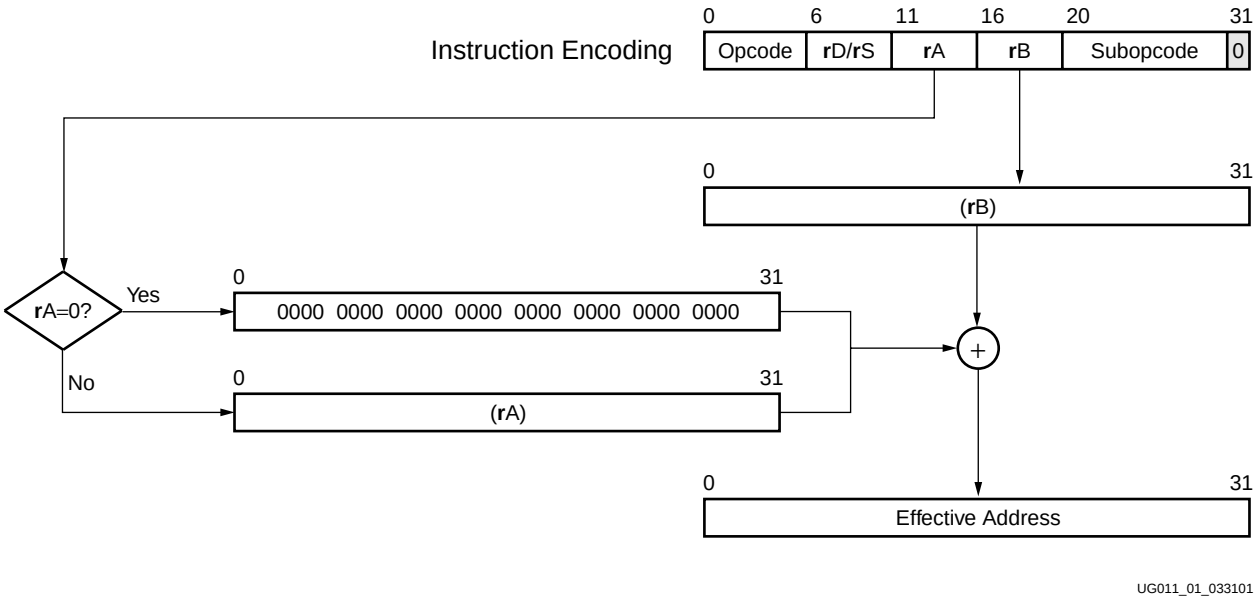


Figure 3-18: **Register-Indirect with Index Addressing**

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 **r**A operand as the effective address. Rather than using the contents of **r**0, a zero in the **r**A operand causes an effective address of zero to be generated. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A | 0).

Figure 3-19 shows how an effective address is generated when using register-indirect addressing.

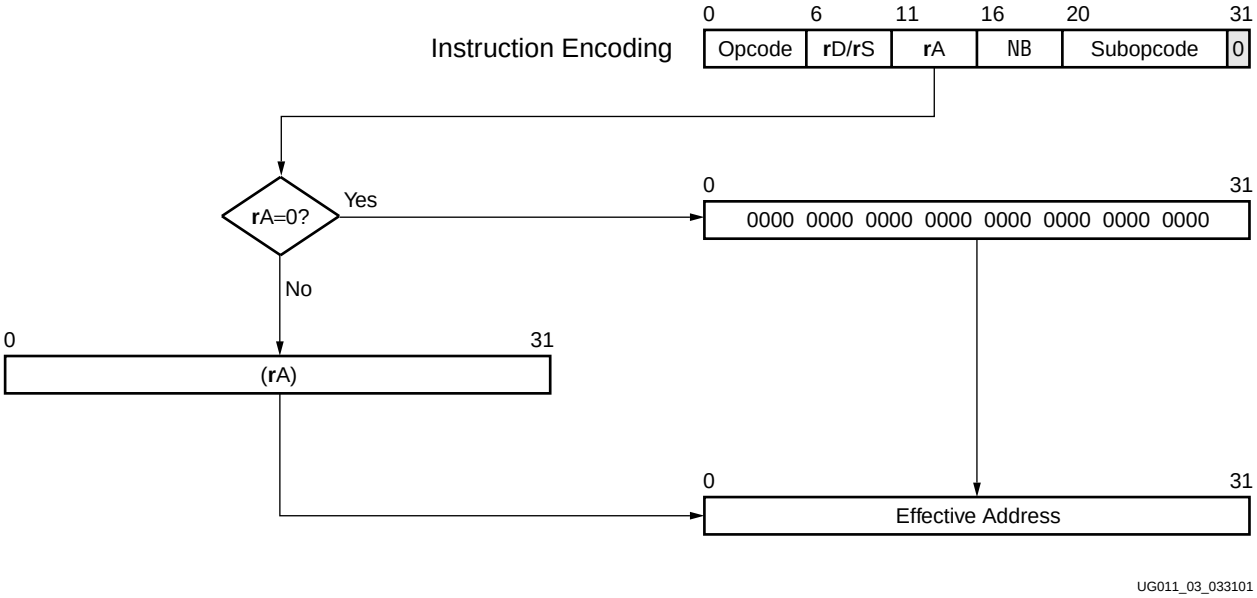


Figure 3-19: **Register-Indirect Addressing**

Load Instructions

Integer-load instructions read an operand from memory and store it in a GPR destination register, **r**D. Each type of load is characterized by what they do with unused high-order bits in **r**D when the operand size is less than a word (32 bits). *Load-and-zero* instructions clear the unused high-order bits in **r**D to zero. *Load-algebraic* instructions fill the unused high-order bits in **r**D with a copy of the most-significant bit in the operand.

Load-with-update instructions are provided, but the following two rules apply:

- **r**A must not be equal to 0. If **r**A = 0, the instruction form is invalid.
- **r**A must not be equal to **r**D. If **r**A = **r**D, 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 **r**D and clear the upper-24 bits of **r**D 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 = (**r**A | 0) + d	**r**D,d(**r**A)
lbzu	Load Byte and Zero with Update	Register-indirect with immediate index EA = (**r**A) + d **r**A ← EA **r**A ≠ 0, **r**A ≠ **r**D	
lbzx	Load Byte and Zero Indexed	Register-indirect with index EA = (**r**A | 0) + (**r**B)	**r**D,**r**A,**r**B
lbzux	Load Byte and Zero with Update Indexed	Register-indirect with index EA = (**r**A) + (**r**B) **r**A ← EA **r**A ≠ 0, **r**A ≠ **r**D	

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 **r**D and clear the upper-16 bits of **r**D to 0.

Table 3-15: **Load Halfword and Zero Instructions**

Mnemonic	Name	Addressing Mode	Operand Syntax
lhz	Load Halfword and Zero	Register-indirect with immediate index EA = (**r**A | 0) + d	**r**D,d(**r**A)
lhzu	Load Halfword and Zero with Update	Register-indirect with immediate index EA = (**r**A) + d **r**A ← EA **r**A ≠ 0, **r**A ≠ **r**D	
lhzx	Load Halfword and Zero Indexed	Register-indirect with index EA = (**r**A | 0) + (**r**B)	**r**D,**r**A,**r**B
lhzux	Load Halfword and Zero with Update Indexed	Register-indirect with index EA = (**r**A) + (**r**B) **r**A ← EA **r**A ≠ 0, **r**A ≠ **r**D	

Load Word and Zero

Table 3-16 lists the PowerPC *load word and zero* instructions. These instructions load a word from memory into **r**D.

Table 3-16: **Load-Word and Zero Instructions**

Mnemonic	Name	Addressing Mode	Operand Syntax
lwz	Load Word and Zero	Register-indirect with immediate index EA = (**r**A | 0) + d	**r**D,d(**r**A)
lwzu	Load Word and Zero with Update	Register-indirect with immediate index EA = (**r**A) + d **r**A ← EA **r**A ≠ 0, **r**A ≠ **r**D	
lwzx	Load Word and Zero Indexed	Register-indirect with index EA = (**r**A | 0) + (**r**B)	**r**D,**r**A,**r**B
lwzux	Load Word and Zero with Update Indexed	Register-indirect with index EA = (**r**A) + (**r**B) **r**A ← EA **r**A ≠ 0, **r**A ≠ **r**D	

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 **r**D. The upper-16 bits of **r**D 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 = (**r**A | 0) + d	**r**D,d(**r**A)
lhau	Load Halfword Algebraic with Update	Register-indirect with immediate index EA = (**r**A) + d **r**A ← EA **r**A ≠ 0, **r**A ≠ **r**D	
lhax	Load Halfword Algebraic Indexed	Register-indirect with index EA = (**r**A | 0) + (**r**B)	**r**D,**r**A,**r**B
lhaux	Load Halfword Algebraic with Update Indexed	Register-indirect with index EA = (**r**A) + (**r**B) **r**A ← EA **r**A ≠ 0, **r**A ≠ **r**D	

Store Instructions

Integer-store instructions read an operand from a GPR source register, **r**S, and write it into memory. Store-with-update instructions are provided, but the following two rules apply:

- **r**A must not be equal to 0. If **r**A = 0, the instruction form is invalid.
- If **r**S = **r**A, **r**S is written to memory first, and then the effective address is loaded into **r**S.

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 **r**S 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 = (**r**A | 0) + d	**r**S,d(**r**A)
stbu	Store Byte with Update	Register-indirect with immediate index EA = (**r**A) + d **r**A ← EA **r**A ≠ 0	
stbx	Store Byte Indexed	Register-indirect with index EA = (**r**A | 0) + (**r**B)	**r**S,**r**A,**r**B
stbux	Store Byte with Update Indexed	Register-indirect with index EA = (**r**A) + (**r**B) **r**A ← EA **r**A ≠ 0	

Store Halfword

Table 3-19 lists the PowerPC *store halfword* instructions. These instructions store the lower-16 bits of **r**S 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 = (**r**A | 0) + d	**r**S,d(**r**A)
sthu	Store Halfword with Update	Register-indirect with immediate index EA = (**r**A) + d **r**A ← EA **r**A ≠ 0	
sthx	Store Halfword Indexed	Register-indirect with index EA = (**r**A | 0) + (**r**B)	**r**S,**r**A,**r**B
sthux	Store Halfword with Update Indexed	Register-indirect with index EA = (**r**A) + (**r**B) **r**A ← EA **r**A ≠ 0	

Store Word

Table 3-20 lists the PowerPC *store word* instructions. These instructions store the entire contents of **r**S 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 = (**r**A | 0) + d	**r**S,d(**r**A)
stwu	Store Word with Update	Register-indirect with immediate index EA = (**r**A) + d **r**A ← EA **r**A ≠ 0	
stwx	Store Word Indexed	Register-indirect with index EA = (**r**A | 0) + (**r**B)	**r**S,**r**A,**r**B
stwux	Store Word with Update Indexed	Register-indirect with index EA = (**r**A) + (**r**B) **r**A ← EA **r**A ≠ 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 **r**D 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 349.

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 = (**r**A | 0) + (**r**B)	**r**D,**r**A,**r**B
lwbrx	Load Word Byte-Reverse Indexed		
sthbrx	Store Halfword Byte-Reverse Indexed	Register-indirect with index EA = (**r**A | 0) + (**r**B)	**r**S,**r**A,**r**B
stwbrx	Store Word Byte-Reverse Indexed		

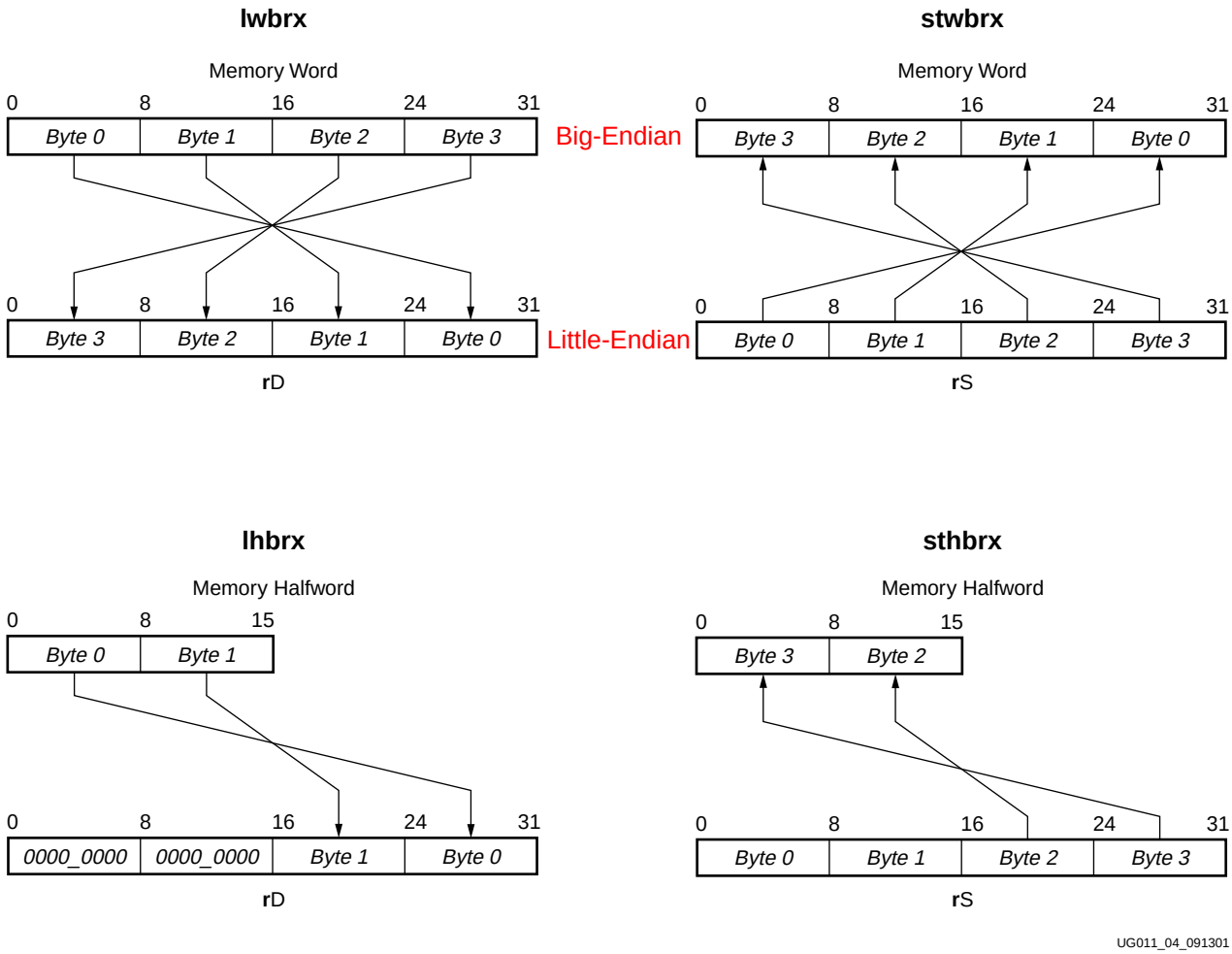


Figure 3-20: **Load and Store with Byte-Reverse Instructions**

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, **r**D through **r**31 are loaded with *n*

consecutive words from memory, where *n*=32-**r**D. For the **lmw** instruction, if **r**A is in the range of registers to be loaded, or if **r**D=0, the instruction form is invalid. When the *store multiple word* instruction (**stmw**) is executed, the *n* consecutive words in **r**S through **r**31 are stored into memory, where *n*=32-**r**S.

Table 3-22: **Load and Store Multiple Instructions**

Mnemonic	Name	Addressing Mode	Operand Syntax
lmw	Load Multiple Word	Register-indirect with immediate index EA = (**r**A | 0) + d	**r**D,d(**r**A)
stmw	Store Multiple Word	Register-indirect with immediate index EA = (**r**A | 0) + d	**r**S,d(**r**A)

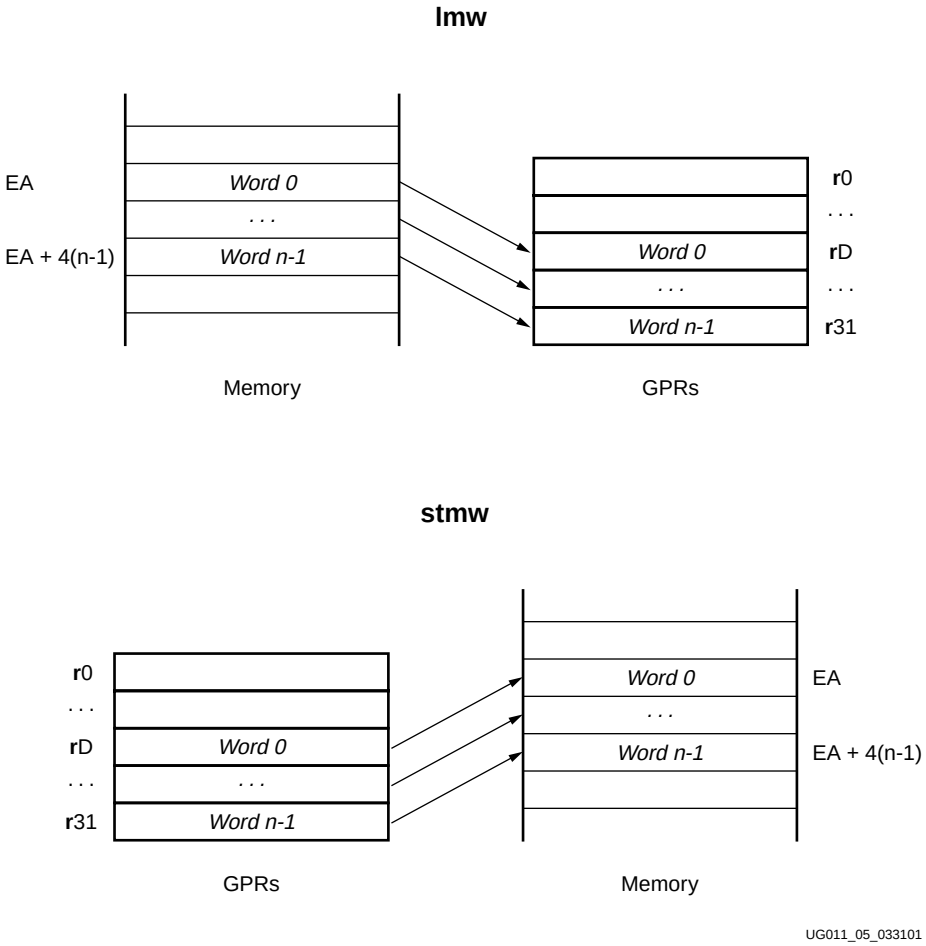


Figure 3-21: **Load and Store Multiple Instructions**

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 = (**r**A | 0)	**r**D,**r**A,NB
lswx	Load String Word Indexed	Register-indirect with index EA = (**r**A | 0) + (**r**B)	**r**D,**r**A,**r**B
stswi	Store String Word Immediate	Register-indirect EA = (**r**A | 0)	**r**S,**r**A,NB
stswx	Store String Word Indexed	Register-indirect with index EA = (**r**A | 0) + (**r**B)	**r**S,**r**A,**r**B

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 353 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 **r**D 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.

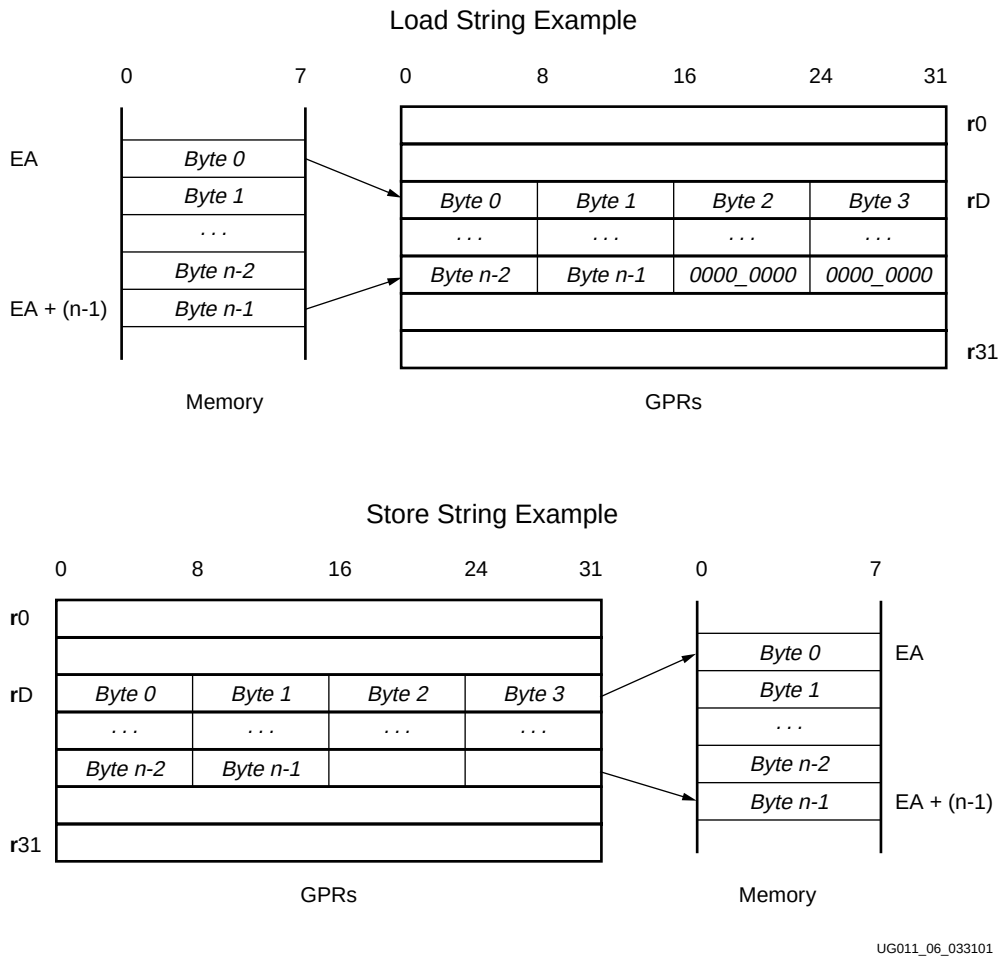


Figure 3-22: **Load and Store String Instructions**

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 361, 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 363 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 **r**3:**r**4, where **r**3 contains the most-significant 32 bits of *i*, and **r**4 contains the least-significant 32 bits. The 64-bit integer *j* is similarly represented by the register pair **r**5:**r**6. The 64-bit result *i+j=r* (represented by the pair **r**7:**r**8) 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
Add Instructions		**r**D is loaded with the sum (**r**A) + (**r**B).	
add	Add	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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
Add-Carrying Instructions		**r**D is loaded with the sum (**r**A) + (**r**B).	
addc	Add Carrying	XER[CA] is updated to reflect the result.	**r**D,**r**A,**r**B
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.	
Add-Immediate Instructions		**r**D is loaded with the sum (**r**A | 0) + SIMM.	
addi	Add Immediate	XER and CR0 are *not* updated.	**r**D,**r**A,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.	
Add Immediate-Shifted Instructions		**r**D is loaded with the sum (**r**A | 0) + (SIMM || 0x0000).	
addis	Add Immediate Shifted	XER and CR0 are *not* updated.	**r**D,**r**A,SIMM
Add-Extended Instructions		**r**D is loaded with the sum (**r**A) + (**r**B) + XER[CA].	
adde	Add Extended	XER[CA] is updated to reflect the result.	**r**D,**r**A,**r**B
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.	
Add to Minus-One-Extended Instructions		**r**D is loaded with the sum (**r**A) + XER[CA] + 0xFFFF_FFFF.	
addme	Add to Minus One Extended	XER[CA] is updated to reflect the result.	**r**D,**r**A
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
Add to Zero-Extended Instructions		**r**D is loaded with the sum (**r**A) + XER[CA].	
addze	Add to Zero Extended	XER[CA] is updated to reflect the result.	**r**D,**r**A
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 (**r**A) from the third operand (**r**B). 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 831 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 **r**3:**r**4, where **r**3 contains the most-significant 32 bits of *i*, and **r**4 contains the least-significant 32 bits. The 64-bit integer *j* is similarly represented by the register pair **r**5:**r**6. The 64-bit result *i–j=r* (represented by the pair **r**7:**r**8) 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
Subtract-From Instructions		**r**D is loaded with the sum ¬(**r**A) + (**r**B) + 1.	
subf	Subtract from	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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		**r**D is loaded with the sum ¬(**r**A) + (**r**B) + 1.	
subfc	Subtract from Carrying	XER[CA] is updated to reflect the result.	**r**D,**r**A,**r**B
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		**r**D is loaded with the sum ¬(**r**A) + SIMM + 1.	
subfic	Subtract from Immediate Carrying	XER[CA] is updated to reflect the result.	**r**D,**r**A,SIMM
Subtract-From Extended Instructions		**r**D is loaded with the sum ¬(**r**A) + (**r**B) + XER[CA].	
subfe	Subtract from Extended	XER[CA] is updated to reflect the result.	**r**D,**r**A,**r**B
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		**r**D is loaded with the sum ¬(**r**A) + XER[CA] + 0xFFFF_FFFF.	
subfme	Subtract from Minus One Extended	XER[CA] is updated to reflect the result.	**r**D,**r**A
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		**r**D is loaded with the sum ¬(**r**A) + XER[CA].	
subfze	Subtract from Zero Extended	XER[CA] is updated to reflect the result.	**r**D,**r**A
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 **r**A and writes the two's-compliment equivalent in **r**D. 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
Negation Instructions		**r**D is loaded with the sum ¬(**r**A) + 1.	
neg	Negate	XER and CR0 are *not* updated.	**r**D,**r**A
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
Multiply Low-Word Instructions		**r**D is loaded with the low-32 bits of the product (**r**A) × (**r**B).	
mullw	Multiply Low Word	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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.	
Multiply Low-Word Immediate Instructions		**r**D is loaded with the low-32 bits of the product (**r**A) × SIMM.	
mulli	Multiply Low Immediate	XER and CR0 are *not* updated.	**r**D,**r**A,SIMM
Multiply High-Word Instructions		**r**D is loaded with the high-32 bits of the product (**r**A) × (**r**B).	
mulhw	Multiply High Word	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
mulhw.	Multiply High Word and Record	CR0 is updated to reflect the result.	
Multiply High-Word Unsigned Instructions		**r**D is loaded with the high-32 bits of the product (**r**A) × (**r**B). The contents of **r**A and **r**B are interpreted as unsigned integers.	
mulhwu	Multiply High Word	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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		**r**D is loaded with the low-32 bits of the 64-bit quotient (**r**A) ÷ (**r**B).	
divw	Divide Word	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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		**r**D is loaded with the low-32 bits of the 64-bit quotient (**r**A) ÷ (**r**B). The contents of **r**A and **r**B are interpreted as unsigned integers.	
divwu	Divide Word Unsigned	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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 **r**A operand specifies a *destination* register rather than a source register. **r**S 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		**r**A is loaded with the logical result (**r**S) AND (**r**B).	
and	AND	CR0 is *not* updated.	**r**A,**r**S,**r**B
and.	AND and Record	CR0 is updated to reflect the result.	
AND-Immediate Instructions		**r**A is loaded with the logical result (**r**S) AND UIMM.	
andi.	AND Immediate and Record	CR0 is updated to reflect the result.	**r**A,**r**S,UIMM
AND Immediate-Shifted Instructions		**r**A is loaded with the logical result (**r**S) AND (UIMM || 0x0000)	
andis.	AND Immediate Shifted and Record	CR0 is updated to reflect the result.	**r**A,**r**S,UIMM
AND with Complement Instructions		**r**A is loaded with the logical result (**r**S) AND ¬(**r**B).	
andc	AND with Complement	CR0 is *not* updated.	**r**A,**r**S,**r**B
andc.	AND with Complement and Record	CR0 is updated to reflect the result.	
NAND Instructions		**r**A is loaded with the logical result ¬((**r**S) AND (**r**B)).	
nand	NAND	CR0 is *not* updated.	**r**A,**r**S,**r**B
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 834 for more information.

Table 3-30: **OR and NOR Instructions**

Mnemonic	Name	Operation	Operand Syntax
NOR Instructions		**r**A is loaded with the logical result ¬((**r**S) OR (**r**B)).	
nor	NOR	CR0 is *not* updated.	**r**A,**r**S,**r**B
nor.	NOR and Record	CR0 is updated to reflect the result.	
OR Instructions		**r**A is loaded with the logical result (**r**S) OR (**r**B).	
or	OR	CR0 is *not* updated.	**r**A,**r**S,**r**B
or.	OR and Record	CR0 is updated to reflect the result.	
OR-Immediate Instructions		**r**A is loaded with the logical result (**r**S) OR UIMM.	
ori	OR Immediate	CR0 is *not* updated.	**r**A,**r**S,UIMM

Table 3-30: **OR and NOR Instructions *(Continued)***

Mnemonic	Name	Operation	Operand Syntax
OR Immediate-Shifted Instructions		**r**A is loaded with the logical result (**r**S) OR (UIMM || 0x0000)	
oris	OR Immediate Shifted	CR0 is *not* updated.	**r**A,**r**S,UIMM
OR with Complement Instructions		**r**A is loaded with the logical result (**r**S) OR ¬(**r**B).	
orc	OR with Complement	CR0 is *not* updated.	**r**A,**r**S,**r**B
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
Equivalence Instructions		**r**A is loaded with the logical result ¬((**r**S) XOR (**r**B)).	
eqv	Equivalent	CR0 is *not* updated.	**r**A,**r**S,**r**B
eqv.	Equivalent and Record	CR0 is updated to reflect the result.	
XOR Instructions		**r**A is loaded with the logical result (**r**S) XOR (**r**B).	
xor	XOR	CR0 is *not* updated.	**r**A,**r**S,**r**B
xor.	XOR and Record	CR0 is updated to reflect the result.	
XOR-Immediate Instructions		**r**A is loaded with the logical result (**r**S) XOR UIMM.	
xori	XOR Immediate	CR0 is *not* updated.	**r**A,**r**S,UIMM
XOR Immediate-Shifted Instructions		**r**A is loaded with the logical result (**r**S) XOR (UIMM || 0x0000)	
xoris	XOR Immediate Shifted	CR0 is *not* updated.	**r**A,**r**S,UIMM

Sign-Extension Instructions

Table 3-32 shows the *sign-extension* instructions. These instructions sign-extend the value in the **r**S register and write the result in the **r**A 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		**r**A[24:31] is loaded with (**r**S[24:31]). The remaining bits **r**A[0:23] are each loaded with a copy of (**r**S[24]).	
extsb	Extend Sign Byte	CR0 is *not* updated.	**r**A,**r**S
extsb.	Extend Sign Byte and Record	CR0 is updated to reflect the result.	
Extend-Sign Halfword Instructions		**r**A[16:31] is loaded with (**r**S[16:31]). The remaining bits **r**A[0:15] are each loaded with a copy of (**r**S[16]).	
extsh	Extend Sign Halfword	CR0 is *not* updated.	**r**A,**r**S
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 **r**S register starting at bit 0. The count result is written to the **r**A 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
Count Leading-Zeros Instructions		**r**A is loaded with a count of leading zeros in **r**S.	
cntlzw	Count Leading Zeros Word	CR0 is *not* updated.	**r**A,**r**S
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, **crf**D, 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 **crf**D operand can be omitted if the comparison results are written to CR0. See **CRn Fields (Compare Instructions)**, page 362 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 828 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	**crf**D[LT,GT,EQ] are loaded with the result of algebraically comparing (**r**A) with (**r**B). CR[SO] is loaded with a copy of XER[SO].	**crf**D,0,**r**A,**r**B
cmpi	Compare Immediate	**crf**D[LT,GT,EQ] are loaded with the result of algebraically comparing (**r**A) with SIMM. CR[SO] is loaded with a copy of XER[SO].	**crf**D,0,**r**A,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	**crf**D[LT,GT,EQ] are loaded with the result of logically comparing (**r**A) with (**r**B). CR[SO] is loaded with a copy of XER[SO].	**crf**D,0,**r**A,**r**B
cmpli	Compare Logical Immediate	**crf**D[LT,GT,EQ] are loaded with the result of logically comparing (**r**A) with UIMM. CR[SO] is loaded with a copy of XER[SO].	**crf**D,0,**r**A,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. **r**S 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 829 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.

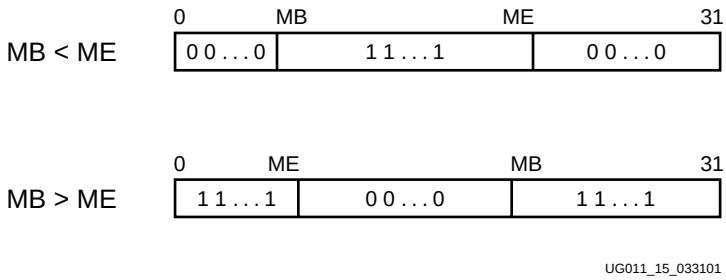


Figure 3-23: **Rotate Mask Generation**

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
Rotate Left then AND-with-Mask Immediate Instructions		**r**A is loaded with the masked result of left-rotating (**r**S) 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 *not* updated.	**r**A,**r**S,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
Rotate Left then AND-with-Mask Instructions		**r**A is loaded with the masked result of left-rotating (**r**S) the number of bits specified by (**r**B). The mask is specified by operands MB and ME.	
rlwnm	Rotate Left Word then AND with Mask	CR0 is *not* updated.	**r**A,**r**S,**r**B,MB,ME
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, **r**A, and the source register, **r**S. **r**S is left-rotated 16 bits and the result is written to **r**A after ANDing with the mask. This has the effect of extracting byte 0 from **r**S (**r**S[0:7]) and placing it in byte 2 of **r**A (**r**A[16:23]).

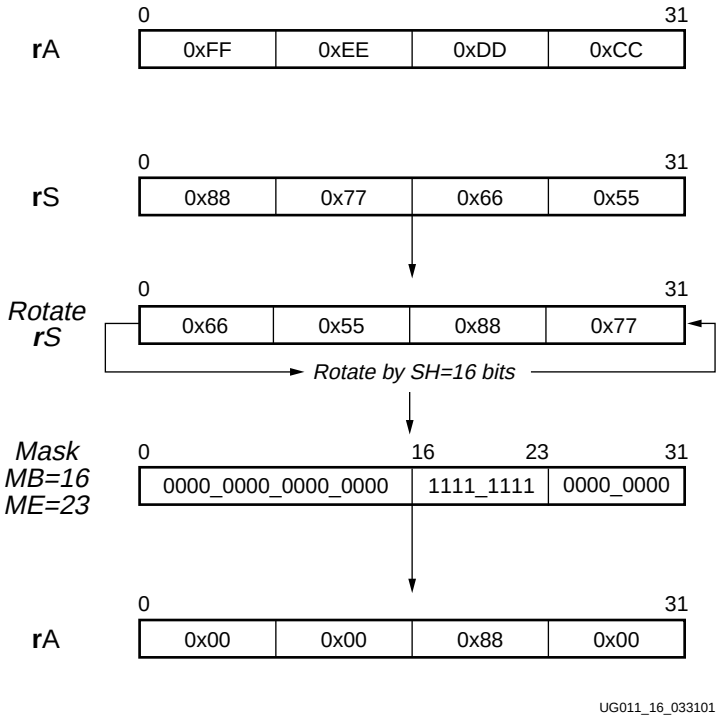


Figure 3-24: **Rotate Left then AND-with-Mask Immediate Example**

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
Rotate Left then Mask-Insert Immediate Instructions		The masked result of left-rotating (**r**S) the number of bits specified by SH is inserted into **r**A. The mask is specified by operands MB and ME.	
rlwimi	Rotate Left Word Immediate then Mask Insert	CR0 is *not* updated.	**r**A,**r**S,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, **r**A, and the source register, **r**S. **r**S is rotated 16 bits and the result is inserted into **r**A after ANDing with the mask. This has the effect of extracting byte 0 from **r**S (**r**S[0:7]) and inserting it into byte 2 of **r**A (**r**A[16:23]), leaving all remaining bytes in **r**A unmodified.

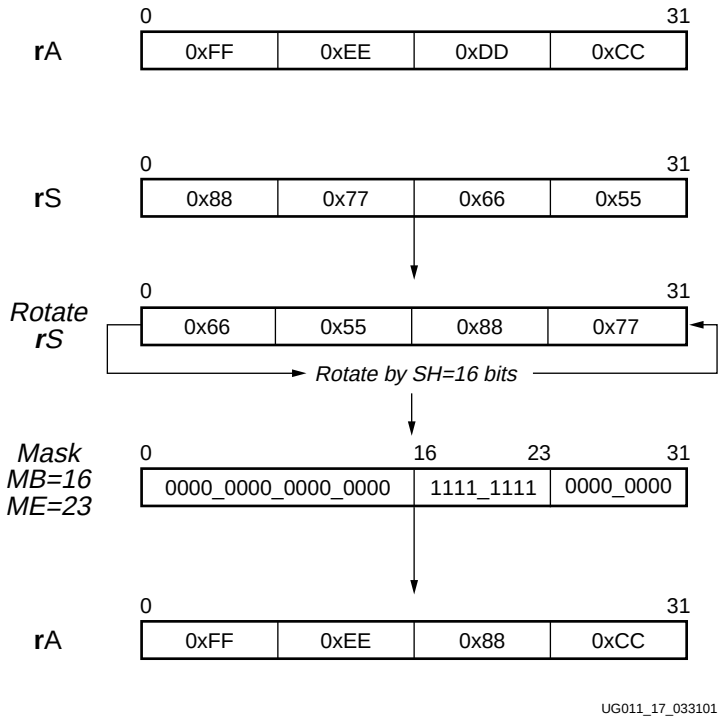


Figure 3-25: **Rotate Left then Mask-Insert Immediate Example**

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. **r**S 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 829 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		**r**A is loaded with the result of logically left-shifting (**r**S) the number of bits specified by (**r**B).	
slw	Shift Left Word	CR0 is *not* updated.	**r**A,**r**S,**r**B
slw.	Shift Left Word and Record	CR0 is updated to reflect the result.	
Shift-Right-Logical Instructions		**r**A is loaded with the result of logically right-shifting (**r**S) the number of bits specified by (**r**B).	
srw	Shift Right Word	CR0 is *not* updated.	**r**A,**r**S,**r**B
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.

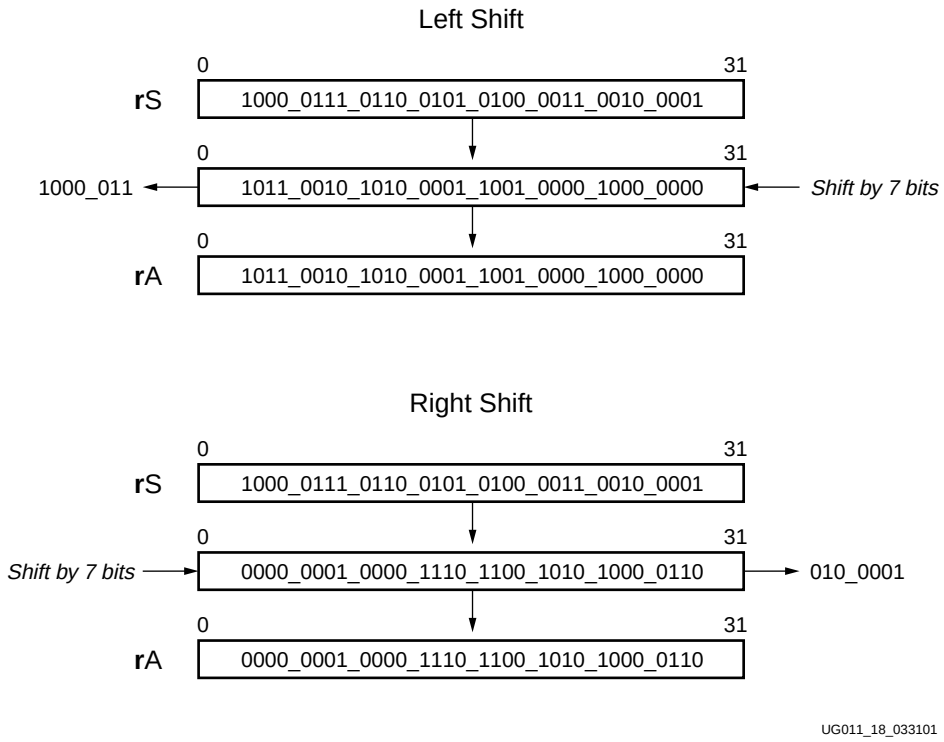


Figure 3-26: **Logical-Shift Examples**

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 840, for more information.

Table 3-39: **Algebraic-Shift Instructions**

Mnemonic	Name	Operation	Operand Syntax
Shift-Right-Algebraic Immediate Instructions		**r**A is loaded with the result of algebraically right-shifting (**r**S) the number of bits specified by SH.	
srawi	Shift Right Algebraic Word Immediate	CR0 is *not* updated. XER[CA] is updated to reflect the result.	**r**A,**r**S,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
Shift-Right-Algebraic Instructions		**r**A is loaded with the result of algebraically right-shifting (**r**S) the number of bits specified by (**r**B).	
sraw	Shift Right Algebraic Word	CR0 is *not* updated. XER[CA] is updated to reflect the result.	**r**A,**r**S,**r**B
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.

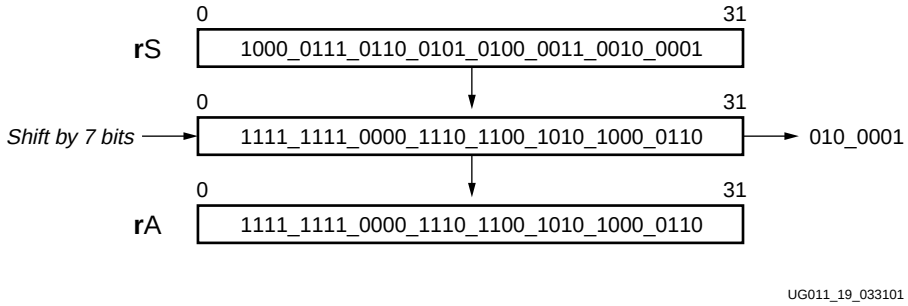


Figure 3-27: **Algebraic-Shift Example**

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 (**r**A[16:31]) and multiply it with the upper halfword of the second source operand (**r**B[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, **r**D, producing a 33-bit intermediate result. Generally, **r**D 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, **r**D 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
Multiply-Accumulate Cross-Halfword to Word Modulo Signed Instructions		**r**D is added to the signed product (**r**A[16:31]) × (**r**B[0:15]), producing a 33-bit result. The low-32 bits of this result are stored in **r**D.	
macchw	Multiply Accumulate Cross Halfword to Word Modulo Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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
Multiply-Accumulate Cross-Halfword to Word Saturate Signed Instructions		**r**D is added to the signed product (**r**A[16:31]) × (**r**B[0:15]), producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in **r**D. Otherwise, the nearest-representable value is stored in **r**D.	
macchws	Multiply Accumulate Cross Halfword to Word Saturate Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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.	
Multiply-Accumulate Cross-Halfword to Word Saturate Unsigned Instructions		**r**D is added to the unsigned product (**r**A[16:31]) × (**r**B[0:15]), producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in **r**D. Otherwise, the nearest-representable value is stored in **r**D.	
macchwsu	Multiply Accumulate Cross Halfword to Word Saturate Unsigned	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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.	
Multiply-Accumulate Cross-Halfword to Word Modulo Unsigned Instructions		**r**D is added to the unsigned product (**r**A[16:31]) × (**r**B[0:15]), producing a 33-bit result. The low-32 bits of this result are stored in **r**D.	
macchwu	Multiply Accumulate Cross Halfword to Word Modulo Unsigned	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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.

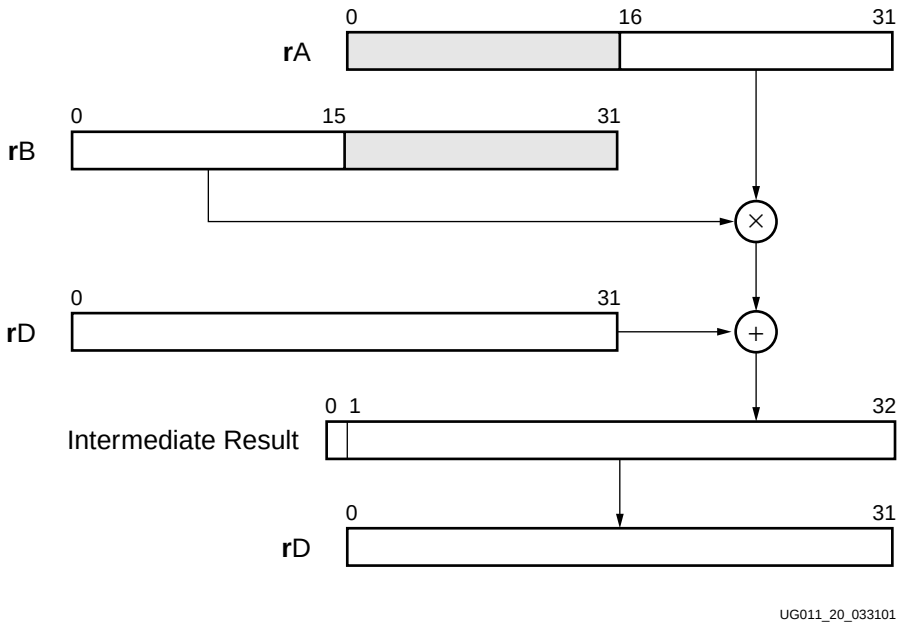


Figure 3-28: **Multiply-Accumulate Cross-Halfword to Word Operation**

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, **r**A[0:15] and **r**B[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, **r**D, producing a 33-bit intermediate result. Generally, **r**D 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, **r**D is loaded with the nearest representable value (see **Modulo and Saturating Arithmetic**, page 405).

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
Multiply-Accumulate High-Halfword to Word Modulo Signed Instructions		**r**D is added to the signed product (**r**A[0:15]) × (**r**B[0:15]), producing a 33-bit result. The low-32 bits of this result are stored in **r**D.	
machhw	Multiply Accumulate High Halfword to Word Modulo Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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
Multiply-Accumulate High-Halfword to Word Saturate Signed Instructions		**r**D is added to the signed product (**r**A[0:15]) × (**r**B[0:15]), producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in **r**D. Otherwise, the nearest-representable value is stored in **r**D.	
machhws	Multiply Accumulate High Halfword to Word Saturate Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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.	
Multiply-Accumulate High-Halfword to Word Saturate Unsigned Instructions		**r**D is added to the unsigned product (**r**A[0:15]) × (**r**B[0:15]), producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in **r**D. Otherwise, the nearest-representable value is stored in **r**D.	
machhwsu	Multiply Accumulate High Halfword to Word Saturate Unsigned	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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.	
Multiply-Accumulate High-Halfword to Word Modulo Unsigned Instructions		**r**D is added to the unsigned product (**r**A[0:15]) × (**r**B[0:15]), producing a 33-bit result. The low-32 bits of this result are stored in **r**D.	
machhwu	Multiply Accumulate High Halfword to Word Modulo Unsigned	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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.

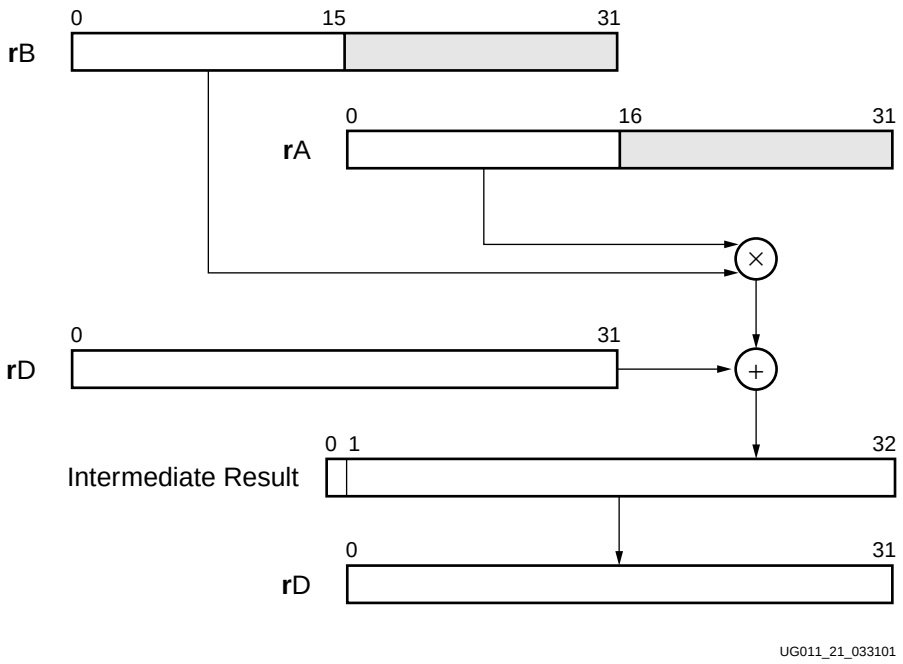


Figure 3-29: **Multiply-Accumulate High-Halfword to Word Operation**

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, **r**A[16:31] and **r**B[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, **r**D, producing a 33-bit intermediate result. Generally, **r**D 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, **r**D is loaded with the nearest representable value (see **Modulo and Saturating Arithmetic**, page 405).

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		**r**D is added to the signed product (**r**A[16:31]) × (**r**B[16:31]), producing a 33-bit result. The low-32 bits of this result are stored in **r**D.	
maclhw	Multiply Accumulate Low Halfword to Word Modulo Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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		**r**D is added to the signed product (**r**A[16:31]) × (**r**B[16:31]), producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in **r**D. Otherwise, the nearest-representable value is stored in **r**D.	
maclhws	Multiply Accumulate Low Halfword to Word Saturate Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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		**r**D is added to the unsigned product (**r**A[16:31]) × (**r**B[16:31]), producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in **r**D. Otherwise, the nearest-representable value is stored in **r**D.	
maclhwsu	Multiply Accumulate Low Halfword to Word Saturate Unsigned	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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
Multiply-Accumulate Low-Halfword to Word Modulo Unsigned Instructions		**r**D is added to the unsigned product (**r**A[16:31]) × (**r**B[16:31]), producing a 33-bit result. The low-32 bits of this result are stored in **r**D.	
maclhwu	Multiply Accumulate Low Halfword to Word Modulo Unsigned	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
maclhwu.	Multiply Accumulate Low Halfword to Word Modulo Unsigned and Record	CR0 is updated to reflect the result.	
maclhwuo	Multiply Accumulate Low Halfword to Word Modulo Unsigned with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
maclhwuo.	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.

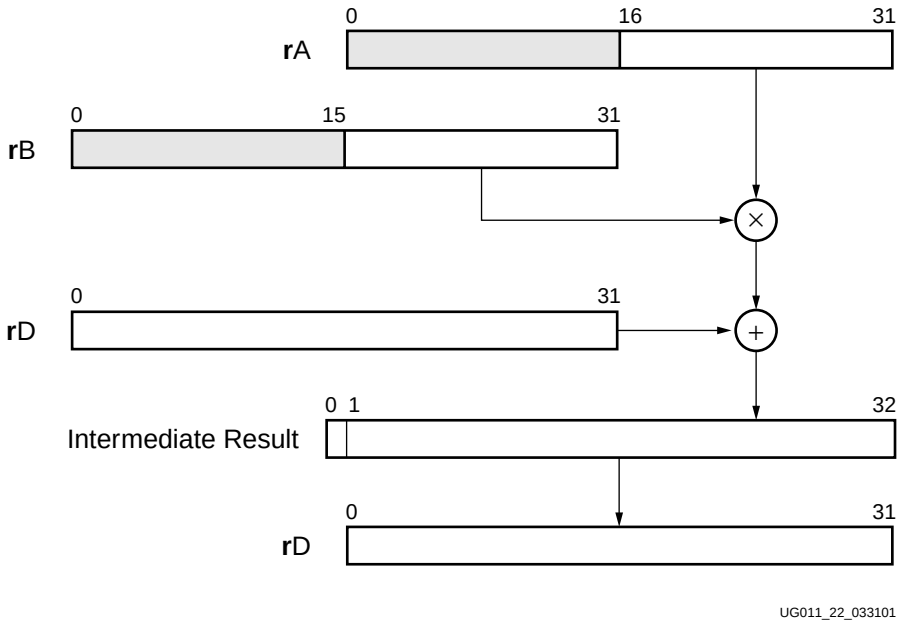


Figure 3-30: **Multiply-Accumulate Low-Halfword to Word Operation**

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 (**r**A[16:31]) and multiply it with the upper halfword of the second source operand (**r**B[0:15]), producing a signed 32-bit product. This product is negated and added to the value in the destination register, **r**D, producing a 33-bit intermediate result (this is the same as subtracting the product from **r**D). Generally, **r**D 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, **r**D 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
Negative Multiply-Accumulate Cross-Halfword to Word Modulo Signed Instructions		The signed product (**r**A[16:31]) × (**r**B[0:15]) is subtracted from **r**D, producing a 33-bit result. The low-32 bits of this result are stored in **r**D.	
nmacchw	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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.	
Negative Multiply-Accumulate Cross-Halfword to Word Saturate Signed Instructions		The signed product (**r**A[16:31]) × (**r**B[0:15]) is subtracted from **r**D, producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in **r**D. Otherwise, the nearest-representable value is stored in **r**D.	
nmacchws	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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.

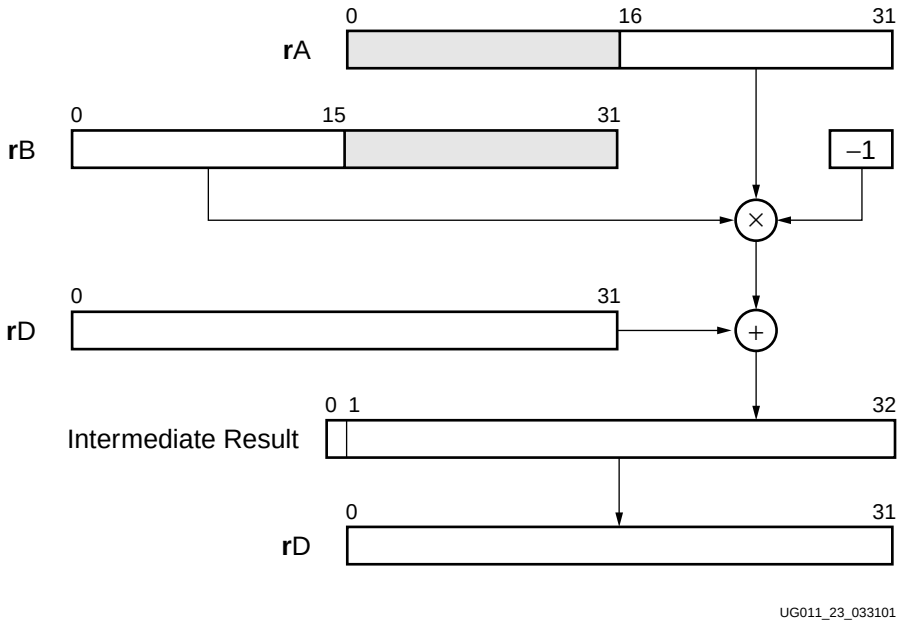


Figure 3-31: **Negative Multiply-Accumulate Cross-Halfword to Word Operation**

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, **r**A[0:15] and **r**B[0:15], producing a signed 32-bit product. This product is negated and added to the value in the destination register, **r**D, producing a 33-bit intermediate result (this is the same as subtracting the product from **r**D). Generally, **r**D 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, **r**D is loaded with the nearest representable value (see **Modulo and Saturating Arithmetic**, page 405).

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
Negative Multiply-Accumulate High-Halfword to Word Modulo Signed Instructions		The signed product (**r**A[0:15]) × (**r**B[0:15]) is subtracted from **r**D, producing a 33-bit result. The low-32 bits of this result are stored in **r**D.	
nmachhw	Negative Multiply Accumulate High Halfword to Word Modulo Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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.	
Negative Multiply-Accumulate High-Halfword to Word Saturate Signed Instructions		The signed product (**r**A[0:15]) × (**r**B[0:15]) is subtracted from **r**D, producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in **r**D. Otherwise, the nearest-representable value is stored in **r**D.	
nmachhws	Negative Multiply Accumulate High Halfword to Word Saturate Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
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.

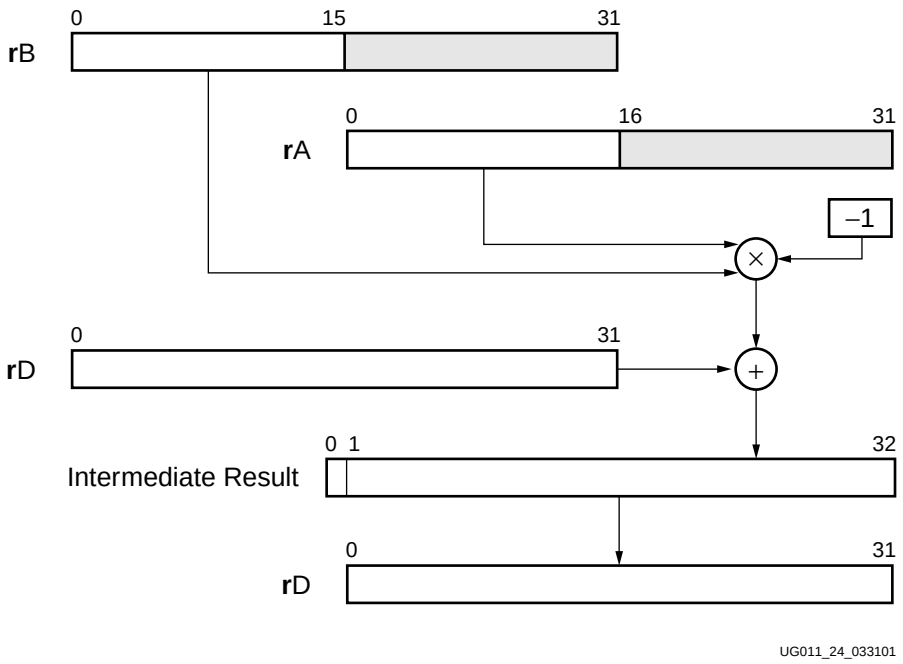


Figure 3-32: **Negative Multiply-Accumulate High-Halfword to Word Operation**

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, **r**A[16:31] and **r**B[16:31], producing a signed 32-bit product. This product is negated and added to the value in the destination register, **r**D, producing a 33-bit intermediate result (this is the same as subtracting the product from **r**D). Generally, **r**D 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, **r**D is loaded with the nearest representable value (see **Modulo and Saturating Arithmetic**, page 405).

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
Negative Multiply-Accumulate Low-Halfword to Word Modulo Signed Instructions		The signed product (**r**A[16:31]) × (**r**B[16:31]) is subtracted from **r**D, producing a 33-bit result. The low-32 bits of this result are stored in **r**D.	
nmaclhw	Negative Multiply Accumulate Low Halfword to Word Modulo Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
nmaclhw.	Negative Multiply Accumulate Low Halfword to Word Modulo Signed and Record	CR0 is updated to reflect the result.	
nmaclhwo	Negative Multiply Accumulate Low Halfword to Word Modulo Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
nmaclhwo.	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.	
Negative Multiply-Accumulate Low-Halfword to Word Saturate Signed Instructions		The signed product (**r**A[16:31]) × (**r**B[16:31]) is subtracted from **r**D, producing a 33-bit result. If the result does not overflow, the low-32 bits of this result are stored in **r**D. Otherwise, the nearest-representable value is stored in **r**D.	
nmaclhws	Negative Multiply Accumulate Low Halfword to Word Saturate Signed	XER and CR0 are *not* updated.	**r**D,**r**A,**r**B
nmaclhws.	Negative Multiply Accumulate Low Halfword to Word Saturate Signed and Record	CR0 is updated to reflect the result.	
nmaclhwso	Negative Multiply Accumulate Low Halfword to Word Saturate Signed with Overflow Enabled	XER[OV,SO] are updated to reflect the result.	
nmaclhwso.	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.

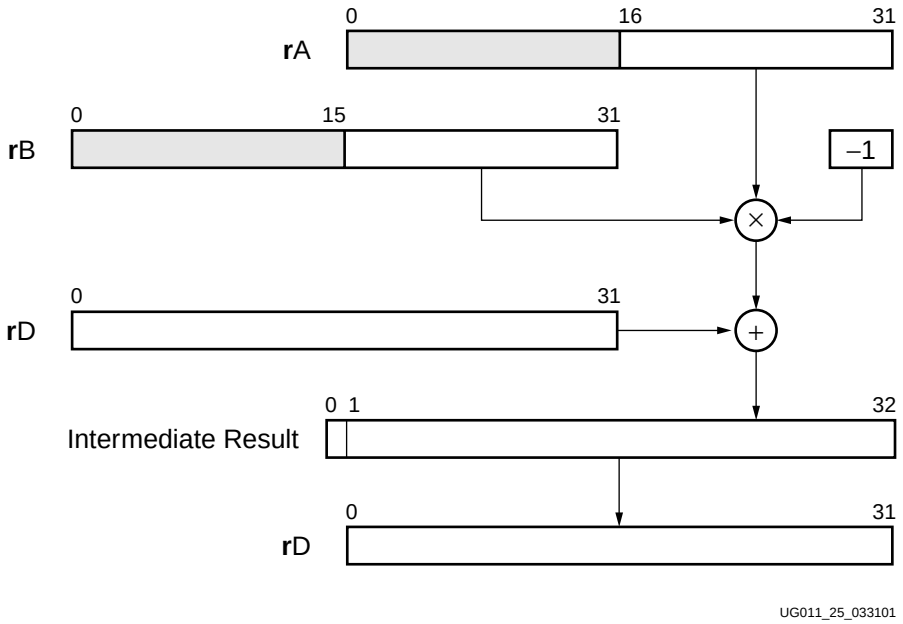


Figure 3-33: **Negative Multiply-Accumulate Low-Halfword to Word Operation**

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 (**r**A[16:31]) and multiply it with the upper halfword of the second source operand (**r**B[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
Multiply Cross-Halfword to Word Signed Instructions		**r**D is loaded with the signed product (**r**A[16:31]) × (**r**B[0:15]).	
mulchw	Multiply Cross Halfword to Word Signed	CR0 is *not* updated.	**r**D,**r**A,**r**B
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
Multiply Cross-Halfword to Word Unsigned Instructions		**r**D is loaded with the unsigned product (**r**A[16:31]) × (**r**B[0:15]).	
mulchwu	Multiply Cross Halfword to Word Unsigned	CR0 is *not* updated.	**r**D,**r**A,**r**B
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.

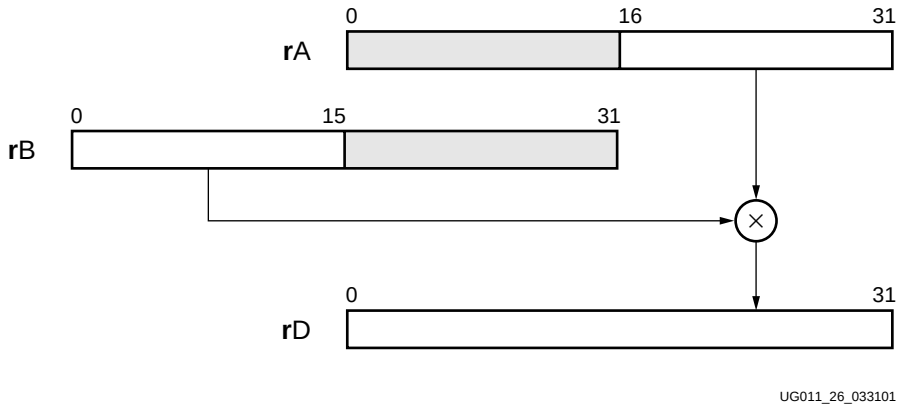


Figure 3-34: **Multiply Cross-Halfword to Word Operation**

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, **r**A[0:15] and **r**B[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
Multiply High-Halfword to Word Signed Instructions		**r**D is loaded with the signed product (**r**A[0:15]) × (**r**B[0:15]).	
mulhhw	Multiply High Halfword to Word Signed	CR0 is *not* updated.	**r**D,**r**A,**r**B
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
Multiply High-Halfword to Word Unsigned Instructions		**r**D is loaded with the unsigned product (**r**A[0:15]) × (**r**B[0:15]).	
mulhhwu	Multiply High Halfword to Word Unsigned	CR0 is *not* updated.	**r**D,**r**A,**r**B
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.

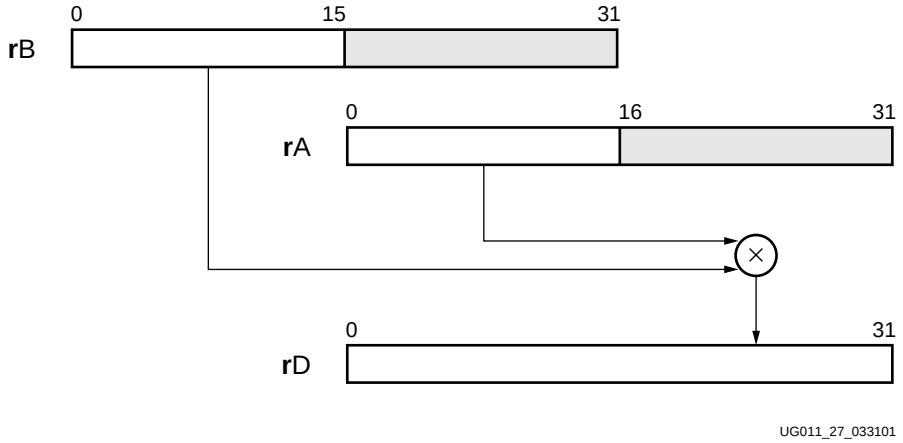


Figure 3-35: **Multiply High-Halfword to Word Operation**

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, **r**A[16:31] and **r**B[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
Multiply Low-Halfword to Word Signed Instructions		**r**D is loaded with the signed product (**r**A[16:31]) × (**r**B[16:31]).	
mullhw	Multiply Low Halfword to Word Signed	CR0 is *not* updated.	**r**D,**r**A,**r**B
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
Multiply Low-Halfword to Word Unsigned Instructions		**r**D is loaded with the unsigned product (**r**A[16:31]) × (**r**B[16:31]).	
mullhwu	Multiply Low Halfword to Word Unsigned	CR0 is *not* updated.	**r**D,**r**A,**r**B
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.

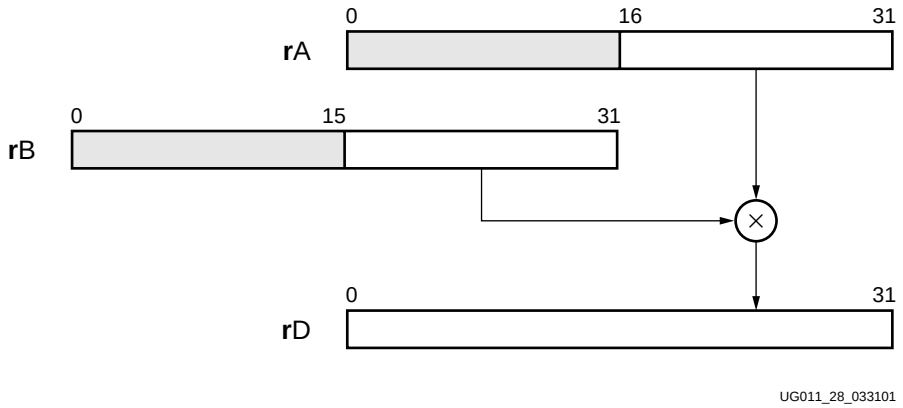


Figure 3-36: **Multiply Low-Halfword to Word Operation**

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 511, 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 376, for information on instructions used to manipulate bits and fields in the CR. See **Conditional Branch Control**, page 367, 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 **crf**D 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.	**crf**D
mfcr	Move from Condition Register	**r**D is loaded with the contents of CR.	**r**D
mtcrf	Move to Condition Register Fields	CR is loaded with the contents of **r**S under the control of a field mask specified by the CRM operand.	CRM,**r**S

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 **r**S. 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 **r**S only when the associated CRM mask bit is set to 1. If the mask bit is cleared to 0, the CR field is unchanged.

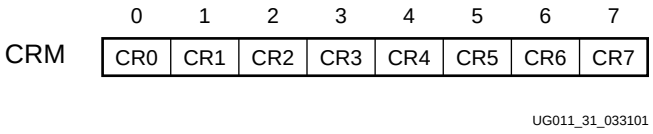


Figure 3-37: **mtcrf Field Mask (CRM) Format**

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 **r**S. All remaining CR fields are unchanged.

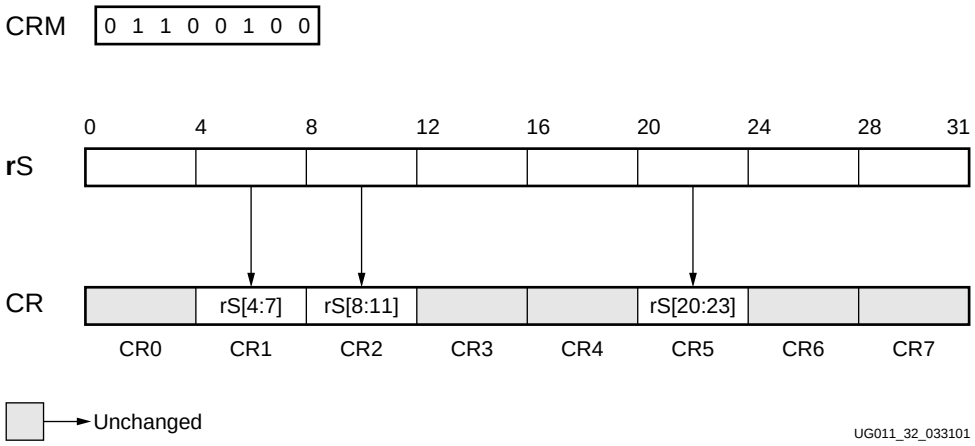


Figure 3-38: **mtcrf Example**

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 571.

Table 3-50: **Special-Purpose Register Instructions**

Mnemonic	Name	Operation	Operand Syntax
mfspr	Move from Special Purpose Register	**r**D is loaded with the contents of the SPR specified by SPRN.	**r**D,SPRN
mtspr	Move to Special Purpose Register	The SPR specified by SPRN is loaded with the contents of **r**S.	SPRN,**r**S

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 342.

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 *not* 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	**eieio**[1]	**eieio**
rfci[2]	**isync**	**sync**
rfi[2]	**mtmsr**[2]	
sc	**rfci**[2]	
	rfi[2]	
	sc	
	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 837.

Table 3-53: **Semaphore Synchronization Instructions**

Mnemonic	Name	Operation	Operand Syntax
lwarx	Load Word and Reserve Indexed	**r**D is loaded with the word in memory addressed using register-indirect with index addressing: EA = (**r**A | 0) + (**r**B) A reservation corresponding to the address is maintained by the processor.	**r**D,**r**A,**r**B
stwcx.	Store Word Conditional Indexed	An effective address is computed using register-indirect with index addressing: EA = (**r**A | 0) + (**r**B) If a reservation exists, the contents of **r**S are stored into the memory word specified by the effective address, and the reservation is cleared. If a reservation does not exist, **r**S is not stored. CR0[EQ] is set to 1 if the reservation exists, otherwise it is cleared to 0.	**r**S,**r**A,**r**B

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 456 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

Chapter 4

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 524.

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 456.
- The processor ID register (PID) is described in **Virtual Mode**, page 472.
- The zone-protection register (ZPR) is described in **Virtual-Mode Access Protection**, page 482.
- The storage-attribute control registers are described in **Memory-System Control**, page 451.
- The exception-handling registers are described in **Interrupt-Handling Registers**, page 497.

- The debug registers are described in **Debug Registers**, page 537.
- The timer registers, including the time base, are described in **Timer Resources**, page 523.

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 770 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 435, 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 830, 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 435, 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 **rfci**), and by the exception mechanism during a control transfer to an interrupt handler.

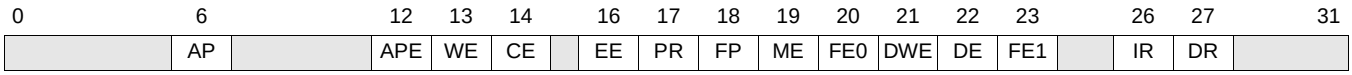


Figure 4-2: **Machine-State Register (MSR)**

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 436, 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 502, 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 502, 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 343, 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 504, 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 537, for more information.
22	DE	Debug Interrupt Enable 0—Disabled. 1—Enabled.	Controls the debug interrupt. See **Debug Interrupt (0x2000)**, page 521, 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 471, 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 471, for an introduction.
28:31		Reserved	

The initial state of the MSR following a processor reset is described in **Machine-State Register**, page 562.

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.

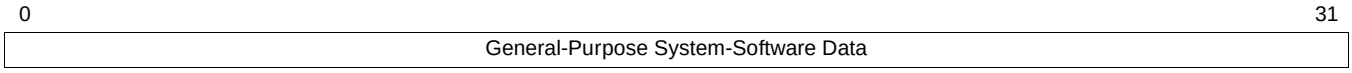


Figure 4-3: **SPR General-Purpose Registers (SPRG0–SPRG7)**

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 **mfspr** and **mtspr** instructions. User-mode software that reads SPRG4–SPRG7 accesses them using different SPR numbers (see page 365).

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.

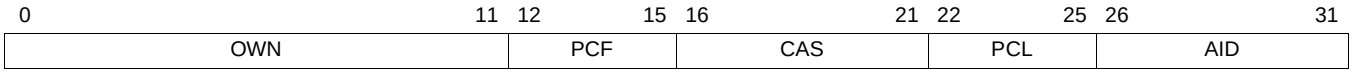


Figure 4-4: **Processor-Version Register (PVR)**

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 **mfspr** 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	mfspr(1)	dcread	tlbsx
	mtdcr	iccci	tlbsync
	mtmsr	icread	tlbwe
	mtspr(2)		
	wrtee		
	wrteei		

Notes:
1. Except for CTR, LR, SPRG4–SPRG7, and XER.
2. 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 494, for more information.	—
rfci	Return from Critical Interrupt	Return from critical-interrupt handler. See **Returning from Interrupt Handlers**, page 494, for more information.	—
sc	System Call	Causes a system-call exception to occur. See **System-Call Interrupt (0x0C00)**, page 514, 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 342, for more information.

Table 4-5: **Machine-State Register Instructions**

Mnemonic	Name	Operation	Operand Syntax
mfmsr	Move from Machine State Register	**r**D is loaded with the contents of the machine-state register.	**r**D
mtmsr	Move to Machine State Register	The machine-state register is loaded with the contents of **r**S.	**r**S
wrtee	Write External Enable	MSR[EE] (bit 16) is loaded with the value in $\mathbf{r}S_{16}$.	**r**S
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 571.

Table 4-6: **Special-Purpose Register Instructions**

Mnemonic	Name	Operation	Operand Syntax
mfspr	Move from Special Purpose Register	**r**D is loaded with the contents of the SPR specified by SPRN.	**r**D,SPRN
mtspr	Move to Special Purpose Register	The SPR specified by SPRN is loaded with the contents of **r**S.	SPRN,**r**S

Simplified instruction mnemonics are available for the **mtspr** and **mfspr** instructions when accessing certain SPRs. See **Special-Purpose Registers**, page 830, 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 571.

Table 4-7: **Device Control Register Instructions**

Mnemonic	Name	Operation	Operand Syntax
mfdcr	Move from Device Control Register	**r**D is loaded with the contents of the DCR specified by DCRN.	**r**D,DCRN
mtdcr	Move to Device Control Register	The DCR specified by DCRN is loaded with the contents of **r**S.	DCRN,**r**S

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.

Chapter 5

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 implemenation 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.

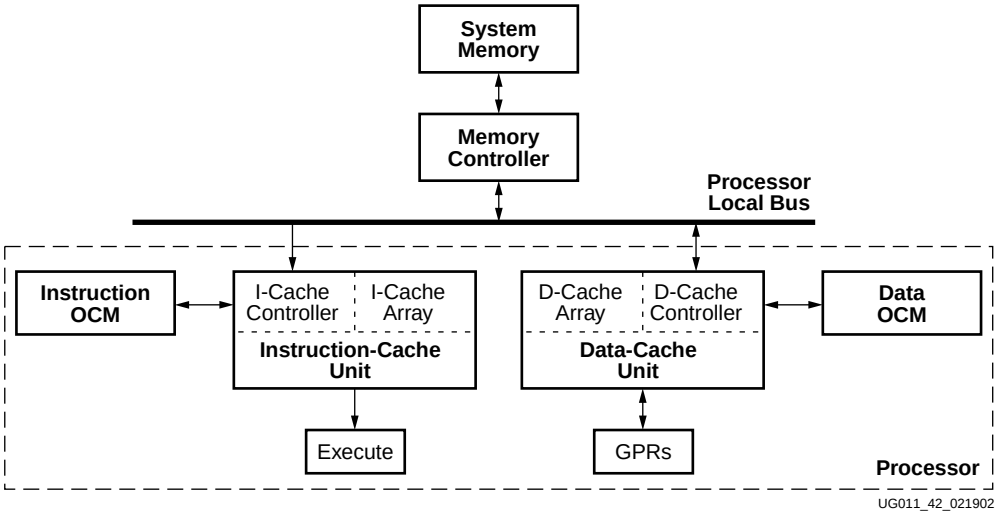


Figure 5-1: **PPC405 Memory-System Organization**

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.

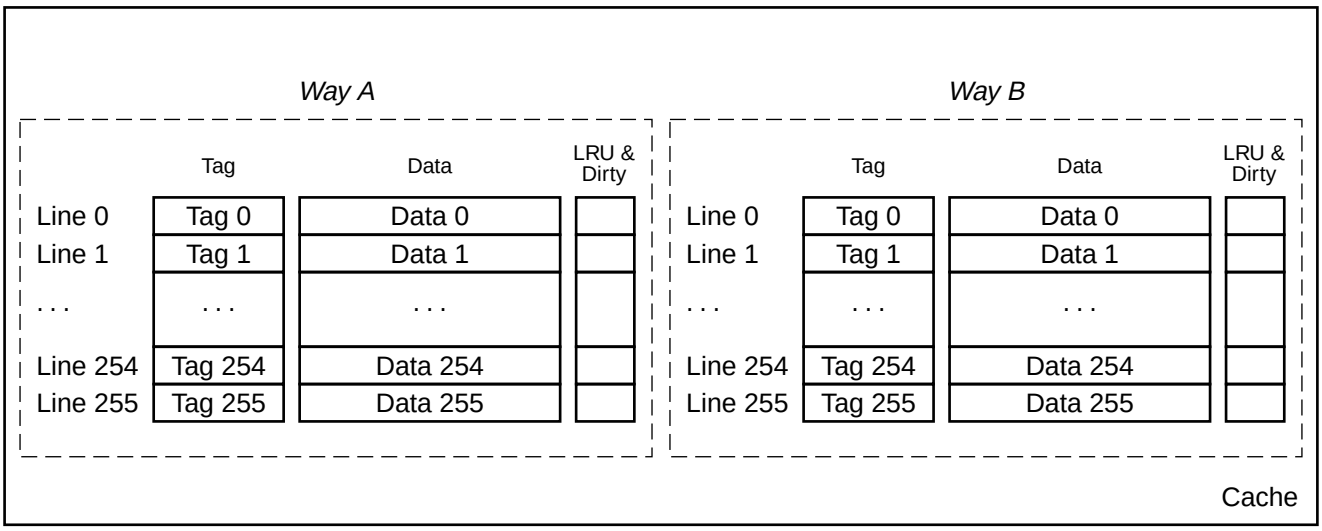


Figure 5-2: **Logical Structure of the PPC405 Cache Arrays**

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 442). Figure 5-3 shows the address fields used in accessing the two caches.

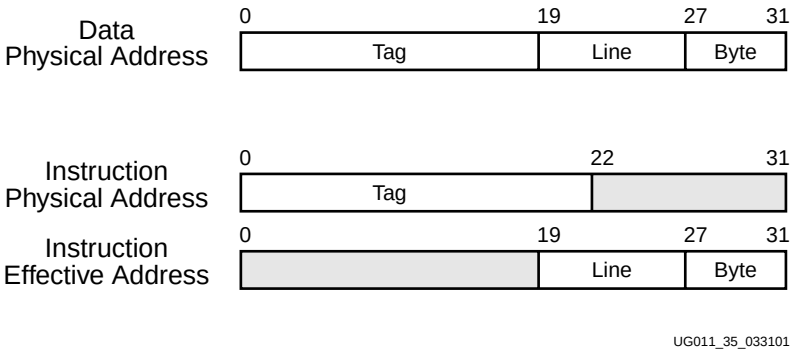


Figure 5-3: **Address Fields Used to Access Caches**

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.

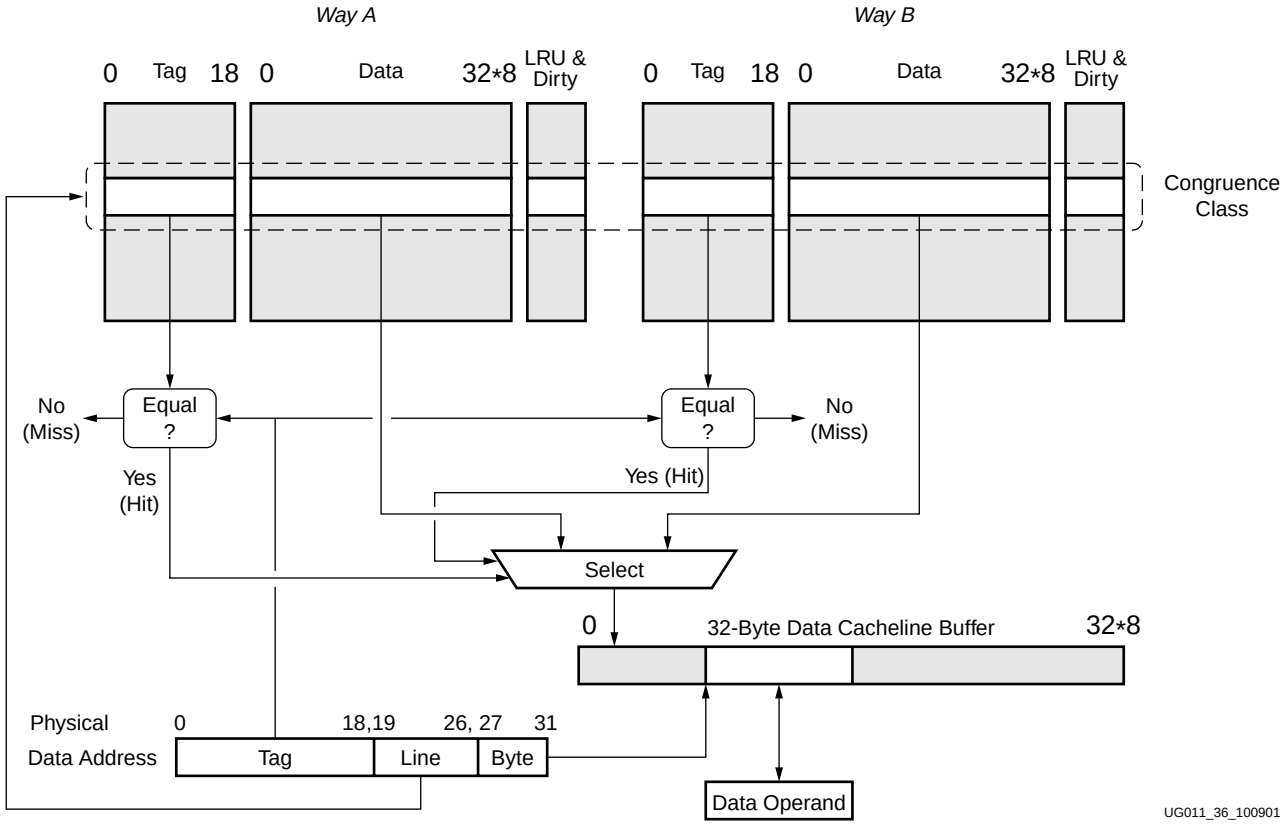


Figure 5-4: **Data-Cache Access Example**

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.

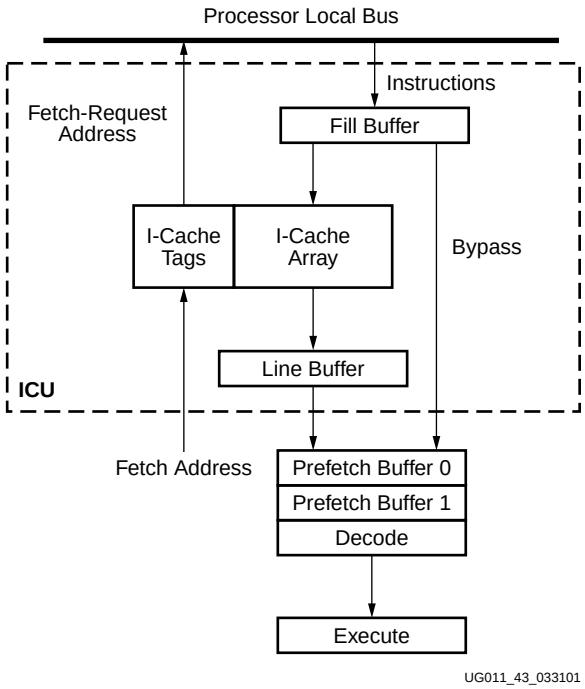


Figure 5-5: **Instruction Flow from the Instruction-Cache Unit**

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 459). 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 454, 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 478, 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 456, for information on this instruction). After the cache is invalidated, the ICCR can be configured.

Core-Configuration Register, page 459, 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.

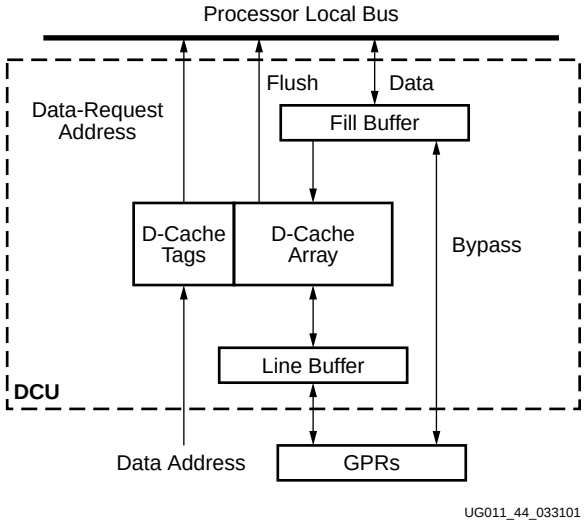


Figure 5-6: **Data Flow to/from the Data-Cache Unit**

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 459). 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 454, 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 456, 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 478, 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 463, 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 453, 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 478, 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 459, 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 (DPP_0) is controlled by the DCU. The low-order bit (DPP_1) can be controlled by software using the DDP1 field in the CCR0 register. See Table 5-6, page 460, for more information on using this CCR0 field.

Table 5-1 shows the conditions under which the DCU asserts and deasserts DPP_0. As is shown in the table, loads from system memory have highest priority and always immediately assert DPP_0.

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 463, 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 **lmw**, **stmw**, **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 424, 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 452). 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 478). In real mode, the storage-attribute control registers are used to define the storage attributes (see **Storage-Attribute Control Registers**, page 452).

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 452, 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 463, 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 449 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 459). 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 506 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 349 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 478 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.

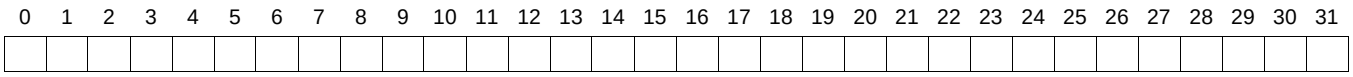


Figure 5-7: **Data-Cache Write-Through Register (DCWR)**

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.

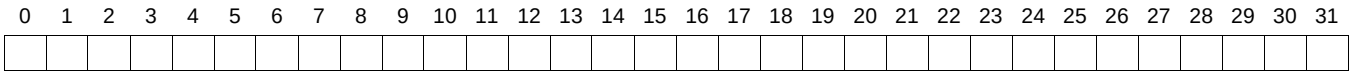


Figure 5-8: **Data-Cache Cacheability Register (DCCR)**

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 456 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 451 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.

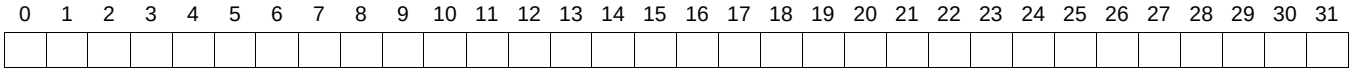


Figure 5-9: **Instruction-Cache Cacheability Register (ICCR)**

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 456 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 451 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 449 for more information.

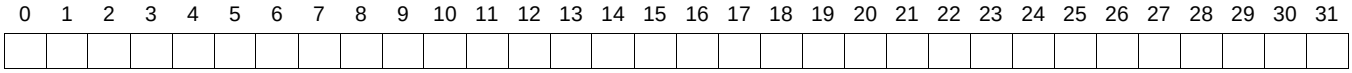


Figure 5-10: **Storage Guarded Register (SGR)**

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 **mfspr** and **mtspr** 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.

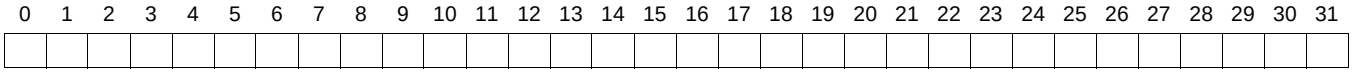


Figure 5-11: **Storage User-Defined 0 Register (SU0R)**

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 459). 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 506 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 **mfspr** and **mtspr** 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 349 for more information on big-endian and little-endian memory accesses.

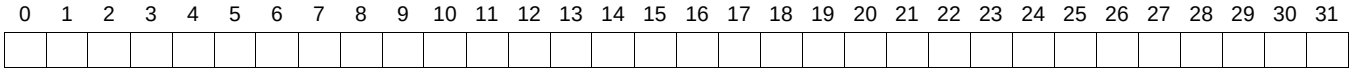


Figure 5-12: **Storage Little-Endian Register (SLER)**

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 **mfspr** and **mtspr** 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 = (**r**A | 0) + (**r**B)	**r**A,**r**B
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 = (**r**A | 0) + (**r**B)	**r**A,**r**B
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 468 for more information. EA is calculated using register-indirect with index addressing: EA = (**r**A | 0) + (**r**B)	**r**A,**r**B

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 = (**r**A | 0) + (**r**B) 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.	**r**A,**r**B
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 = (**r**A | 0) + (**r**B)	**r**A,**r**B
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 = (**r**A | 0) + (**r**B)	**r**A,**r**B
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 = (**r**A | 0) + (**r**B)	**r**A,**r**B
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 = (**r**A | 0) + (**r**B)	**r**A,**r**B
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 = (**r**A | 0) + (**r**B)	**r**A,**r**B

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 = (**r**A | 0) + (**r**B) 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.	**r**A,**r**B
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 = (**r**A | 0) + (**r**B)	**r**A,**r**B
dcread	Data Cache Read	If the byte specified by the effective address (EA) is cached by the data cache, **r**D 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 **r**D. See **dcread Instruction**, page 469 for more information. EA is calculated using register-indirect with index addressing: EA = (**r**A | 0) + (**r**B)	**r**D,**r**A,**r**B

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.

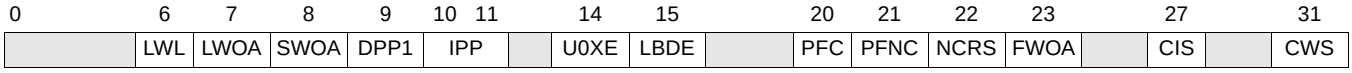


Figure 5-13: **Core-Configuration Register (CCR0)**

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 461 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 461 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 506 for more information.
15	LDBE	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 468 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 468 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 **r**N, **r**M, **r**X, and **r**Z 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.
mfspr  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.
mfspr  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				
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	Data (Words)			
—	No	No	—	—	—	—
—	No	No	—	—	—	—
—	No	No	—	—	—	—
—	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*.

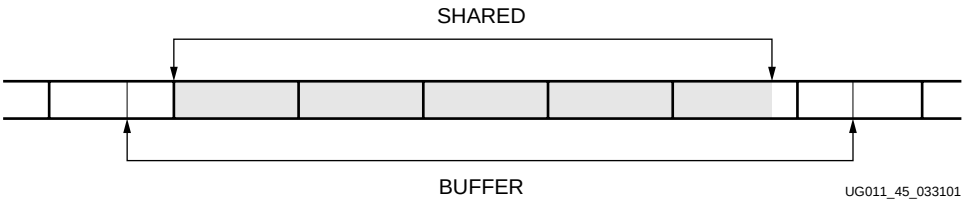


Figure 5-14: **Example of Shared-Memory Allocation**

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.
    mtctrr3

    loop:
    dcbz  0,r1            ! Flush one way of the cache set.
    dcbz  r2,r1           ! Flush the other way of the cache set.
    dccci0,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
    mtctrr3

  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 an 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 459). 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.

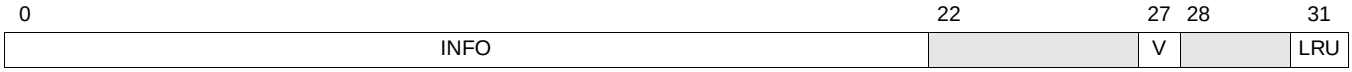


Figure 5-15: **Instruction-Cache Debug-Data Register (ICDBDR)**

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 **mfspr** that reads the ICDBDR contents. This guarantees that the values read by **mfspr** 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, **r**D. Figure 5-15 shows the format of the cache information loaded into **r**D. The information fields loaded in **r**D are defined as shown in Table 5-8.

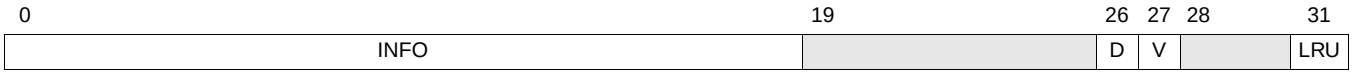


Figure 5-16: **Information Fields Loaded by dcread into rD**

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.

Chapter 6

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 344). 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 452, 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).

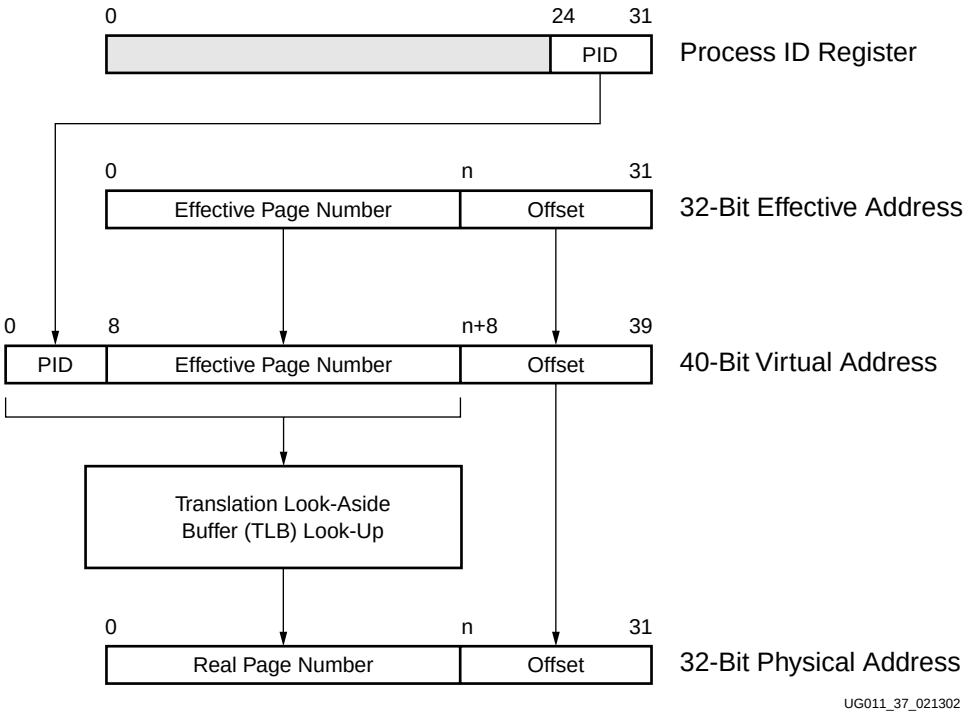


Figure 6-1: **Virtual-Mode Address Translation**

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 474). 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 478.

System software maintains a page-translation table that contains entries used to translate each virtual page into a physical page (see page 474). 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 478, 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 475, 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).

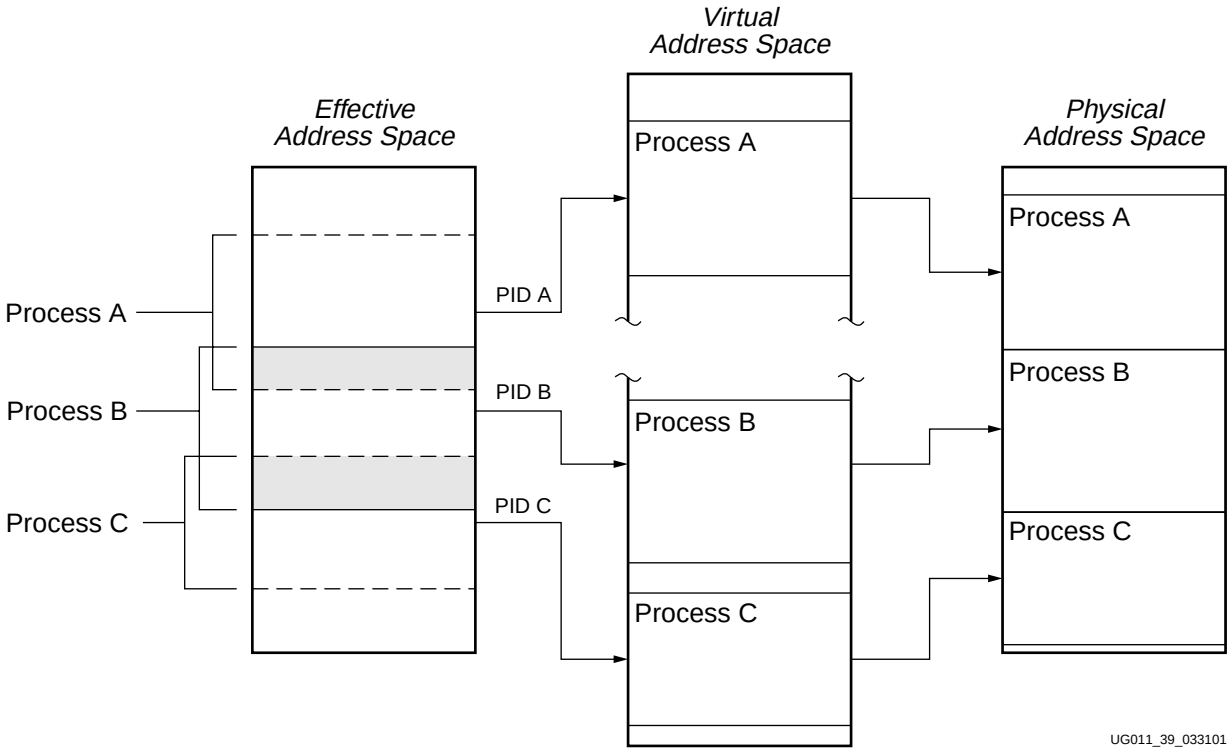


Figure 6-2: **Process-Mapping Example**

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.

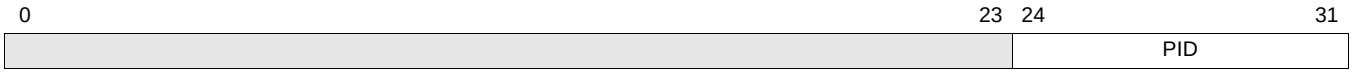


Figure 6-3: **Process-ID Register (PID)**

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 **mfspr** and **mtspr** 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.

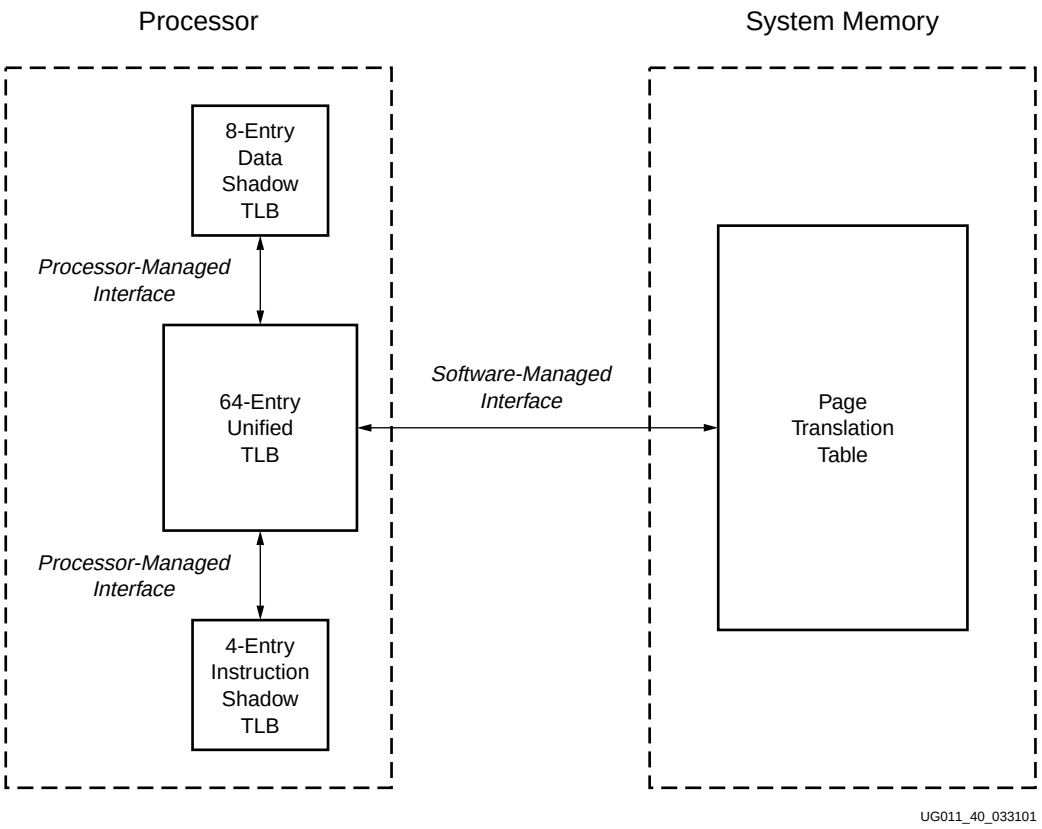


Figure 6-4: **Page-Translation Table and TLB Organization**

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.

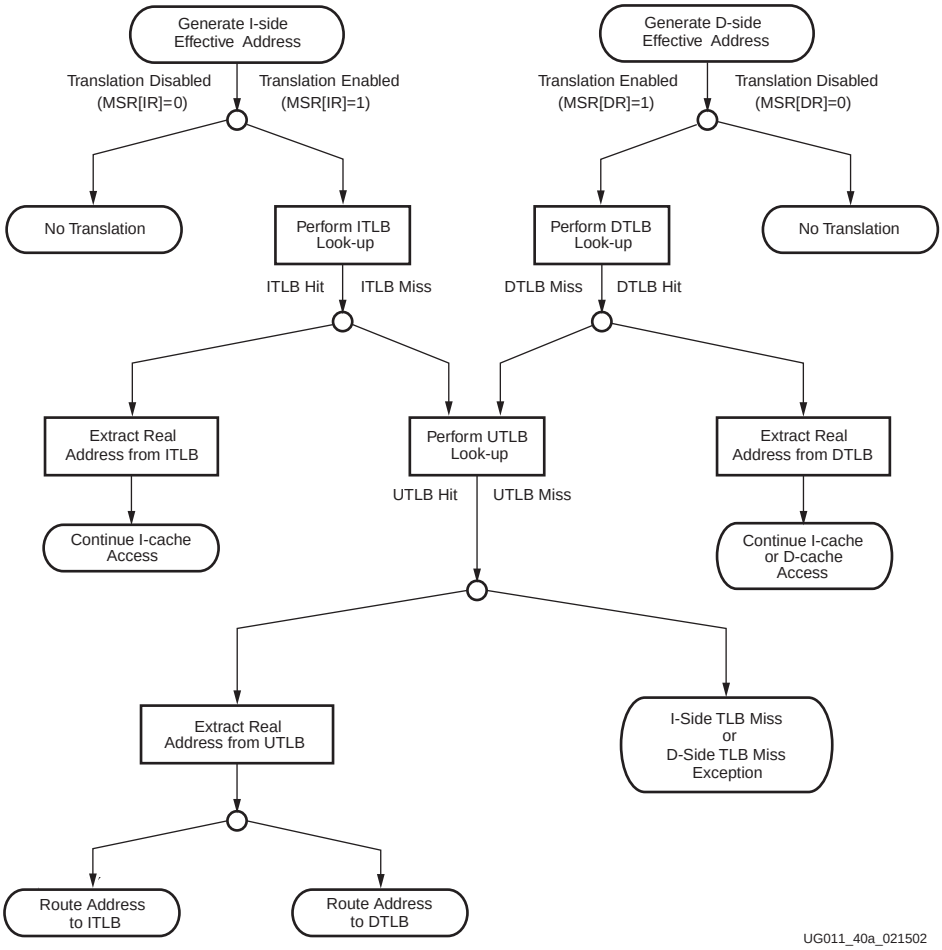


Figure 6-5: **ITLB/DTLB/UTLB Address Translation Flow**

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 487, 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.

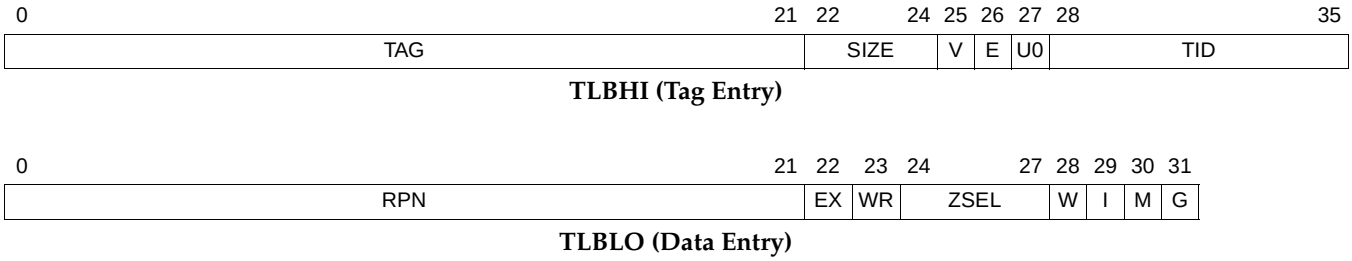


Figure 6-6: **TLB-Entry Format**

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 478, 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 478. 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 478, 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 482, 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 349, 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 452, 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 451, 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 451, 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 452, for more information.

In real mode, the storage-attribute control registers are used to define storage attributes. See **Storage-Attribute Control Registers**, page 452 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 472.

When assigning sizes to instruction pages, software must be careful to avoid creating the opportunity for instruction-cache synonyms. See **Instruction-Cache Synonyms**, page 442, 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	*n* (Figure 6-1)
1 KB	0b000	$TAG_{0:21} \leftrightarrow EA_{0:21}$	$EA_{22:31}$	$RPN_{0:21}$	—	22
4 KB	0b001	$TAG_{0:19} \leftrightarrow EA_{0:19}$	$EA_{20:31}$	$RPN_{0:19}$	20:21	20
16 KB	0b010	$TAG_{0:17} \leftrightarrow EA_{0:17}$	$EA_{18:31}$	$RPN_{0:17}$	18:21	18
64 KB	0b011	$TAG_{0:15} \leftrightarrow EA_{0:15}$	$EA_{16:31}$	$RPN_{0:15}$	16:21	16
256 KB	0b100	$TAG_{0:13} \leftrightarrow EA_{0:13}$	$EA_{14:31}$	$RPN_{0:13}$	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	*n* (Figure 6-1)
1 MB	0b101	$TAG_{0:11} \leftrightarrow EA_{0:11}$	$EA_{12:31}$	$RPN_{0:11}$	12:21	12
4 MB	0b110	$TAG_{0:9} \leftrightarrow EA_{0:9}$	$EA_{10:31}$	$RPN_{0:9}$	10:21	10
16 MB	0b111	$TAG_{0:7} \leftrightarrow EA_{0:7}$	$EA_{8:31}$	$RPN_{0:7}$	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 480.

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 478). 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 482, 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 478, for more information.

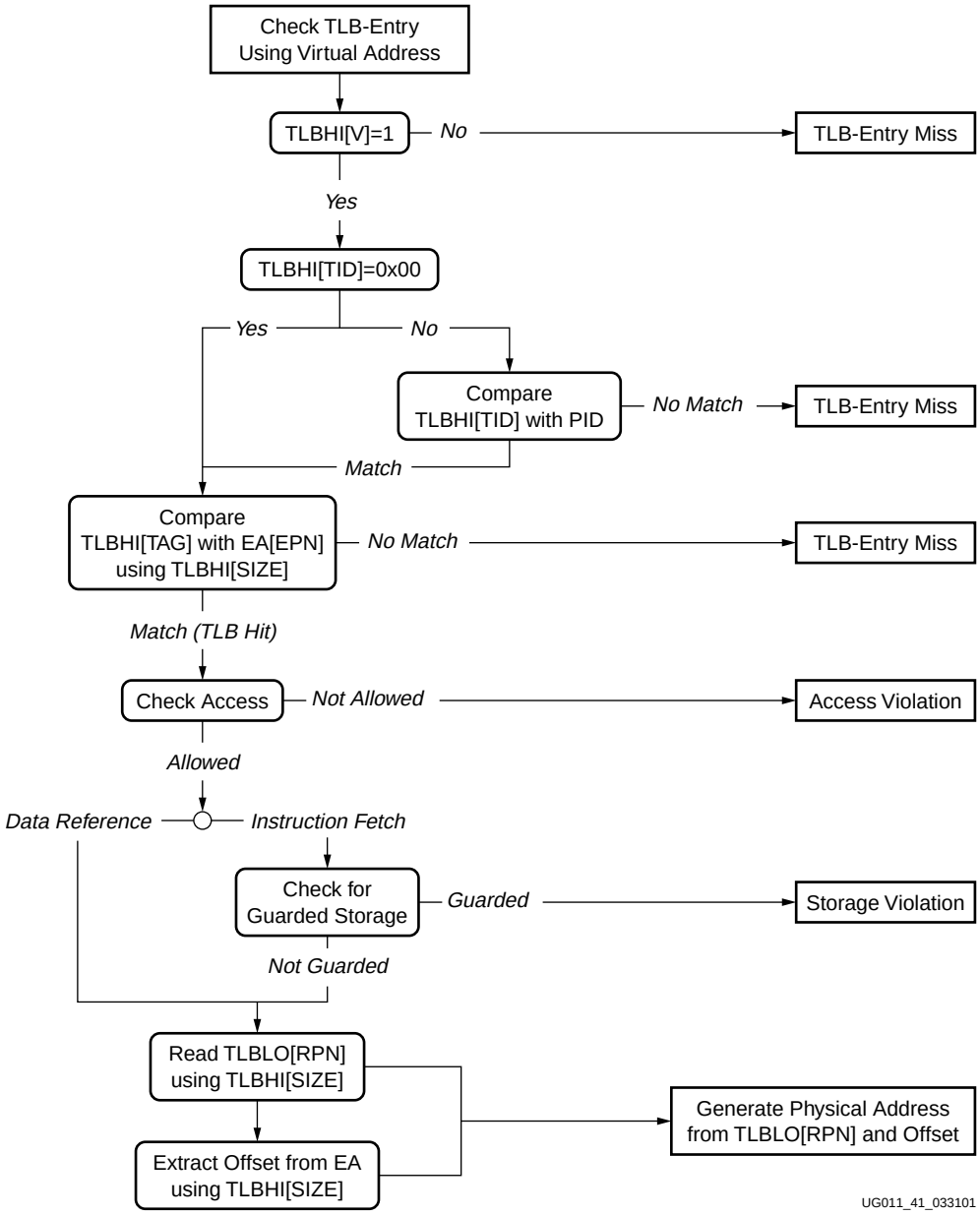


Figure 6-7: **General Process for Examining a TLB Entry**

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[Z*n*]=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[Z*n*]≠ 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[Z*n*]≠ 10 and ZPR[Z*n*]≠ 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 506, for more information on this exception and **Zone Protection**, page 482, 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[Z*n*]=00).
  - The TLB entry specifies a non-executable page (TLBLO[EX]=0) that is not otherwise overridden by the zone field (ZPR[Z*n*]≠ 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[Z*n*]≠ 10 and ZPR[Z*n*]≠ 11).
  - The TLB entry specifies a guarded-storage page (TLBLO[G]=1).

See **Instruction-Storage Interrupt (0x0400)**, page 508, for more information on this exception, **Guarded (G)**, page 452, for more information on guarded storage, and **Zone Protection**, page 482, 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 519, 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 520, 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[Z*n*] = 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[Z*n*]=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[Z*n*]=00 in user mode only.
  - **dcbst**—Affected by ZPR[Z*n*]=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[Z*n*]=00.
  - **dcbt**—Affected by ZPR[Z*n*]=00 in user mode only. This instruction is treated as a load and is therefore not affected by TLBLO[WR].
  - **dcbtst**—Affected by ZPR[Z*n*]=00 in user mode only. This instruction is treated as a load and is therefore not affected by TLBLO[WR].
  - **icbt**—Affected by ZPR[Z*n*]=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[Z*n*]=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[Z*n*]=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[Z*n*]=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	**r**A contains an index value ranging from 0 to 63. Part of the UTLB entry specified by the index in **r**A is loaded into **r**D. If WS=0, the tag portion (TLBHI) is loaded into **r**D *and* the PID is updated with the TLBHI[TID] field. If WS=1, the data portion (TLBLO) is loaded into **r**D.	**r**D,**r**A,WS

Table 6-5: **TLB-Management Instructions**

Mnemonic	Name	Operation	Operand Syntax
tlbsx	TLB Search Indexed	If a translation is found, **r**D is loaded with the *index* of the UTLB entry for the page specified by EA. If a translation is not found, **r**D is undefined. The index is used by the **tlbre** and **tlbre** instructions. EA is calculated using register-indirect with index addressing: EA = (**r**A | 0) + (**r**B)	**r**D,**r**A,**r**B
tlbsx.	TLB Search Indexed and Record	If a translation is found, **r**D is loaded with the *index* of the UTLB entry for the page specified by EA, and CR0[EQ] is set to 1. If a translation is not found, **r**D 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 = (**r**A | 0) + (**r**B)	
tlbsync	TLB Synchronize	On the PPC405, this instruction performs no operation.	—
tlbwe	TLB Write Entry	**r**A contains an index value ranging from 0 to 63. Part of the UTLB entry specified by the index in **r**A is loaded with the value in **r**S. If WS=0, the tag portion (TLBHI) is loaded from **r**S *and* the TLBHI[TID] field is updated with the PID. If WS=1, the data portion (TLBLO) is loaded from **r**S.	**r**S,**r**A,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 832, 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.

XILINX®

Chapter 7

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 a 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 353).
- The accessed memory location must be protected by the guarded storage attribute (see **Guarded (G)**, page 452).

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 502, 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.

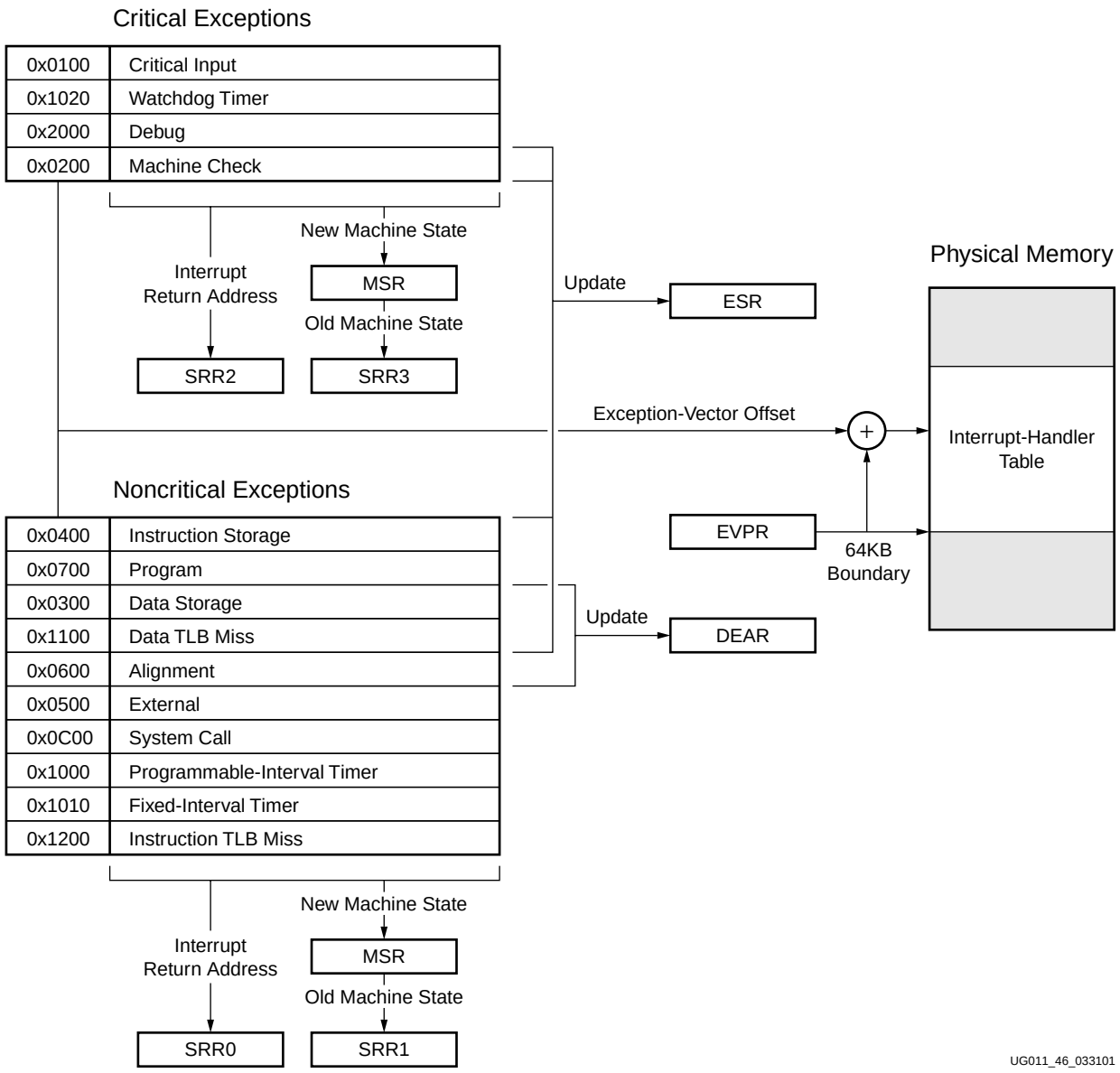


Figure 7-1: **PPC405 Exception Mechanism**

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 502 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.
   - **rfci** 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 **rfi**.
   - Instructions are fetched from the address in SRR2 following completion of the **rfci**.

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-though 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

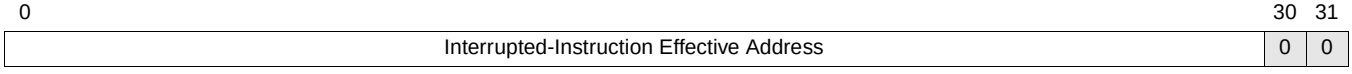
During an interrupt, the contents of the MSR (see page 431) are loaded into either SRR1 (noncritical interrupts) or SRR3 (critical interrupts). Depending on the interrupt, the MSR is updated with the values shown in Table 7-3.

Table 7-3: **Effect of Interrupts on Machine-State Register Contents**

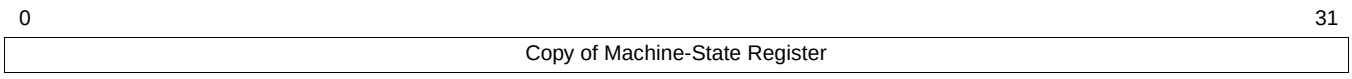
Bit	Name	Interrupt	Value	Description
0:5		All	0	Reserved
6	AP	All	0	This unsupported bit is cleared, but otherwise ignored.
7:11		All	0	Reserved
12	APE	All	0	This unsupported bit is cleared, but otherwise ignored.
13	WE	All	0	Processor wait state is disabled.
14	CE	Critical-Input Interrupt Machine-Check 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.



SRR0



SRR1

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).

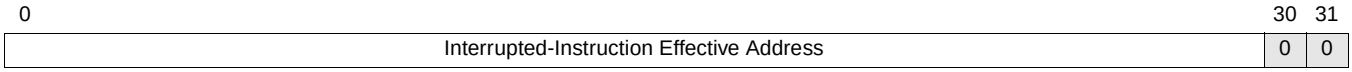
SRR0 is a privileged SPR with an address of 26 (0x01A) and SRR1 is a privileged SPR with an address of 27 (0x01B). Both registers are read and written using the **mfspr** and **mtspr** instructions.

Save/Restore Registers 2 and 3

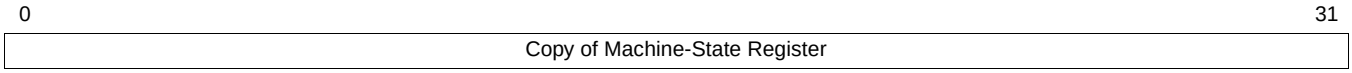
The save/restore registers 2 and 3 (SRR2 and SRR3) are 32-bit registers used to save machine state when a critical interrupt occurs. Interrupts defined as critical are:

- Critical-Input Interrupt.
- Machine-Check Interrupt.
- Watchdog-Timer Interrupt.
- Debug Interrupt.

The format of each register is shown in Figure 7-3.



SRR2



SRR3

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 **mfspr** and **mtspr** 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.

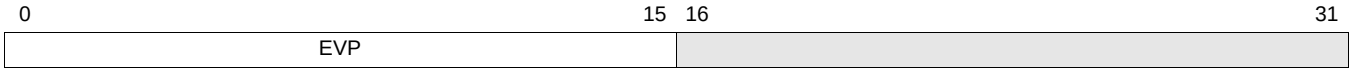


Figure 7-4: **Exception-Vector Prefix Register (EVPR)**

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 **mfspr** and **mtspr** 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.

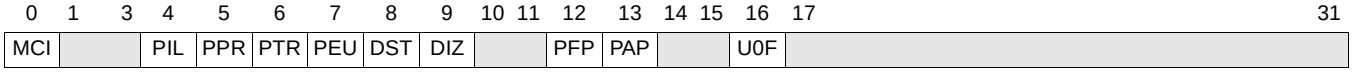


Figure 7-5: **Exception-Syndrome Register (ESR)**

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 **mfspr** and **mtspr** 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.

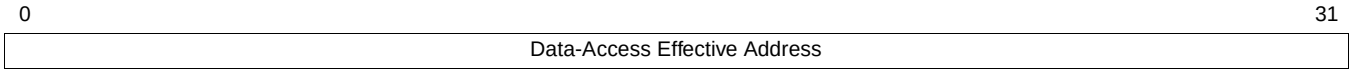


Figure 7-6: **Data Exception-Address Register (DEAR)**

The DEAR is a privileged SPR with an address of 981 (0x3D5) and is read and written using the **mfspr** and **mtspr** 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 **rfci** 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] ← 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] ← 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] ← 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] ← 1 for attempted execution of an illegal instruction, otherwise 0. This bit is set if software attempts to execute a floating-point instruction. [PPR] ← 1 for attempted execution of a privileged instruction in user mode, otherwise 0. [PTR] ← 1 for exceptions due to trap instructions, otherwise 0. [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.

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 **rfci** 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] ← 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 479, 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] ← 1 if the operation is a store, **dcbi**, **dcbz**, or **dccci**, 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 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 479, 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 **rfci** 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 **rfci** 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 *following* the instruction that caused the debug exception.
	EDE	Loaded with the 32-bit exception-vector physical address of the *exception* 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 541, for more information on the DBSR.

Register	Value After Interrupt
DBSR	Updated to reflect the debug event.

Chapter 8

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.

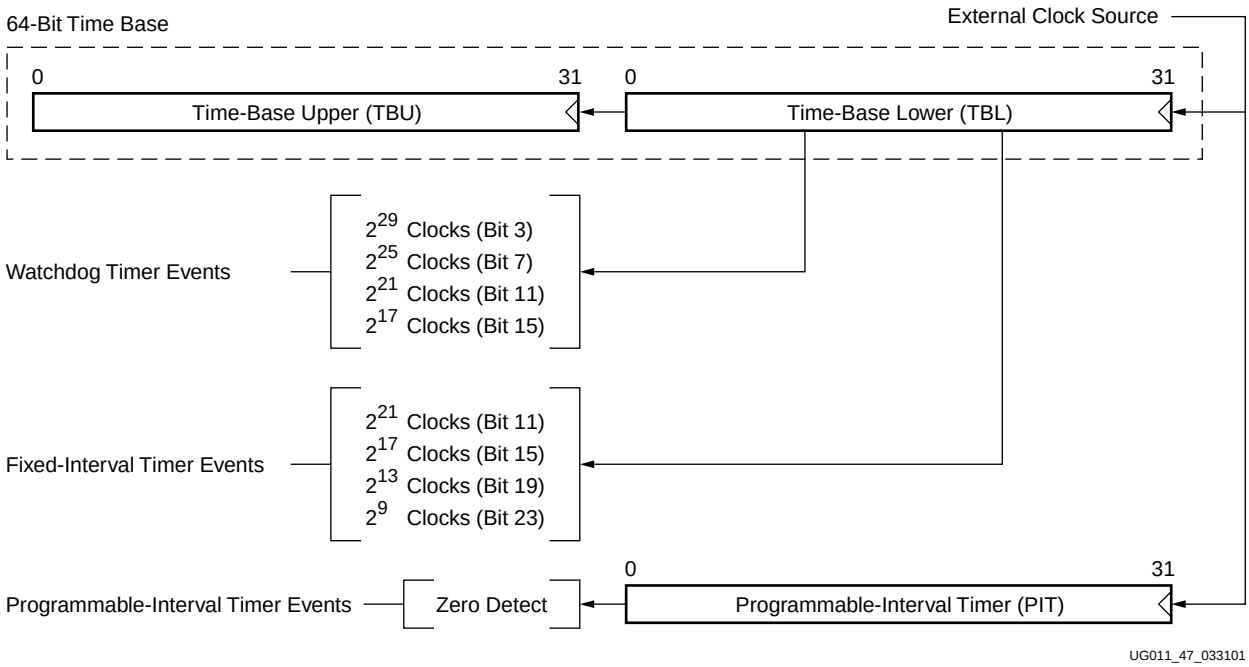


Figure 8-1: **PPC405 Timer Resources**

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.

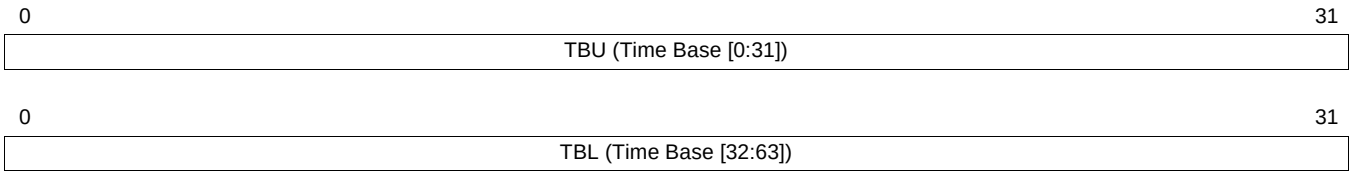


Figure 8-2: **Time-Base Register**

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 571).

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. **r**D is loaded with the contents of the time-base register specified by TBRN.	**r**D,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 **r**S.	SPRN,**r**S

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 830, 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.

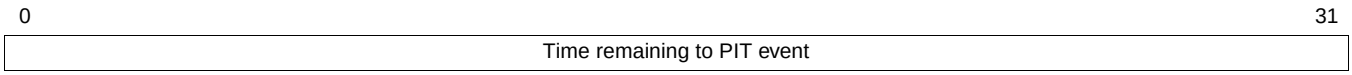


Figure 8-3: **Programmable-Interval Timer Register (PIT)**

The PIT is a privileged SPR with an address of 987 (0x3DB). It is read and written using the **mfspr** and **mtspr** 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 532. 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 **mtspr** 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.

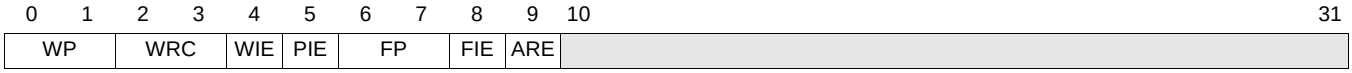


Figure 8-4: **Timer-Control Register (TCR)**

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.

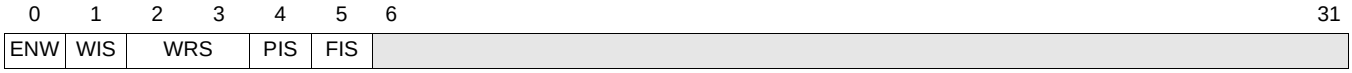


Figure 8-5: **Timer-Status Register (TSR)**

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 **mfspr** instruction. The register is cleared, but not directly written, using the **mtspr** instruction. Values in the source register, **r**S, behave as a mask when clearing the TSR. Here, a value of 0b1 in any bit position of **r**S *clears* the corresponding bit in the TSR. A value of 0b0 in an **r**S 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 530, 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 502, 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.

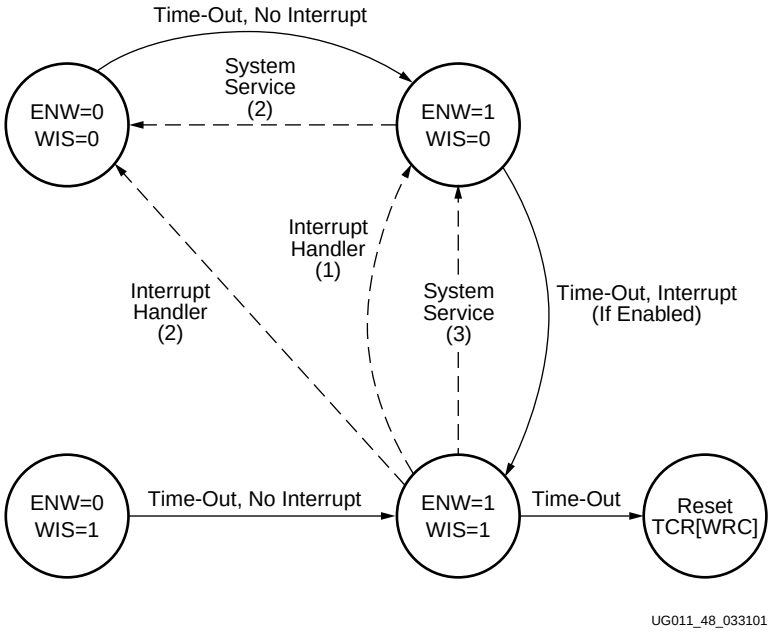


Figure 8-6: **Watchdog-Event State Machine**

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 µsec
01	19	2^{13}	27.3 µsec
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.

XILINX®

Chapter 9

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.

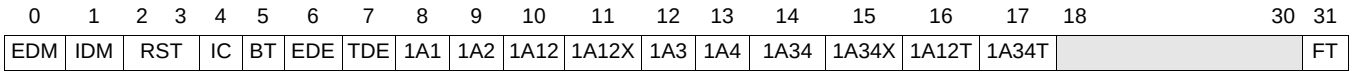


Figure 9-1: **Debug-Control Register 0 (DBCR0)**

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.

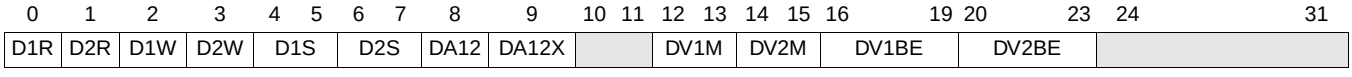


Figure 9-2: **Debug-Control Register 1 (DBCR1)**

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 *disabled* 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 *disabled* 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 **mfspr** and **mtspr** 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.

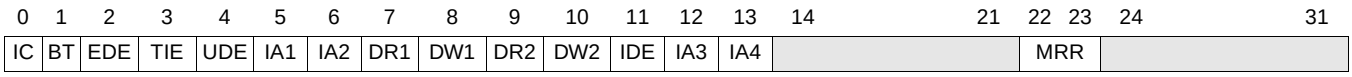


Figure 9-3: **Debug-Status Register (DBSR)**

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 **mfspr** instruction. The register is cleared, but not directly written, using the **mtspr** instruction. Values in the source register, **r**S, behave as a mask when clearing the DBSR. Here, a value of 0b1 in any bit position of **r**S *clears* the corresponding bit in the DBSR. A value of 0b0 in an **r**S 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).

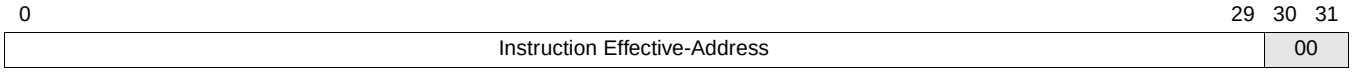


Figure 9-4: **Instruction Address-Compare Registers (IAC1–IAC4)**

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 **mfspr** and **mtspr** 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.

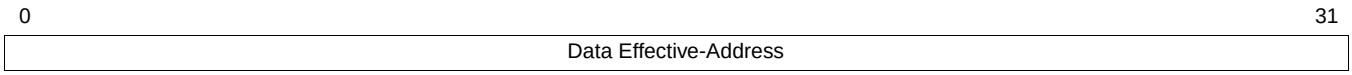


Figure 9-5: **Data Address-Compare Registers (DAC1, DAC2)**

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 **mfspr** and **mtspr** 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 9-5 shows the format of the DVC*n* registers. Any data value can be loaded in these registers.

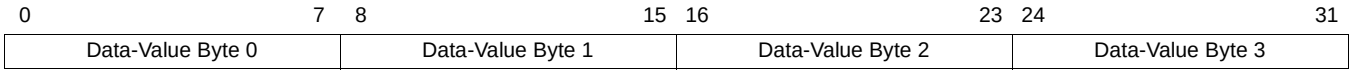


Figure 9-6: **Data Value-Compare Registers (DVC1, DVC2)**

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 **mfspr** and **mtspr** 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 556 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	DBCR0	DBCR1	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 DBCR0[IC]=1 and disabled by clearing DBCR0[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 IA*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.

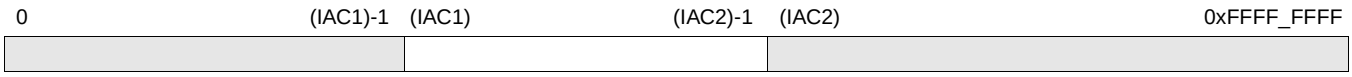
  If DBCR0[IA12X]=0, instruction addresses from (IAC1) to (IAC2)-1 fall within the range. Addresses from 0 to (IAC1)-1 and (IAC2) to 0xFFFF_FFFF fall outside the range.

  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.

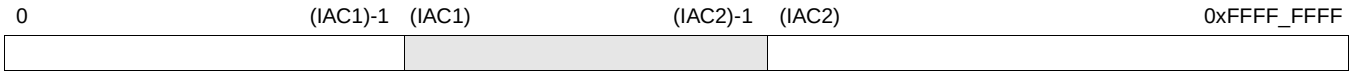
If DBCR0[IA12X]=1, instruction addresses from 0 to (IAC1)-1 and (IAC2) to 0xFFFF_FFFF fall within the range. Addresses from (IAC1) to (IAC2)-1 fall outside the range.

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.



Inclusive Range, DBCR0[IA12X]=0



Exclusive Range, DBCR0[IA12X]=1

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 0.
- 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 DAC*n* 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 DACn 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

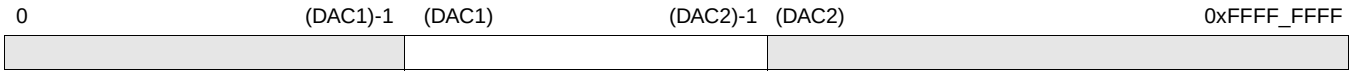
a read-access match is detected, the corresponding status bits in the DBSR are set (DR1 and/or DR2). Likewise, write-access for the entire range is checked by setting the D1W and/or D2W bits in the DBCR1 register. If a write-access match is detected, the corresponding status bits in the DBSR are set (DW1 and/or DW2).

Inclusive and Exclusive Ranges

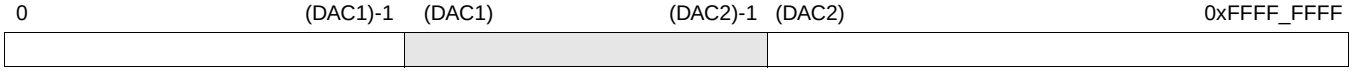
The DBCR1[DA12X] bit determines whether the address range specified by the DAC*n* registers is inclusive or exclusive:

- 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.



Inclusive Range, DBCR1[DA12X]=0



Exclusive Range, DBCR1[DA12X]=1

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 (≠ 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 543, the byte-enables specify which DVC*n* register bytes participate in the DVC comparison:

- DV*n*BE$_0$ controls participation of DVC*n* data-value byte 0.
- DV*n*BE$_1$ controls participation of DVC*n* data-value byte 1.
- DV*n*BE$_2$ controls participation of DVC*n* data-value byte 2.
- DV*n*BE$_3$ controls participation of DVC*n* data-value byte 3.

When a DV*n*BE 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*n*BE=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*n*BE must match the corresponding operand bytes.
- 10—OR mode. At least one of the DVC*n* bytes selected by DV*n*BE 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*n*R and D*n*W 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 occurance 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.

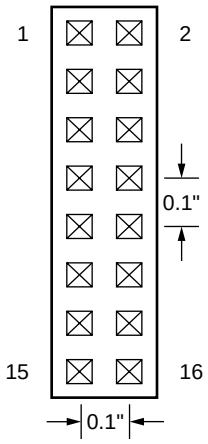


Figure 9-9: **JTAG-Connector Physical Layout**

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 10KΩ 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 (1KΩ or less) should be used to provide short-circuit current-limiting protection.
3. A 10KΩ 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.

XILINX®

Chapter 10

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:
1. 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.
2. 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.

1. 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 */
/* ------------------------------- */

@0xFFFFFFFC:      /* Initial instruction fetch from
0xFFFF_FFFC. */
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(SU0R, 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.

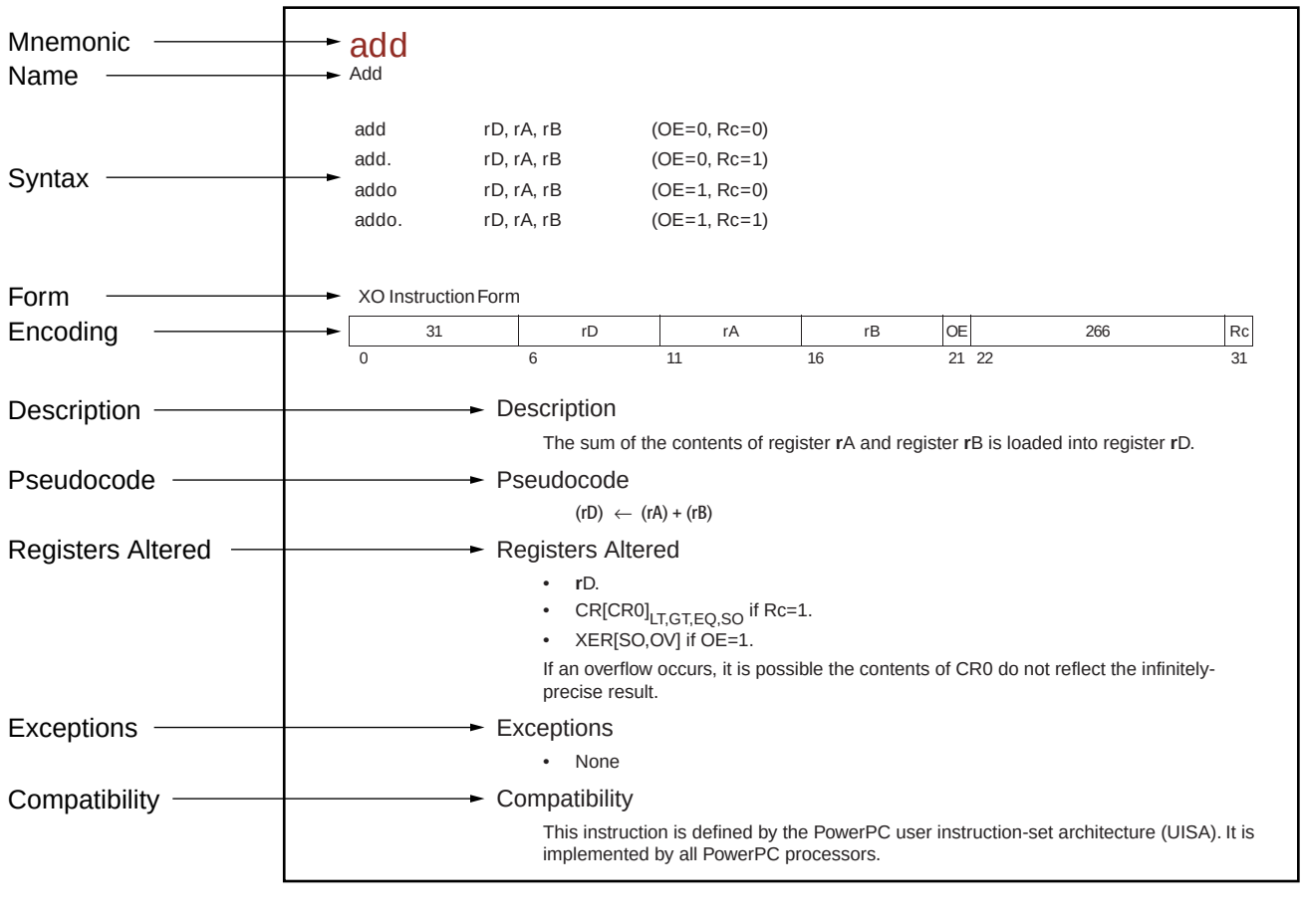


Figure 11-1: **Instruction Description Format**

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 792 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 315. The precedence of pseudocode operations is further described in the Preface in **Operator Precedence**, page 317.

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 **mfspr** and **mtspr** instructions, SPRN is the assembler operand and SPRF is the split field. SPRF corresponds to SPRN as follows:
  - $SPRF_{0:4}$ is equivalent to $SPRN_{5:9}$.
  - $SPRF_{5:9}$ is equivalent to $SPRN_{0:4}$.
- In the **mfdcr** and **mtdcr** instructions, DCRN is the assembler operand and DCRF is the split field. DCRF corresponds to DCRN as follows:
  - $DCRF_{0:4}$ is equivalent to $DCRN_{5:9}$.
  - $DCRF_{5:9}$ is equivalent to $DCRN_{0:4}$.
- In the **mftb** instruction, TBRN is the assembler operand and TBRF is the split field. TBRF corresponds to TBRN as follows:
  - $TBRF_{0:4}$ is equivalent to $TBRN_{5:9}$.
  - $TBRF_{5:9}$ is equivalent to $TBRN_{0:4}$.

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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
add.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
addo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
addo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	266	Rc
0	6	11	16	21	22	31

Description

The sum of the contents of register **r**A and register **r**B is loaded into register **r**D.

Pseudocode

(**r**D) ← (**r**A) + (**r**B)

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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 (UISA). It is implemented by all PowerPC processors.

addc

Add Carrying

addc	**r**D, **r**A, **r**B	(OE=0, Rc=0)
addc.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
addco	**r**D, **r**A, **r**B	(OE=1, Rc=0)
addco.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	10	Rc
0	6	11	16	21	22	31

Description

The sum of the contents of register **r**A and register **r**B is loaded into register **r**D. XER[CA] is updated to reflect the unsigned magnitude of the resulting sum.

Pseudocode

```
(rD) ← (rA) + (rB)
if    (rD) ≥u 2^32 − 1
   then   XER[CA]  ←  1
   else   XER[CA]  ←  0
```

Registers Altered

- **r**D.
- 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
adde.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
addeo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
addeo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	138	Rc
0	6	11	16	21	22	31

Description

The sum of the contents of register **r**A, register **r**B, and XER[CA] is loaded into register **r**D. 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 390.

Pseudocode

```
(rD) ← (rA) + (rB) + XER[CA]
if    (rD) ≥u 2^32 − 1
  then  XER[CA]  ←  1
  else  XER[CA]  ←  0
```

Registers Altered

- **r**D.
- 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.

addi

Add Immediate

addi **r**D, **r**A, SIMM

D Instruction Form

14	rD	rA	SIMM
0	6	11	16 31

Description

If the **r**A field is 0, the SIMM field is sign-extended to 32 bits and loaded into register **r**D. If the **r**A field is nonzero, the SIMM field is sign-extended to 32 bits and added to the contents of register **r**A. The resulting sum is loaded into register **r**D.

Simplified mnemonics defined for this instruction are described in the following sections:

- **Load Address**, page 834.
- **Load Immediate**, page 834.
- **Subtract Instructions**, page 831.

Pseudocode

(**r**D) ← (**r**A|0) + EXTS(SIMM)

Registers Altered

- **r**D.

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 **r**D, **r**A, SIMM

D Instruction Form

12	rD	rA	SIMM
0	6	11	16 31

Description

The SIMM field is sign-extended to 32 bits and added to the contents of register **r**A. The resulting sum is loaded into register **r**D. 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 831.

Pseudocode

```
(rD) ← (rA) + EXTS(SIMM)
if  (rD) ≥u 2^32 − 1
  then  XER[CA] ← 1
  else  XER[CA] ← 0
```

Registers Altered

- **r**D.
- XER[CA].

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 and Record

addic. **r**D, **r**A, SIMM

D Instruction Form

13	rD	rA	SIMM
0	6	11	16 31

Description

The SIMM field is sign-extended to 32 bits and added to the contents of register **r**A. The resulting sum is loaded into register **r**D. 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 831.

Pseudocode

```
(rD) ← (rA) + EXTS(SIMM)
if   (rD) ≥u 2^32 − 1
  then  XER[CA] ← 1
  else  XER[CA] ← 0
```

Registers Altered

- **r**D.
- XER[CA].
- $CR[CR0]_{LT, GT, EQ, SO}$.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

addis

Add Immediate Shifted

addis **r**D, **r**A, SIMM

D Instruction Form

15	rD	rA	SIMM
0	6	11	16 31

Description

If the **r**A field is 0, the SIMM field is concatenated on the right with sixteen 0-bits and the result is loaded into register **r**D. If the **r**A 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 **r**A. The resulting sum is loaded into register **r**D.

Simplified mnemonics defined for this instruction are described in the following sections:

- **Load Immediate**, page 834.
- **Subtract Instructions**, page 831.

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

(**r**D) ← (**r**A|0) + (SIMM || ^{16}0)

Registers Altered

- **r**D.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

addme

Add to Minus One Extended

addme	**r**D, **r**A	(OE=0, Rc=0)
addme.	**r**D, **r**A	(OE=0, Rc=1)
addmeo	**r**D, **r**A	(OE=1, Rc=0)
addmeo.	**r**D, **r**A	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	0 0 0 0 0	OE	234	Rc
0	6	11	16	21	22	31

Description

The sum of the contents of register **r**A, the XER[CA] bit, and the value −1 is loaded into register **r**D. 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 390.

Pseudocode

```
(rD) ← (rA) + XER[CA] + (−1)
if    (rD) ≥u 2^32 − 1
  then  XER[CA] ← 1
  else  XER[CA] ← 0
```

Registers Altered

- **r**D.
- 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 (UISA). It is implemented by all PowerPC processors.

addze

Add to Zero Extended

addze	**r**D, **r**A	(OE=0, Rc=0)
addze.	**r**D, **r**A	(OE=0, Rc=1)
addzeo	**r**D, **r**A	(OE=1, Rc=0)
addzeo.	**r**D, **r**A	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	0 0 0 0 0	OE	202	Rc
0	6	11	16	21	22	31

Description

The sum of the contents of register **r**A and XER[CA] is loaded into register **r**D. 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 390.

Pseudocode

```
(rD) ← (rA) + XER[CA]
if    (rD) ≥u 2^32 − 1
  then  XER[CA] ← 1
  else  XER[CA] ← 0
```

Registers Altered

- **r**D.
- 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 (UISA). It is implemented by all PowerPC processors.

and

AND

and	**r**A, **r**S, **r**B	(Rc=0)
and.	**r**A, **r**S, **r**B	(Rc=1)

X Instruction Form

31	rS	rA	rB	28	Rc
0	6	11	16	21	31

Description

The contents of register **r**S are ANDed with the contents of register **r**B and the result is loaded into register **r**A.

Pseudocode

(**r**A) ← (**r**S) ∧ (**r**B)

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

andc

AND with Complement

andc	**r**A, **r**S, **r**B	(Rc=0)
andc.	**r**A, **r**S, **r**B	(Rc=1)

X Instruction Form

31	rS	rA	rB	60	Rc
0	6	11	16	21	31

Description

The contents of register **r**S are ANDed with the one's complement of the contents of register **r**B and the result is loaded into register **r**A.

Pseudocode

(**r**A) ← (**r**S) ∧ ¬(**r**B)

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

andi.

AND Immediate

andi. **r**A, **r**S, UIMM

D Instruction Form

28	rS	rA	UIMM
0	6	11	16 31

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the left. The contents of register **r**S are ANDed with the extended UIMM field and the result is loaded into register **r**A.

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

$$(\mathbf{r}A) \leftarrow (\mathbf{r}S) \wedge (^{16}0 \parallel UIMM)$$

Registers Altered

- **r**A.
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

andis.

AND Immediate Shifted

andis. **r**A, **r**S, UIMM

D Instruction Form

29	rS	rA	UIMM
0	6	11	31
		16	

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the right. The contents of register **r**S are ANDed with the extended UIMM field and the result is loaded into register **r**A.

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

$(\mathbf{r}A) \leftarrow (\mathbf{r}S) \wedge (UIMM \parallel {}^{16}0)$

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$.

Exceptions

- None.

Compatibility

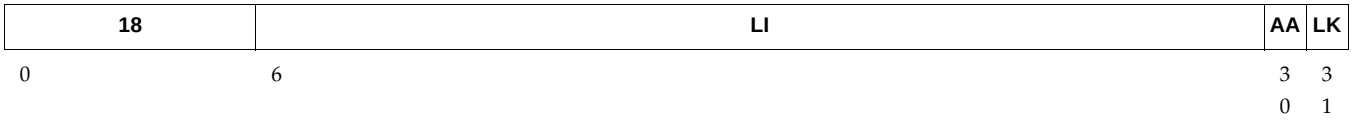
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

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



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

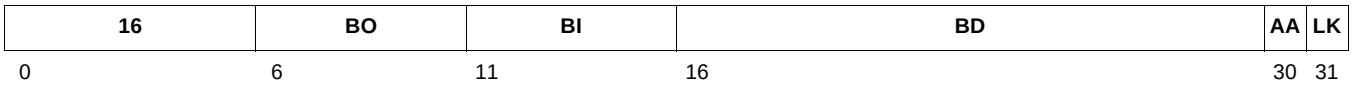
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

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



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 367. 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 BO_2=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	11	16	21	31

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 367. 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 BO_2=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.
- BO_2=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	11	16	21	31

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 367. 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 BO_2=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 **crf**D, 0, **r**A, **r**B

X Instruction Form

31	crfD	0	0	rA	rB	0	0
0	6	9	10	15	16	20	30

Description

A 32-bit signed comparison is performed between the contents of register **r**A and register **r**B. **crf**D 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 828.

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[CR*n*] as specified by the **crf**D 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 **crf**D, 0, **r**A, 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 **r**A and the sign-extended SIMM field. **crf**D 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 828.

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 **crf**D 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.

cmpl

Compare Logical

cmpl **crf**D, 0, **r**A, **r**B

X Instruction Form

31	crfD	0	0	rA	rB	32	0
0	6	9	10	15	16	20	21 31

Description

A 32-bit unsigned comparison is performed between the contents of register **r**A and register **r**B. **crf**D 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 828.

Pseudocode

$c_{0:3} \leftarrow$ 0b0000
if (**r**A) $<^u$ (**r**B) then $c_0 \leftarrow 1$
if (**r**A) $>^u$ (**r**B) then $c_1 \leftarrow 1$
if (**r**A) = (**r**B) then $c_2 \leftarrow 1$
$c_3 \leftarrow$ XER[SO]
n $\leftarrow$ **crf**D
CR[CRn] $\leftarrow c_{0:3}$

Registers Altered

- CR[CR*n*] as specified by the **crf**D 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 **crf**D, 0, **r**A, UIMM

D Instruction Form

10	crfD	0 0	rA	UIMM
0	6	9	11	16 31

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 **r**A and the zero-extended UIMM field. **crf**D 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 828.

Pseudocode

```
c0:3 ← 0b0000
if (rA) <u (160 || UIMM)then   c0 ← 1
if (rA) >u (160 || UIMM)then   c1 ← 1
if (rA) = (160 || UIMM)then    c2 ← 1
c3 ← XER[SO]
n ← crfD
CR[CRn] ← c0:3
```

Registers Altered

- CR[CR*n*] as specified by the **crf**D 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	**r**A, **r**S	(Rc=0)
cntlzw.	**r**A, **r**S	(Rc=1)

X Instruction Form

31	rS	rA	0 0 0 0 0	26	Rc
0	6	11	16	21	31

Description

The consecutive leading 0 bits in register **r**S are counted and the count is loaded into register **r**A. 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

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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 (UISA). It is implemented by all PowerPC processors.

crand

Condition Register AND

crand **crb**D, **crb**A, **crb**B

XL Instruction Form

19	crbD	crbA	crbB	257	0
0	6	11	16	21	31

Description

The CR bit specified by **crb**A is ANDed with the CR bit specified by **crb**B and the result is loaded into the CR bit specified by **crb**D.

Pseudocode

CR[**crb**D] ← CR[**crb**A] ∧ CR[**crb**B]

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 **crb**D, **crb**A, **crb**B

XL Instruction Form

19	**crb**D	**crb**A	**crb**B	129	0
0	6	11	16	21	31

Description

The CR bit specified by **crb**A is ANDed with the one's complement of the CR bit specified by **crb**B and the result is loaded into the CR bit specified by **crb**D.

Pseudocode

CR[**crb**D] ← CR[**crb**A] ∧ ¬CR[**crb**B]

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 **crb**D, **crb**A, **crb**B

XL Instruction Form

19	crbD	crbA	crbB	289	0
0	6	11	16	21	31

Description

The CR bit specified by **crb**A is XORed with the CR bit specified by **crb**B and the one's complement of the result is loaded into the CR bit specified by **crb**D.

Simplified mnemonics defined for this instruction are described in **CR-Logical Instructions**, page 828.

Pseudocode

CR[**crb**D] ← ¬(CR[**crb**A] ⊕ CR[**crb**B])

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 **crb**D, **crb**A, **crb**B

XL Instruction Form

19	crbD	crbA	crbB	225	0
0	6	11	16	21	31

Description

The CR bit specified by **crb**A is ANDed with the CR bit specified by **crb**B and the one's complement of the result is loaded into the CR bit specified by **crb**D.

Pseudocode

CR[**crb**D] ← ¬(CR[**crb**A] ∧ CR[**crb**B])

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 **crb**D, **crb**A, **crb**B

XL Instruction Form

19	crbD	crbA	crbB	33	0
0	6	11	16	21	31

Description

The CR bit specified by **crb**A is ORed with the CR bit specified by **crb**B and the one's complement of the result is loaded into the CR bit specified by **crb**D.

Simplified mnemonics defined for this instruction are described in **CR-Logical Instructions**, page 828.

Pseudocode

CR[**crb**D] ← ¬(CR[**crb**A] ∨ CR[**crb**B])

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.

cror

Condition Register OR

cror **crb**D, **crb**A, **crb**B

XL Instruction Form

19	crbD	crbA	crbB	449	0
0	6	11	16	21	31

Description

The CR bit specified by **crb**A is ORed with the CR bit specified by **crb**B and the result is loaded into the CR bit specified by **crb**D.

Simplified mnemonics defined for this instruction are described in **CR-Logical Instructions**, page 828.

Pseudocode

CR[**crb**D] ← CR[**crb**A] ∨ CR[**crb**B]

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.

crorc

Condition Register OR with Complement

crorc **crb**D, **crb**A, **crb**B

XL Instruction Form

19	crbD	crbA	crbB	417	0
0	6	11	16	21	31

Description

The CR bit specified by **crb**A is ORed with the one's complement of the CR bit specified by **crb**B and the result is loaded into the CR bit specified by **crb**D.

Pseudocode

CR[**crb**D] ← CR[**crb**A] ∨ ¬CR[**crb**B]

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 **crb**D, **crb**A, **crb**B

XL Instruction Form

19	crbD	crbA	crbB	193	0
0	6	11	16	21	31

Description

The CR bit specified by **crb**A is XORed with the CR bit specified by **crb**B and the result is loaded into the CR bit specified by **crb**D.

Simplified mnemonics defined for this instruction are described in **CR-Logical Instructions**, page 828.

Pseudocode

CR[**crb**D] ← CR[**crb**A] ⊕ CR[**crb**B]

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.

dcba

Data Cache Block Allocate

dcba **r**A, **r**B

X Instruction Form

31	0 0 0 0 0	rA	rB	758	0
0	6	1 1	1 6	2 1	3 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 **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 ← (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 **r**A, **r**B

X Instruction Form

31	0 0 0 0 0	rA	rB	86	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **r**A, **r**B

X Instruction Form

31	0 0 0 0 0	rA	rB	470	0
0	6	11	16	21	31

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 **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 ← (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 **r**A, **r**B

X Instruction Form

31	0 0 0 0 0	rA	rB	54	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **r**A, **r**B

X Instruction Form

31	0 0 0 0 0	rA	rB	278	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 ← (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 **r**A, **r**B

X Instruction Form

31	0 0 0 0 0	rA	rB	246	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 ← (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 **r**A, **r**B

X Instruction Form

31	0 0 0 0 0	rA	rB	1014	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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.

dccci

Data Cache Congruence Class Invalidate

dccci **r**A, **r**B

X Instruction Form

31	0 0 0 0 0	rA	rB	454	0
0	6	11	16	21	31

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 **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **dccci** instructions should be executed, one for each congruence class. Afterwards, cachability can be enabled.

Pseudocode

EA $\leftarrow$ (**r**A|0) + (**r**B)
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 **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	486	0
0	6	11	16	21	31

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 **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **r**D.

$EA_{19:26}$ 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, **r**D 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

- **r**D.

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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
divw.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
divwo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
divwo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	491	Rc
0	6	11	16	21	22	31

Description

The contents of register **r**A (dividend) are divided by the contents of register **r**B (divisor). The quotient is loaded into register **r**D. Both the dividend and the divisor are interpreted as signed integers. The quotient is the unique signed integer that satisfies the equation:

dividend = (quotient × divisor) + remainder

where the remainder has the same sign as the dividend, and:

- $0 \le \text{remainder} < |\text{divisor}|$, if the dividend is positive.
- $-|\text{divisor}| < \text{remainder} \le 0$, 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 × divisor
subf    rD, rD, rA      # rD = remainder
```

The contents of register **r**D are undefined if an attempt is made to perform either of the following invalid divisions:

- 0x8000 0000 ÷ −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

(**r**D) ← (**r**A) ÷ (**r**B)

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
divwu.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
divwuo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
divwuo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	459	Rc
0	6	11	16	21	22	31

Description

The contents of register **r**A (dividend) are divided by the contents of register **r**B (divisor). The quotient is loaded into register **r**D. Both the dividend and the divisor are interpreted as unsigned integers. The quotient is the unique unsigned integer that satisfies the equation:

dividend = (quotient × divisor) + remainder

where 0 ≤ remainder < 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 × 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 **r**D 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

(**r**D) ← (**r**A) ÷ (**r**B)

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

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			21	31

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	**r**A, **r**S, **r**B	(Rc=0)
eqv.	**r**A, **r**S, **r**B	(Rc=1)

X Instruction Form

31	rS	rA	rB	284	Rc
0	6	11	16	21	31

Description

The contents of register **r**S are XORed with the contents of register **r**B. A one's complement of the result is calculated and loaded in register **r**A.

Pseudocode

(**r**A) ← ¬((**r**S) ⊕ (**r**B))

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

extsb

Extend Sign Byte

extsb	**r**A, **r**S	(Rc=0)
extsb.	**r**A, **r**S	(Rc=1)

X Instruction Form

31	rS	rA	0 0 0 0 0	954	Rc
0	6	11	16	21	31

Description

The least-significant byte of register **r**S is sign-extended to 32 bits by replicating bit $\mathbf{r}S_{24}$ into bits 0 through 23 of the result. The result is loaded into register **r**A.

Pseudocode

(**r**A) ← EXTS($\mathbf{r}S_{24:31}$)

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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 (UISA). It is implemented by all PowerPC processors.

extsh

Extend Sign Halfword

extsh	**r**A, **r**S	(Rc=0)
extsh.	**r**A, **r**S	(Rc=1)

X Instruction Form

31	rS	rA	0 0 0 0 0	922	Rc
0	6	11	16	21	31

Description

The least-significant halfword of register **r**S is sign-extended to 32 bits by replicating bit $\mathbf{r}S_{16}$ into bits 0 through 15 of the result. The result is loaded into register **r**A.

Pseudocode

(**r**A) ← EXTS($\mathbf{r}S_{16:31}$)

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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 (UISA). It is implemented by all PowerPC processors.

icbi

Instruction Cache Block Invalidate

icbi **r**A, **r**B

X Instruction Form

31	0 0 0 0 0	rA	rB	982	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 ← (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 **r**A, **r**B

X Instruction Form

31	0 0 0 0 0	rA	rB	262	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 ← (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	11	16	21	31

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 **r**A, **r**B

X Instruction Form

31	0 0 0 0 0	rA	rB	998	0
0	6	11	16	21	31

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 **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **mfspr** that reads the ICDBDR. An **isync** instruction should be inserted between the **icread** and the **mfspr** 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			21	31

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 **r**D, d(**r**A)

D Instruction Form

34	rD	rA	d
0	6	11	16 31

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 **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **r**D.

Pseudocode

EA ← (**r**A|0) + EXTS(d)
(**r**D) ← $^{24}0$ || MS(EA,1)

Registers Altered

- **r**D.

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.

lbzu

Load Byte and Zero with Update

lbzu **r**D, d(**r**A)

D Instruction Form

35	rD	rA	d
0	6	11	16 31

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 **r**A 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 **r**D. The EA is loaded into **r**A.

Pseudocode

EA ← (**r**A) + EXTS(d)
(**r**D) ← $^{24}0$ || MS(EA,1)
(**r**A) ← EA

Registers Altered

- **r**A.
- **r**D.

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:

- **r**A=**r**D.
- **r**A=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 **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	119	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- The contents of register **r**A 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 **r**D. The EA is loaded into **r**A.

Pseudocode

EA ← (**r**A) + (**r**B)
(**r**D) ← $^{24}0$ || MS(EA,1)
(**r**A) ← EA

Registers Altered

- **r**A.
- **r**D.

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.
- **r**A=**r**D.
- **r**A=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

lbzx

Load Byte and Zero Indexed

lbzx **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	87	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **r**D.

Pseudocode

EA ← (**r**A|0) + (**r**B)
(**r**D) ← $^{24}0$ || MS(EA,1)

Registers Altered

- **r**D.

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 (UISA). It is implemented by all PowerPC processors.

lha

Load Halfword Algebraic

lha **r**D, d(**r**A)

D Instruction Form

42	rD	rA	d
0	6	11	16 31

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 **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The halfword referenced by EA is sign-extended to 32 bits and loaded into register **r**D.

Pseudocode

```
EA    ← (rA|0) + EXTS(d)
(rD)  ← EXTS(MS(EA,2))
```

Registers Altered

- **r**D.

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.

lhau

Load Halfword Algebraic with Update

lhau **r**D, d(**r**A)

D Instruction Form

43	rD	rA	d
0	6	11	16 31

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 **r**A are used as the base address.

The halfword referenced by EA is sign-extended to 32 bits and loaded into register **r**D. The EA is loaded into **r**A.

Pseudocode

```
EA    ← (rA) + EXTS(d)
(rD)  ← EXTS(MS(EA,2))
(rA)  ← EA
```

Registers Altered

- **r**A.
- **r**D.

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:

- **r**A=**r**D.
- **r**A=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 **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	375	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- The contents of register **r**A are used as the base address.

The halfword referenced by EA is sign-extended to 32 bits and loaded into register **r**D. The EA is loaded into **r**A.

Pseudocode

```
EA   ← (rA) + (rB)
(rD) ← EXTS(MS(EA,2))
(rA) ← EA
```

Registers Altered

- **r**A.
- **r**D.

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.
- **r**A=**r**D.
- **r**A=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

lhax

Load Halfword Algebraic Indexed

lhax **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	343	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The halfword referenced by EA is sign-extended to 32 bits and loaded into register **r**D.

Pseudocode

```
EA    ← (rA|0) + (rB)
(rD)  ← EXTS(MS(EA,2))
```

Registers Altered

- **r**D.

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 (UISA). It is implemented by all PowerPC processors.

lhbrx

Load Halfword Byte-Reverse Indexed

lhbrx **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	790	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **r**D. The byte-reversal operation consists of:

- Bits 0:7 of the memory word are loaded into **r**D[24:31].
- Bits 8:15 of the memory word are loaded into **r**D[16:23].
- 16 0-bits are loaded into **r**D[0:15].

Pseudocode

EA ← (**r**A|0) + (**r**B)
(**r**D) ← $^{16}0$ || MS(EA +1,1) || MS(EA,1)

Registers Altered

- **r**D.

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 (UISA). It is implemented by all PowerPC processors.

lhz

Load Halfword and Zero

lhz **r**D, d(**r**A)

D Instruction Form

40	rD	rA	d
0	6	11	16 31

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 **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **r**D.

Pseudocode

```
EA    ← (rA|0) + EXTS(d)
(rD)  ← 16^0 || MS(EA,2)
```

Registers Altered

- **r**D.

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.

lhzu

Load Halfword and Zero with Update

lhzu **r**D, d(**r**A)

D Instruction Form

41	rD	rA	d
0	6	11	16 31

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 **r**A 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 **r**D. The EA is loaded into **r**A.

Pseudocode

```
EA    ← (rA) + EXTS(d)
(rD)  ← 160 || MS(EA,2)
(rA)  ← EA
```

Registers Altered

- **r**A.
- **r**D.

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:

- **r**A=**r**D.
- **r**A=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 **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	311	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- The contents of register **r**A 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 **r**D. The EA is loaded into **r**A.

Pseudocode

EA ← (**r**A) + (**r**B)
(**r**D) ← $^{16}0$ || MS(EA,2)
(**r**A) ← EA

Registers Altered

- **r**A.
- **r**D.

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.
- **r**A=**r**D.
- **r**A=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

lhzx

Load Halfword and Zero Indexed

lhzx **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	279	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **r**D.

Pseudocode

EA ← (**r**A|0) + (**r**B)
(**r**D) ← $^{16}0$ || MS(EA,2)

Registers Altered

- **r**D.

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 (UISA). It is implemented by all PowerPC processors.

lmw

Load Multiple Word

lmw **r**D, d(**r**A)

D Instruction Form

46	rD	rA	d
0 5	6 10	11 15	16 31

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 **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

Let $n = 32 - \mathbf{r}D$.

n consecutive words starting at the memory address referenced by EA are loaded into GPRs **r**D through **r**31.

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

- **r**D through **r**31.

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:

- **r**A is in the range of registers to be loaded, including the case **r**A=**r**D=0. The word that would have been loaded into **r**A 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 **r**D, **r**A, NB

X Instruction Form

31	rD	rA	NB	597	0
0	6	11	16	21	31

Description

An effective address (EA) is determined by the **r**A field as follows:

- If the **r**A field is 0, the EA is 0.
- If the **r**A field is not 0, the contents of register **r**A 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* = 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 **r**D through R_{FINAL}. The sequence of registers wraps around to **r**0 if necessary. R_{FINAL} = **r**D + 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

- **r**D 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:

- **r**A is in the range of registers to be loaded, including the case **r**A=**r**D=0. Bytes that would have been loaded into **r**A are discarded.
- 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.

lswx

Load String Word Indexed

lswx **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	533	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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* = 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 **r**D through R_{FINAL}. The sequence of registers wraps around to **r**0 if necessary. R_{FINAL} = **r**D + 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 **r**D 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

- **r**D 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:

- **r**A is in the range of registers to be loaded, including the case **r**A=**r**D=0. Bytes that would have been loaded into **r**A are discarded.
- 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.

lwarx

Load Word and Reserve Indexed

lwarx **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	20	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The word referenced by EA is loaded into register **r**D. A reservation bit internal to the processor is set.

The **lwarx** and the **stwcx.** instructions should paired in a loop to create the effect of an atomic memory operation for accessing a semaphore. See **Semaphore Synchronization**, page 426 for more information.

Pseudocode

```
EA        ← (rA|0) + (rB)
(rD)      ← MS(EA,4)
RESERVE   ← 1
```

Registers Altered

- **r**D.

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 (UISA). It is implemented by all PowerPC processors.

lwbrx

Load Word Byte-Reverse Indexed

lwbrx **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	534	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The memory word referenced by EA is byte-reversed and the result is loaded into register **r**D. The byte-reversal operation consists of:

- Bits 0:7 of the memory word are loaded into **r**D[24:31].
- Bits 8:15 of the memory word are loaded into **r**D[16:23].
- Bits 16:23 of the memory word are loaded into **r**D[8:15].
- Bits 23:31 of the memory word are loaded into **r**D[0:7].

Pseudocode

```
EA   ← (rA|0) + (rB)
(rD) ← MS(EA+3,1) || MS(EA+2,1) || MS(EA+1,1) || MS(EA,1)
```

Registers Altered

- **r**D.

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 (UISA). It is implemented by all PowerPC processors.

lwz

Load Word and Zero

lwz **r**D, d(**r**A)

D Instruction Form

32	rD	rA	d
0	6	11	16 31

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 **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The word referenced by EA is loaded into register **r**D.

Pseudocode

```
EA    ← (rA|0) + EXTS(d)
(rD)  ← MS(EA,4)
```

Registers Altered

- **r**D.

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 **r**D, d(**r**A)

D Instruction Form

33	rD	rA	d
0	6	11	16 31

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 **r**A are used as the base address.

The word referenced by EA is loaded into register **r**D. The EA is loaded into **r**A.

Pseudocode

```
EA    ← (rA) + EXTS(d)
(rD)  ← MS(EA,4)
(rA)  ← EA
```

Registers Altered

- **r**A.
- **r**D.

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:

- **r**A=**r**D.
- **r**A=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 **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	55	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- The contents of register **r**A are used as the base address.

The word referenced by EA is loaded into register **r**D. The EA is loaded into **r**A.

Pseudocode

```
EA    ← (rA) + (rB)
(rD)  ← MS(EA,4)
(rA)  ← EA
```

Registers Altered

- **r**A.
- **r**D.

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.
- **r**A=**r**D.
- **r**A=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

lwzx

Load Word and Zero Indexed

lwzx **r**D, **r**A, **r**B

X Instruction Form

31	rD	rA	rB	23	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The word referenced by EA is loaded into register **r**D.

Pseudocode

EA ← (**r**A|0) + (**r**B)
(**r**D) ← MS(EA,4)

Registers Altered

- **r**D.

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 (UISA). It is implemented by all PowerPC processors.

macchw

Multiply Accumulate Cross Halfword to Word Modulo Signed

macchw	**r**D, **r**A, **r**B	(OE=0, Rc=0)
macchw.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
macchwo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
macchwo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	172	Rc
0	6	1 1	1 6	2 1	2 2	3 1

Description

The low-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The signed product is added to the contents of **r**D and the sum is stored as a 33-bit temporary result. The contents of **r**D are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in Figure 3-28, page 408.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{0:15}$ signed
$temp_{0:32} \leftarrow prod_{0:31} + (\mathbf{r}D)$
$(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
macchws.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
macchwso	**r**D, **r**A, **r**B	(OE=1, Rc=0)
macchwso.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	236	Rc
0	6	11	16	21	22	31

Description

The low-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The signed product is added to the contents of **r**D 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 **r**D. If the result overflows, **r**D is loaded with the nearest representable value. If the result is less than -2^{31}, the value stored in **r**D is -2^{31}. If the result is greater than $2^{31} - 1$, the value stored in **r**D is $2^{31} - 1$. An example of this operation is shown in Figure 3-28, page 408.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{0:15}$ signed
$temp_{0:32} \leftarrow prod_{0:31} + (\mathbf{r}D)$
if $((prod_0 = \mathbf{r}D_0) \wedge (\mathbf{r}D_0 \neq temp_1))$
  then $(\mathbf{r}D) \leftarrow (\mathbf{r}D_0 \,||\, {}^{31}(\neg \mathbf{r}D_0))$
  else $(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
macchwsu.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
macchwsuo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
macchwsuo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	204	Rc
0	6	11	16	21	22	31

Description

The low-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The unsigned product is added to the contents of **r**D 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 **r**D. If the result overflows, **r**D is loaded with the nearest representable value. If the result is greater than $2^{32} - 1$, the value stored in **r**D is $2^{32} - 1$. An example of this operation is shown in Figure 3-28, page 408.

Pseudocode

$$\begin{aligned} prod_{0:31} &\leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{0:15} \text{ unsigned} \\ temp_{0:32} &\leftarrow prod_{0:31} + (\mathbf{r}D) \\ (\mathbf{r}D) &\leftarrow (temp_{1:32} \vee {}^{32}temp_{0}) \end{aligned}$$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
macchwu.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
macchwuo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
macchwuo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	140	Rc
0	6	11	16	21	22	31

Description

The low-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The unsigned product is added to the contents of **r**D and the sum is stored as a 33-bit temporary result. The contents of **r**D are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in Figure 3-28, page 408.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{0:15}$ unsigned
$temp_{0:32} \leftarrow prod_{0:31} + (\mathbf{r}D)$
$(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
machhw.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
machhwo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
machhwo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	44	Rc
0	6	11	16	21	22	31

Description

The high-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The signed product is added to the contents of **r**D and the sum is stored as a 33-bit temporary result. The contents of **r**D are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in Figure 3-29, page 410.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{0:15} \times (\mathbf{r}B)_{0:15}$ signed
$temp_{0:32} \leftarrow prod_{0:31} + (\mathbf{r}D)$
$(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
machhws.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
machhwso	**r**D, **r**A, **r**B	(OE=1, Rc=0)
machhwso.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	108	Rc
0	6	11	16	21	22	31

Description

The high-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The signed product is added to the contents of **r**D 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 **r**D. If the result overflows, **r**D is loaded with the nearest representable value. If the result is less than -2^{31}, the value stored in **r**D is -2^{31}. If the result is greater than $2^{31} - 1$, the value stored in **r**D is $2^{31} - 1$. An example of this operation is shown in Figure 3-29, page 410.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{0:15} \times (\mathbf{r}B)_{0:15}$ signed
$temp_{0:32} \leftarrow prod_{0:31} + (\mathbf{r}D)$
if $((prod_0 = \mathbf{r}D_0) \wedge (\mathbf{r}D_0 \neq temp_1))$
  then $(\mathbf{r}D) \leftarrow (\mathbf{r}D_0 \,||\, {}^{31}(\neg \mathbf{r}D_0))$
  else $(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

machhwsu

Multiply Accumulate High Halfword to Word Saturate Unsigned

machhwsu	**r**D, **r**A, **r**B	(OE=0, Rc=0)
machhwsu.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
machhwsuo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
machhwsuo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	76	Rc
0	6	11	16	21	22	31

Description

The high-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The unsigned product is added to the contents of **r**D 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 **r**D. If the result overflows, **r**D is loaded with the nearest representable value. If the result is greater than $2^{32} - 1$, the value stored in **r**D is $2^{32} - 1$. An example of this operation is shown in Figure 3-29, page 410.

Pseudocode

$$\begin{aligned} prod_{0:31} &\leftarrow (\mathbf{r}A)_{0:15} \times (\mathbf{r}B)_{0:15} \text{ unsigned} \\ temp_{0:32} &\leftarrow prod_{0:31} + (\mathbf{r}D) \\ (\mathbf{r}D) &\leftarrow (temp_{1:32} \vee {}^{32}temp_{0}) \end{aligned}$$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
machhwu.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
machhwuo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
machhwuo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	12	Rc
0	6	11	16	21	22	31

Description

The high-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The unsigned product is added to the contents of **r**D and the sum is stored as a 33-bit temporary result. The contents of **r**D are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in Figure 3-29, page 410.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{0:15} \times (\mathbf{r}B)_{0:15}$ unsigned
$temp_{0:32} \leftarrow prod_{0:31} + (\mathbf{r}D)$
$(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

maclhw

Multiply Accumulate Low Halfword to Word Modulo Signed

maclhw	**r**D, **r**A, **r**B	(OE=0, Rc=0)
maclhw.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
maclhwo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
maclhwo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	428	Rc
0	6	11	16	21	22	31

Description

The low-order halfword of **r**A is multiplied by the low-order halfword of **r**B. The signed product is added to the contents of **r**D and the sum is stored as a 33-bit temporary result. The contents of **r**D are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in Figure 3-30, page 413.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{16:31}$ signed
$temp_{0:32} \leftarrow prod_{0:31} + (\mathbf{r}D)$
$(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
maclhws.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
maclhwso	**r**D, **r**A, **r**B	(OE=1, Rc=0)
maclhwso.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	492	Rc
0	6	11	16	21	22	31

Description

The low-order halfword of **r**A is multiplied by the low-order halfword of **r**B. The signed product is added to the contents of **r**D 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 **r**D. If the result overflows, **r**D is loaded with the nearest representable value. If the result is less than -2^{31}, the value stored in **r**D is -2^{31}. If the result is greater than $2^{31} - 1$, the value stored in **r**D is $2^{31} - 1$. An example of this operation is shown in Figure 3-30, page 413.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{16:31}$ signed
$temp_{0:32} \leftarrow prod_{0:31} + (\mathbf{r}D)$
if $((prod_0 = \mathbf{r}D_0) \wedge (\mathbf{r}D_0 \neq temp_1))$
  then $(\mathbf{r}D) \leftarrow (\mathbf{r}D_0 \,||\, {}^{31}(\neg \mathbf{r}D_0))$
  else $(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
maclhwsu.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
maclhwsuo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
maclhwsuo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	460	Rc
0	6	11	16	21	22	31

Description

The low-order halfword of **r**A is multiplied by the low-order halfword of **r**B. The unsigned product is added to the contents of **r**D 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 **r**D. If the result overflows, **r**D is loaded with the nearest representable value. If the result is greater than $2^{32} - 1$, the value stored in **r**D is $2^{32} - 1$. An example of this operation is shown in Figure 3-30, page 413.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{16:31}$ unsigned
$temp_{0:32} \leftarrow prod_{0:31} + (\mathbf{r}D)$
$(\mathbf{r}D) \leftarrow (temp_{1:32} \vee {}^{32}temp_{0})$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

maclhwu

Multiply Accumulate Low Halfword to Word Modulo Unsigned

maclhwu	**r**D, **r**A, **r**B	(OE=0, Rc=0)
maclhwu.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
maclhwuo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
maclhwuo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	396	Rc
0	6	11	16	21	22	31

Description

The low-order halfword of **r**A is multiplied by the low-order halfword of **r**B. The unsigned product is added to the contents of **r**D and the sum is stored as a 33-bit temporary result. The contents of **r**D are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in Figure 3-30, page 413.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{16:31}$ unsigned
$temp_{0:32} \leftarrow prod_{0:31} + (\mathbf{r}D)$
$(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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 **crf**D, **crf**S

XL Instruction Form

19	crfD	0 0	crfS	0 0 0 0 0 0 0	0	0
0	6	9	11	14	21	31

Description

The contents of the CR field specified by **crf**S are loaded into the CR field specified by **crf**D.

Pseudocode

```
m      ← crfS
n      ← crfD
(CR[CRn]) ← (CR[CRm])
```

Registers Altered

- CR[CR*n*] where *n* is specified by **crf**D.

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 **crf**D

X Instruction Form

31	crfD	0 0 0 0 0 0 0 0 0 0 0	512	0
0	6	9	21	31

Description

The contents of $XER_{0:3}$ are loaded into the CR field specified by **crf**D. The contents of $XER_{0:3}$ are then cleared to 0.

Pseudocode

```
n           ← crfD
(CR[CRn])   ← XER0:3
XER0:3      ← 0b0000
```

Registers Altered

- CR[CR*n*] where *n* is specified by the **crf**D 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 (UISA). It is implemented by all PowerPC processors.

mfcr

Move from Condition Register

mfcr **r**D

X Instruction Form

31	rD	0 0 0 0 0 0 0 0 0 0	19	0
0	6	1 1	2 1	3 1

Description

The contents of the CR are loaded into register **r**D.

Pseudocode

(**r**D) ← (CR)

Registers Altered

- **r**D.

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.

mfdcr

Move from Device Control Register

mfdcr **r**D, DCRN

XFX Instruction Form

31	rD	DCRF	323	0
0	6	11	21	31

Description

This is a privileged instruction.

The contents of the DCR specified by the DCR number (DCRN) are loaded into register **r**D. The DCRF opcode field is a split field representing DCRN. See **Split-Field Notation**, page 571 for more information.

Pseudocode

$$DCRN \leftarrow DCRF_{5:9} \,||\, DCRF_{0:4}$$
$$(\mathbf{r}D) \leftarrow (DCR(DCRN))$$

Registers Altered

- **r**D.

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 **r**D

X Instruction Form

31	rD	0 0 0 0 0 0 0 0 0 0	83	0
0	6	1 1	2 1	3 1

Description

This is a privileged instruction.

The contents of the MSR are loaded into register **r**D.

Pseudocode

(**r**D) ← (MSR)

Registers Altered

- **r**D.

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.

mfspr

Move from Special Purpose Register

mfspr **r**D, SPRN

XFX Instruction Form

31	rD	SPRF	339	0
0	6	11	21	31

Description

The contents of the SPR specified by the SPR number (SPRN) are loaded into register **r**D. The SPRF opcode field is a split field representing SPRN. See **Split-Field Notation**, page 571 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 830.

Pseudocode

$$SPRN \leftarrow SPRF_{5:9} \parallel SPRF_{0:4}$$
$$(\mathbf{r}D) \leftarrow (SPR(SPRN))$$

Registers Altered

- **r**D.

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 **r**D, TBRN

XFX Instruction Form

31	rD	TBRF	371	0
0	6	11	21	31

Description

The contents of the TBR specified by the TBR number (TBRN) are loaded into register **r**D. The TBRF opcode field is a split field representing TBRN. See **Split-Field Notation**, page 571 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 830.

Pseudocode

$$TBRN \leftarrow TBRF_{5:9} \parallel TBRF_{0:4}$$
$$(\mathbf{r}D) \leftarrow (TBR(TBRN))$$

Registers Altered

- **r**D.

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, **r**S

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 **r**S 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 **r**S and CR registers. If a CRM bit is set to 1, the specified set of bits in **r**S are copied into the corresponding CR bits. If a CRM bit is cleared to 0, the specified set of bits in **r**S are not copied and the corresponding CR bits are unchanged. The following table shows the relationship between the CRM field and the **r**S 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 423, 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 834.

Pseudocode

$$\text{mask} \leftarrow {}^4(\text{CRM}_0) \parallel {}^4(\text{CRM}_1) \parallel \ldots \parallel {}^4(\text{CRM}_6) \parallel {}^4(\text{CRM}_7)$$
$$(\text{CR}) \leftarrow ((\mathbf{r}\text{S}) \wedge \text{mask}) \vee ((\text{CR}) \wedge \neg\text{mask})$$

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, **r**S

XFX Instruction Form

31	rS	DCRF	451	0
0	6	11	21	31

Description

This is a privileged instruction.

The contents of register **r**S 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 571 for more information.

Pseudocode

$$DCRN \leftarrow DCRF_{5:9} \,||\, DCRF_{0:4}$$
$$(DCR(DCRN)) \leftarrow (\mathbf{r}S)$$

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 **r**S

X Instruction Form

31	rS	0 0 0 0 0 0 0 0 0 0	146	0
0	6	1 1	2 1	3 1

Description

This is a privileged instruction.

The contents of register **r**S are loaded into the MSR.

Pseudocode

(MSR) ← (**r**S)

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, **r**S

XFX Instruction Form

31	rS	SPRF	467	0
0	6	1 1	2 1	3 1

Description

The contents of register **r**S 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 571 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 830.

Pseudocode

$$\begin{aligned} \text{SPRN} &\leftarrow \text{SPRF}_{5:9} \parallel \text{SPRF}_{0:4} \\ \text{(SPR(SPRN))} &\leftarrow \text{(rS)} \end{aligned}$$

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	**r**D, **r**A, **r**B	(Rc=0)
mulchw.	**r**D, **r**A, **r**B	(Rc=1)

X Instruction Form

4	rD	rA	rB	168	Rc
0	6	11	16	21	31

Description

The low-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The resulting signed 32-bit product is loaded into register **r**D. An example of this operation is shown in Figure 3-34, page 420.

Pseudocode

$(\mathbf{r}D)_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{0:15}$ signed

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(Rc=0)
mulchwu.	**r**D, **r**A, **r**B	(Rc=1)

X Instruction Form

4	rD	rA	rB	136	Rc
0	6	11	16	21	31

Description

The low-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The resulting unsigned 32-bit product is loaded into register **r**D. An example of this operation is shown in Figure 3-34, page 420.

Pseudocode

$(\mathbf{r}D)_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{0:15}$ unsigned

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

mulhhw

Multiply High Halfword to Word Signed

mulhhw	**r**D, **r**A, **r**B	(Rc=0)
mulhhw.	**r**D, **r**A, **r**B	(Rc=1)

X Instruction Form

4	rD	rA	rB	40	Rc
0	6	11	16	21	31

Description

The high-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The resulting signed 32-bit product is loaded into register **r**D. An example of this operation is shown in Figure 3-35, page 421.

Pseudocode

$(\mathbf{r}D)_{0:31} \leftarrow (\mathbf{r}A)_{0:15} \times (\mathbf{r}B)_{0:15}$ signed

Registers Altered

- **r**D.
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ 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	**r**D, **r**A, **r**B	(Rc=0)
mulhhwu.	**r**D, **r**A, **r**B	(Rc=1)

X Instruction Form

4	rD	rA	rB	8	Rc
0	6	11	16	21	31

Description

The high-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The resulting unsigned 32-bit product is loaded into register **r**D. An example of this operation is shown in Figure 3-35, page 421.

Pseudocode

$(\mathbf{r}D)_{0:31} \leftarrow (\mathbf{r}A)_{0:15} \times (\mathbf{r}B)_{0:15}$ unsigned

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(Rc=0)
mulhw.	**r**D, **r**A, **r**B	(Rc=1)

XO Instruction Form

31	rD	rA	rB	0	75	Rc
0	6	11	16	21	22	31

Description

The contents of register **r**A are multiplied with the contents of register **r**B, forming a 64-bit signed product. The most-significant 32 bits of the result are loaded into register **r**D.

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

$prod_{0:63} \leftarrow$ (**r**A) $\times$ (**r**B) signed
(**r**D) $\leftarrow$ $prod_{0:31}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(Rc=0)
mulhwu.	**r**D, **r**A, **r**B	(Rc=1)

XO Instruction Form

31	rD	rA	rB	0	11	Rc
0	6	11	16	21		31

Description

The contents of register **r**A are multiplied with the contents of register **r**B, forming a 64-bit unsigned product. The most-significant 32 bits of the result are loaded into register **r**D.

mulhw should be used if the operands are to be interpreted as signed quantities.

Pseudocode

$prod_{0:63} \leftarrow$ (**r**A) $\times$ (**r**B) unsigned
(**r**D) $\leftarrow$ $prod_{0:31}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

mullhw

Multiply Low Halfword to Word Signed

mullhw	**r**D, **r**A, **r**B	(Rc=0)
mullhw.	**r**D, **r**A, **r**B	(Rc=1)

X Instruction Form

4	rD	rA	rB	424	Rc
0	6	11	16	21	31

Description

The low-order halfword of **r**A is multiplied by the low-order halfword of **r**B. The resulting signed 32-bit product is loaded into register **r**D. An example of this operation is shown in Figure 3-36, page 422.

Pseudocode

$$(\mathbf{r}D)_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{16:31} \text{ signed}$$

Registers Altered

- **r**D.
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc=1.

Exceptions

- None.

Compatibility

This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

mullhwu

Multiply Low Halfword to Word Unsigned

mullhwu	**r**D, **r**A, **r**B	(OE=0, Rc=0)
mullhwu.	**r**D, **r**A, **r**B	(OE=0, Rc=1)

X Instruction Form

4	rD	rA	rB	392	Rc
0	6	11	16	21	31

Description

The low-order halfword of **r**A is multiplied by the low-order halfword of **r**B. The resulting unsigned 32-bit product is loaded into register **r**D. An example of this operation is shown in Figure 3-36, page 422.

Pseudocode

$(\mathbf{r}D)_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{16:31}$ unsigned

Registers Altered

- **r**D.
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ 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 **r**D, **r**A, SIMM

D Instruction Form

7	rD	rA	SIMM
0	6	1 1	1 6 3 1

Description

The contents of register **r**A 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 **r**D.

The result loaded into register **r**D 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

$prod_{0:47} \leftarrow$ (**r**A) $\times$ EXTS(SIMM) signed
(**r**D) $\leftarrow$ $prod_{16:47}$

Registers Altered

- **r**D.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

mullw

Multiply Low Word

mullw	**r**D, **r**A, **r**B	(OE=0, Rc=0)
mullw.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
mullwo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
mullwo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	235	Rc
0	6	1 1	1 6	2 1	2 2	3 1

Description

The contents of register **r**A are multiplied with the contents of register **r**B, forming a 64-bit signed product. The least-significant 32 bits of the result are loaded into register **r**D.

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 **r**D 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

$prod_{0:63} \leftarrow$ (**r**A) $\times$ (**r**B) signed
(**r**D) $\leftarrow$ $prod_{32:63}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

nand

NAND

nand	**r**A, **r**S, **r**B	(Rc=0)
nand.	**r**A, **r**S, **r**B	(Rc=1)

X Instruction Form

31	rS	rA	rB	476	Rc
0	6	11	16	21	31

Description

The contents of register **r**S are ANDed with the contents of register **r**B and the one's complement of the result is loaded into register **r**A.

The one's complement of a number can be obtained using **nand** with **r**S = **r**B.

Pseudocode

(**r**A) $\leftarrow \neg((\mathbf{r}S) \wedge (\mathbf{r}B))$

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

neg

Negate

neg	**r**D, **r**A	(OE=0, Rc=0)
neg.	**r**D, **r**A	(OE=0, Rc=1)
nego	**r**D, **r**A	(OE=1, Rc=0)
nego.	**r**D, **r**A	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	0 0 0 0 0	OE	104	Rc
0	6	11	16	21	22	31

Description

The two's complement of the contents of register **r**A are loaded into register **r**D.

If **r**A 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

(**r**D) ← ¬(**r**A) + 1

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
nmacchw.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
nmacchwo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
nmacchwo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	174	Rc
0	6	1 1	1 6	2 1	2 2	3 1

Description

The low-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The negated signed product is added to the contents of **r**D and the sum is stored as a 33-bit temporary result. The contents of **r**D are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in Figure 3-31, page 415.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{0:15}$ signed
$nprod_{0:31} \leftarrow -1 \times prod_{0:31}$ signed
$temp_{0:32} \leftarrow nprod_{0:31} + (\mathbf{r}D)$
$(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
nmacchws.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
nmacchwso	**r**D, **r**A, **r**B	(OE=1, Rc=0)
nmacchwso.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	238	Rc
0	6	1 1	1 6	2 1	2 2	3 1

Description

The low-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The negated signed product is added to the contents of **r**D 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 **r**D. If the result overflows, **r**D is loaded with the nearest representable value. If the result is less than -2^{31}, the value stored in **r**D is -2^{31}. If the result is greater than $2^{31}-1$, the value stored in **r**D is $2^{31}-1$. An example of this operation is shown in Figure 3-31, page 415.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{0:15}$ signed
$nprod_{0:31} \leftarrow -1 \times prod_{0:31}$ signed
$temp_{0:32} \leftarrow nprod_{0:31} + (\mathbf{r}D)$
if $((nprod_0 = \mathbf{r}D_0) \wedge (\mathbf{r}D_0 \neq temp_1))$
  then $(\mathbf{r}D) \leftarrow (\mathbf{r}D_0 \,||\, {}^{31}(\neg \mathbf{r}D_0))$
  else $(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
nmachhw.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
nmachhwo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
nmachhwo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	46	Rc
0	6	1 1	1 6	2 1	2 2	3 1

Description

The high-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The negated signed product is added to the contents of **r**D and the sum is stored as a 33-bit temporary result. The contents of **r**D are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in Figure 3-32, page 417.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{0:15} \times (\mathbf{r}B)_{0:15}$ signed
$nprod_{0:31} \leftarrow -1 \times prod_{0:31}$ signed
$temp_{0:32} \leftarrow nprod_{0:31} + (\mathbf{r}D)$
$(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
nmachhws.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
nmachhwso	**r**D, **r**A, **r**B	(OE=1, Rc=0)
nmachhwso.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	110	Rc
0	6	11	16	21	22	31

Description

The high-order halfword of **r**A is multiplied by the high-order halfword of **r**B. The negated signed product is added to the contents of **r**D 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 **r**D. If the result overflows, **r**D is loaded with the nearest representable value. If the result is less than -2^{31}, the value stored in **r**D is -2^{31}. If the result is greater than $2^{31} - 1$, the value stored in **r**D is $2^{31} - 1$. An example of this operation is shown in Figure 3-32, page 417.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{0:15} \times (\mathbf{r}B)_{0:15}$ signed
$nprod_{0:31} \leftarrow -1 \times prod_{0:31}$ signed
$temp_{0:32} \leftarrow nprod_{0:31} + (\mathbf{r}D)$
if $((nprod_0 = \mathbf{r}D_0) \wedge (\mathbf{r}D_0 \neq temp_1))$
  then $(\mathbf{r}D) \leftarrow (\mathbf{r}D_0 \,||\, {}^{31}(\neg \mathbf{r}D_0))$
  else $(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

nmaclhw

Negative Multiply Accumulate Low Halfword to Word Modulo Signed

nmaclhw	**r**D, **r**A, **r**B	(OE=0, Rc=0)
nmaclhw.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
nmaclhwo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
nmaclhwo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	430	Rc
0	6	11	16	21	22	31

Description

The low-order halfword of **r**A is multiplied by the low-order halfword of **r**B. The negated signed product is added to the contents of **r**D and the sum is stored as a 33-bit temporary result. The contents of **r**D are replaced by the low-order 32 bits of the temporary result. An example of this operation is shown in Figure 3-33, page 419.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{16:31}$ signed
$nprod_{0:31} \leftarrow -1 \times prod_{0:31}$ signed
$temp_{0:32} \leftarrow nprod_{0:31} + (\mathbf{r}D)$
$(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**D, **r**A, **r**B	(OE=0, Rc=0)
nmaclhws.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
nmaclhwso	**r**D, **r**A, **r**B	(OE=1, Rc=0)
nmaclhwso.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

4	rD	rA	rB	OE	494	Rc
0	6	11	16	21	22	31

Description

The low-order halfword of **r**A is multiplied by the low-order halfword of **r**B. The negated signed product is added to the contents of **r**D 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 **r**D. If the result overflows, **r**D is loaded with the nearest representable value. If the result is less than -2^{31}, the value stored in **r**D is -2^{31}. If the result is greater than $2^{31} - 1$, the value stored in **r**D is $2^{31} - 1$. An example of this operation is shown in Figure 3-33, page 419.

Pseudocode

$prod_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{16:31}$ signed
$nprod_{0:31} \leftarrow -1 \times prod_{0:31}$ signed
$temp_{0:32} \leftarrow nprod_{0:31} + (\mathbf{r}D)$
if $((nprod_0 = \mathbf{r}D_0) \wedge (\mathbf{r}D_0 \neq temp_1))$
  then $(\mathbf{r}D) \leftarrow (\mathbf{r}D_0 \,||\, {}^{31}(\neg \mathbf{r}D_0))$
  else $(\mathbf{r}D) \leftarrow temp_{1:32}$

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**A, **r**S, **r**B	(Rc=0)
nor.	**r**A, **r**S, **r**B	(Rc=1)

X Instruction Form

31	rS	rA	rB	124	Rc
0	6	1 1	1 6	2 1	3 1

Description

The contents of register **r**S are ORed with the contents of register **r**B and the one's complement of the result is loaded into register **r**A.

The one's complement of a number can be obtained using **nor** with **r**S = **r**B.

Simplified mnemonics defined for this instruction are described in **Other Simplified Mnemonics**, page 834.

Pseudocode

(**r**A) $\leftarrow \neg((\mathbf{r}S) \vee (\mathbf{r}B))$

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

or

OR

or	**r**A, **r**S, **r**B	(Rc=0)
or.	**r**A, **r**S, **r**B	(Rc=1)

X Instruction Form

31	rS	rA	rB	444	Rc
0	6	11	16	21	31

Description

The contents of register **r**S are ORed with the contents of register **r**B and the result is loaded into register **r**A.

The contents of one register can be copied into another register using **or** with **r**S = **r**B.

Simplified mnemonics defined for this instruction are described in **Other Simplified Mnemonics**, page 834.

Pseudocode

(**r**A) ← (**r**S) ∨ (**r**B)

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

orc

OR with Complement

orc	**r**A, **r**S, **r**B	(Rc=0)
orc.	**r**A, **r**S, **r**B	(Rc=1)

X Instruction Form

31	rS	rA	rB	412	Rc
0	6	11	16	21	31

Description

The contents of register **r**S are ORed with the one's complement of the contents of register **r**B and the result is loaded into register **r**A.

Pseudocode

(**r**A) ← (**r**S) ∨ ¬(**r**B)

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

ori

OR Immediate

ori **r**A, **r**S, UIMM

D Instruction Form

24	rS	rA	UIMM
0	6	11	16 31

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the left. The contents of the register **r**S are ORed with the extended UIMM field and the result is loaded into register **r**A.

Simplified mnemonics defined for this instruction are described in **Other Simplified Mnemonics**, page 834. The preferred no-operation (an instruction that does nothing) is:

```
ori 0,0,0
```

Pseudocode

(**r**A) ← (**r**S) ∨ (^{16}0 || UIMM)

Registers Altered

- **r**A.

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 **r**A, **r**S, UIMM

D Instruction Form

25	rS	rA	UIMM
0	6	11	16 31

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the right. The contents of the register **r**S are ORed with the extended UIMM field and the result is loaded into register **r**A.

Pseudocode

(**r**A) ← (**r**S) ∨ (UIMM || $^{16}0$)

Registers Altered

- **r**A.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). 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	51	0
0	6	21	31

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	50	0
0	6	21	31

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	**r**A, **r**S, SH, MB, ME (Rc=0)
rlwimi.	**r**A, **r**S, SH, MB, ME (Rc=1)

M Instruction Form

20	rS	rA	SH	MB	ME	Rc
0	6	11	16	21	26	31

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 400.

The contents of register **r**S are rotated left by the number of bit positions specified by the SH field. The rotated data is inserted into register **r**A 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 **r**A. If a mask bit contains a 0, the corresponding bit in **r**A 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 829.

Pseudocode

```
m      ← MASK(MB, ME)
r      ← ROTL((rS), SH)
(rA)   ← (r ∧ m) ∨ ((rA) ∧ ¬m)
```

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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	**r**A, **r**S, SH, MB, ME (Rc=0)
rlwinm.	**r**A, **r**S, SH, MB, ME (Rc=1)

M Instruction Form

21	rS	rA	SH	MB	ME	Rc
0	6	11	16	21	26	31

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 400.

The contents of register **r**S 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 **r**A.

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 829.

Pseudocode

```
m       ← MASK(MB, ME)
r       ← ROTL((rS), SH)
(rA)    ← r ∧ m
```

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

rlwnm

Rotate Left Word then AND with Mask

rlwnm	**r**A, **r**S, **r**B, MB, ME	(Rc=0)
rlwnm.	**r**A, **r**S, **r**B, MB, ME	(Rc=1)

M Instruction Form

23	rS	rA	rB	MB	ME	Rc
0	6	11	16	21	26	31

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 400.

The contents of register **r**S are rotated left by the number of bit positions specified by the contents of register $\mathbf{r}B_{27:31}$. The rotated data is ANDed with the mask and the result is loaded into register **r**A.

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 829.

Pseudocode

```
m       ← MASK(MB, ME)
r       ← ROTL((rS), (rB)27:31)
(rA)    ← r ∧ m
```

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

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 0 0	1	0
0	6	30	31

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	**r**A, **r**S, **r**B	(Rc=0)
slw.	**r**A, **r**S, **r**B	(Rc=1)

X Instruction Form

31	rS	rA	rB	24	Rc
0	6	11	16	21	31

Description

The contents of register **r**S are shifted left by the number of bits specified by the contents of register **r**$B_{27:31}$. 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 **r**A.

If **r**B_{26} = 1, register **r**A 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

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

sraw

Shift Right Algebraic Word

sraw	**r**A, **r**S, **r**B	(Rc=0)
sraw.	**r**A, **r**S, **r**B	(Rc=1)

X Instruction Form

31	rS	rA	rB	792	Rc
0	6	11	16	21	31

Description

The contents of register **r**S are shifted right by the number of bits specified by the contents of register $\mathbf{r}B_{27:31}$. Bits shifted right out of the least-significant bit are lost. The most-significant bit of register **r**S ($\mathbf{r}S_0$) is replicated to fill vacated bit positions on the left. The result is loaded into register **r**A.

If **r**S 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 $\mathbf{r}B_{26} = 1$, XER[CA] and all bits in register **r**A are set to the value of $\mathbf{r}S_0$.

Pseudocode

```
n      ← (rB)27:31
r      ← ROTL((rS), 32 − n)
if (rB)26 = 0
  then m ← MASK(n, 31)
  else m ← 32 0
s      ← (rS)0
(rA)   ← (r ∧ m) ∨ (32 s ∧ ¬m)
XER[CA] ← s ∧ ((r ∧ ¬m) ≠ 0)
```

Registers Altered

- **r**A.
- XER[CA].
- $\text{CR[CR0]}_{\text{LT, GT, EQ, SO}}$ 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.

srawi

Shift Right Algebraic Word Immediate

srawi	**r**A, **r**S, SH	(Rc=0)
srawi.	**r**A, **r**S, SH	(Rc=1)

X Instruction Form

31	rS	rA	SH	824	Rc
0	6	11	16	21	31

Description

The contents of register **r**S 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 **r**S (**r**S_0) is replicated to fill vacated bit positions on the left. The result is loaded into register **r**A.

If **r**S 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

- **r**A.
- 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 (UISA). It is implemented by all PowerPC processors.

srw

Shift Right Word

srw	**r**A, **r**S, **r**B	(Rc=0)
srw.	**r**A, **r**S, **r**B	(Rc=1)

X Instruction Form

31	rS	rA	rB	536	Rc
0	6	11	16	21	31

Description

The contents of register **r**S are shifted right by the number of bits specified by the contents of register $\mathbf{r}B_{27:31}$. 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 **r**A.

If $\mathbf{r}B_{26} = 1$, register **r**A 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 ← 32 0
(rA)   ← r ∧ m
```

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

stb

Store Byte

stb **r**S, d(**r**A)

D Instruction Form

38	rS	rA	d
0	6	11	16 31

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 **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The least-significant byte of register **r**S is stored into the byte referenced by EA.

Pseudocode

EA ← (**r**A|0) + EXTS(d)
MS(EA, 1) ← $(\mathbf{r}S)_{24:31}$

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 (UISA). It is implemented by all PowerPC processors.

stbu

Store Byte with Update

stbu **r**S, d(**r**A)

D Instruction Form

39	rS	rA	d
0	6	11	16 31

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 **r**A are used as the base address.

The least-significant byte of register **r**S is stored into the byte referenced by EA. The EA is loaded into **r**A.

Pseudocode

EA ← (**r**A) + EXTS(d)
MS(EA, 1) ← $(\mathbf{r}S)_{24:31}$
(**r**A) ← EA

Registers Altered

- **r**A.

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:

- **r**A=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 **r**S, **r**A, **r**B

X Instruction Form

31	rS	rA	rB	247	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- The contents of register **r**A are used as the base address.

The least-significant byte of register **r**S is stored into the byte referenced by EA. The EA is loaded into **r**A.

Pseudocode

EA ← (**r**A) + (**r**B)
MS(EA, 1) ← $(\mathbf{r}S)_{24:31}$
(**r**A) ← EA

Registers Altered

- **r**A.

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.
- **r**A=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

stbx

Store Byte Indexed

stbx **r**S, **r**A, **r**B

X Instruction Form

31	rS	rA	rB	215	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The least-significant byte of register **r**S is stored into the byte referenced by EA.

Pseudocode

EA ← (**r**A|0) + (**r**B)
MS(EA, 1) ← $(\mathbf{r}S)_{24:31}$

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 (UISA). It is implemented by all PowerPC processors.

sth

Store Halfword

sth **r**S, d(**r**A)

D Instruction Form

44	rS	rA	d
0	6	11	16 31

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 **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The least-significant halfword of register **r**S is stored into the halfword referenced by EA.

Pseudocode

EA ← (**r**A|0) + EXTS(d)
MS(EA, 2) ← $(\mathbf{r}S)_{16:31}$

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 (UISA). It is implemented by all PowerPC processors.

sthbrx

Store Halfword Byte-Reverse Indexed

sthbrx **r**S, **r**A, **r**B

X Instruction Form

31	rS	rA	rB	918	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The least-significant halfword of register **r**S is byte-reversed and stored into the halfword referenced by EA as follows:

- **r**S[24:31] are stored into the byte referenced by EA.
- **r**S[16:23] are stored into the byte referenced by EA+1.

Pseudocode

EA $\leftarrow$ (**r**A|0) + (**r**B)
MS(EA, 2) $\leftarrow$ $(\mathbf{r}S)_{24:31} \parallel (\mathbf{r}S)_{16:23}$

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.

sthu

Store Halfword with Update

sthu **r**S, d(**r**A)

D Instruction Form

45	rS	rA	d
0	6	11	16 31

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 **r**A are used as the base address.

The least-significant halfword of register **r**S is stored into the halfword referenced by EA. The EA is loaded into **r**A.

Pseudocode

EA ← (**r**A) + EXTS(d)
MS(EA, 2) ← $(\mathbf{r}S)_{16:31}$
(**r**A) ← EA

Registers Altered

- **r**A.

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:

- **r**A=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

sthux

Store Halfword with Update Indexed

sthux **r**S, **r**A, **r**B

X Instruction Form

31	rS	rA	rB	439	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- The contents of register **r**A are used as the base address.

The least-significant halfword of register **r**S is stored into the halfword referenced by EA. The EA is loaded into **r**A.

Pseudocode

EA ← (**r**A) + (**r**B)
MS(EA, 2) ← $(\mathbf{r}S)_{16:31}$
(**r**A) ← EA

Registers Altered

- **r**A.

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.
- **r**A=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

sthx

Store Halfword Indexed

sthx **r**S, **r**A, **r**B

X Instruction Form

31	rS	rA	rB	407	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The least-significant halfword of register **r**S is stored into the halfword referenced by EA.

Pseudocode

EA ← (**r**A|0) + (**r**B)
MS(EA, 2) ← $(\mathbf{r}S)_{16:31}$

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 (UISA). It is implemented by all PowerPC processors.

stmw

Store Multiple Word

stmw **r**S, d(**r**A)

D Instruction Form

47	rS	rA	d
0	6	11	16 31

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 **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

Let $n = 32 - rS$.

GPRs **r**S through **r**31 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 (UISA). It is implemented by all PowerPC processors.

stswi

Store String Word Immediate

stswi **r**S, **r**A, NB

X Instruction Form

31	rS	rA	NB	725	0
0	6	11	16	21	31

Description

An effective address (EA) is determined by the **r**A field as follows:

- If the **r**A field is 0, the EA is 0.
- If the **r**A field is not 0, the contents of register **r**A 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* = CEIL(*n*÷4).

GPRs **r**S through **r**S + *nr* – 1 are stored into *n* consecutive bytes starting at the memory address referenced by EA. The sequence of registers wraps around to **r**0 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 **r**S, **r**A, **r**B

X Instruction Form

31	rS	rA	rB	661	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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* = CEIL(*n*÷4).

GPRs **r**S through **r**S + *nr* – 1 are stored into *n* consecutive bytes starting at the memory address referenced by EA. The sequence of registers wraps around to **r**0 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 (UISA). It is implemented by all PowerPC processors.

stw

Store Word

stw **r**S, d(**r**A)

D Instruction Form

36	rS	rA	d
0	6	11	16 31

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 **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The contents of register **r**S are stored into the word referenced by EA.

Pseudocode

```
EA          ← (rA|0) + EXTS(d)
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.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

stwbrx

Store Word Byte-Reverse Indexed

stwbrx **r**S, **r**A, **r**B

X Instruction Form

31	rS	rA	rB	662	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The least-significant halfword of register **r**S is byte-reversed and stored into the halfword referenced by EA as follows:

- **r**S[24:31] are stored into the byte referenced by EA.
- **r**S[16:23] are stored into the byte referenced by EA+1.
- **r**S[8:15] are stored into the byte referenced by EA+2.
- **r**S[0:7] are stored into the byte referenced by EA+3.

Pseudocode

$$EA \leftarrow (\mathbf{r}A|0) + (\mathbf{r}B)$$
$$MS(EA, 4) \leftarrow (\mathbf{r}S)_{24:31} \parallel (\mathbf{r}S)_{16:23} \parallel (\mathbf{r}S)_{8:15} \parallel (\mathbf{r}S)_{0:7}$$

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. **r**S, **r**A, **r**B

X Instruction Form

31	rS	rA	rB	150	1
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **r**S 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 paired in a loop to create the effect of an atomic memory operation when accessing a semaphore. See **Semaphore Synchronization**, page 426 for more information.

Pseudocode

```
EA ← (rA|0) + (rB)
if RESERVE = 1
  then
    MS(EA, 4)← (rS)
    RESERVE← 0
    (CR[CR0])← 0b00 || 1 || XER_SO
  else
    (CR[CR0])← 0b00 || 0 || XER_SO
```

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 **r**S, d(**r**A)

D Instruction Form

37	rS	rA	d
0	6	11	16 31

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 **r**A are used as the base address.

The contents of register **r**S are stored into the word referenced by EA. The EA is loaded into **r**A.

Pseudocode

```
EA          ← (rA) + EXTS(d)
MS(EA, 4)   ← (rS)
(rA)        ← EA
```

Registers Altered

- **r**A.

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:

- **r**A=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

stwux

Store Word with Update Indexed

stwux **r**S, **r**A, **r**B

X Instruction Form

31	rS	rA	rB	183	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- The contents of register **r**A are used as the base address.

The contents of register **r**S are stored into the word referenced by EA. The EA is loaded into **r**A.

Pseudocode

```
EA          ← (rA) + (rB)
MS(EA, 4)   ← (rS)
(rA)        ← EA
```

Registers Altered

- **r**A.

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.
- **r**A=0.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

stwx

Store Word Indexed

stwx **r**S, **r**A, **r**B

X Instruction Form

31	rS	rA	rB	151	0
0	6	11	16	21	31

Description

An effective address (EA) is calculated by adding an index to a base address, which are formed as follows:

- The contents of register **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A are used as the base address.

The contents of register **r**S are stored into the word referenced by EA.

Pseudocode

EA ← (**r**A|0) + (**r**B)
MS(EA,4) ← (**r**S)

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.

subf

Subtract From

subf	**r**D, **r**A, **r**B	(OE=0, Rc=0)
subf.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
subfo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
subfo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	40	Rc
0	6	1 1	1 6	2 1	2 2	3 1

Description

The contents of register **r**A are subtracted from the contents of register **r**B, producing a two's-complement result that is loaded into register **r**D. The subtraction operation is equivalent to adding the contents of register **r**B to the one's complement of register **r**A and adding 1 to the result.

Simplified mnemonics defined for this instruction are described in **Subtract Instructions**, page 831.

Pseudocode

(**r**D) ← ¬(**r**A) + (**r**B) + 1

Registers Altered

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

subfc

Subtract from Carrying

subfc	**r**D, **r**A, **r**B	(OE=0, Rc=0)
subfc.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
subfco	**r**D, **r**A, **r**B	(OE=1, Rc=0)
subfco.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	8	Rc
0	6	11	16	21	22	31

Description

The contents of register **r**A are subtracted from the contents of register **r**B, producing a two's-complement result that is loaded into register **r**D. The subtraction operation is equivalent to adding the contents of register **r**B to the one's complement of register **r**A 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 831.

Pseudocode

```
(rD)  ←  ¬(rA) + (rB) + 1
if    (rD) ≯u 2^32 − 1
   then  XER[CA]  ←  1
   else  XER[CA]  ←  0
```

Registers Altered

- **r**D.
- 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.

subfe

Subtract from Extended

subfe	**r**D, **r**A, **r**B	(OE=0, Rc=0)
subfe.	**r**D, **r**A, **r**B	(OE=0, Rc=1)
subfeo	**r**D, **r**A, **r**B	(OE=1, Rc=0)
subfeo.	**r**D, **r**A, **r**B	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	rB	OE	136	Rc
0	6	1 1	1 6	2 1	2 2	3 1

Description

The contents of register **r**B are added to the one's complement of register **r**A. The contents of XER[CA] are added to the result. The result is loaded into register **r**D.

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 392.

Pseudocode

```
(rD)  ←  ¬(rA) + (rB) + XER[CA]
if    (rD) ≯u 2^32 − 1
  then  XER[CA]  ←  1
  else  XER[CA]  ←  0
```

Registers Altered

- **r**D.
- 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.

subfic

Subtract from Immediate Carrying

subfic **r**D, **r**A, SIMM

D Instruction Form

8	rD	rA	SIMM
0	6	1 1	1 6	3 1

Description

The contents of register **r**A are subtracted from the sign-extended SIMM field, producing a two's-complement result that is loaded into register **r**D. 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 **r**A 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) ≤u 2^32 − 1
  then  XER[CA] ← 1
  else  XER[CA] ← 0
```

Registers Altered

- **r**D.
- XER[CA].

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

subfme

Subtract from Minus One Extended

subfme	**r**D, **r**A	(OE=0, Rc=0)
subfme.	**r**D, **r**A	(OE=0, Rc=1)
subfmeo	**r**D, **r**A	(OE=1, Rc=0)
subfmeo.	**r**D, **r**A	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	0 0 0 0 0	OE	232	Rc
0	6	11	16	21	22	31

Description

The value -1 is added to the one's complement of register **r**A. The contents of XER[CA] are added to the result. The result is loaded into register **r**D.

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 392.

Pseudocode

```
(rD)  ←  ¬(rA) + 0xFFFF_FFFF + XER[CA]
if    (rD) ≯u 2^32 − 1
   then  XER[CA]  ←  1
   else  XER[CA]  ←  0
```

Registers Altered

- **r**D.
- 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	**r**D, **r**A	(OE=0, Rc=0)
subfze.	**r**D, **r**A	(OE=0, Rc=1)
subfzeo	**r**D, **r**A	(OE=1, Rc=0)
subfzeo.	**r**D, **r**A	(OE=1, Rc=1)

XO Instruction Form

31	rD	rA	0 0 0 0 0	OE	200	Rc
0	6	11	16	21	22	31

Description

The one's complement of register **r**A is added to XER[CA] and the result is loaded into register **r**D.

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 392.

Pseudocode

```
(rD)  ←  ¬(rA) + XER[CA]
if    (rD) ≯ᵘ 2^32 − 1
   then  XER[CA]  ←  1
   else  XER[CA]  ←  0
```

Registers Altered

- **r**D.
- 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.

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			21	31

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 (UISA). 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			21	31

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 **r**D, **r**A, WS

X Instruction Form

31	rD	rA	WS	946	0
0	6	11	16	21	31

Description

This is a privileged instruction.

This instruction reads an entry from the TLB. **r**$A_{26:31}$ 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 **r**D. If WS=0, the tag portion (TLBHI) is loaded into **r**D and the PID is updated with the TLBHI[TID] field. If WS=1, the data portion (TLBLO) is loaded into **r**D and the PID is not modified.

See **TLB Entries**, page 476 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 832.

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

- **r**D.
- 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	**r**D, **r**A, **r**B	(Rc=0)
tlbsx.	**r**D, **r**A, **r**B	(Rc=1)

X Instruction Form

31	rD	rA	rB	914	Rc
0	6	11	16	21	31

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 **r**B are used as the index.
- If the **r**A field is 0, the base address is 0.
- If the **r**A field is not 0, the contents of register **r**A 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 **r**D.

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

- **r**D.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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			21	31

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 **r**S, **r**A, WS

X Instruction Form

31	rS	rA	WS	978	0
0	6	11	16	21	31

Description

This is a privileged instruction.

This instruction writes a new entry into the TLB. **r**$A_{26:31}$ 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 **r**S. If WS=0, the tag portion (TLBHI) is written from **r**S and the PID field ($PID_{24:31}$) is written into the TLBHI[TID] field. If WS=1, the data portion (TLBLO) is written from **r**S.

See **TLB Entries**, page 476 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 832.

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, **r**A, **r**B

X Instruction Form

31	TO	rA	rB	4	0
0	6	1 1	1 6	2 1	3 1

Description

The TO opcode field specifies the test conditions to be performed on the contents of registers **r**A and **r**B. See Table 3-13, page 377 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 511.
- **Debug Interrupt (0x2000)**, page 521.
- **Trap-Instruction Debug Event**, page 546.
- **Internal-Debug Mode**, page 536.
- **External-Debug Mode**, page 536.

Simplified mnemonics defined for this instruction are described in **Trap Instructions**, page 832.

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, **r**A, SIMM

D Instruction Form

3	TO	rA	SIMM
0	6	11	16 31

Description

The TO opcode field specifies the test conditions to be performed on the contents of register **r**A and the sign-extended SIMM field (sign-extended to 32 bits). See Table 3-13, page 377 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 511.
- **Debug Interrupt (0x2000)**, page 521.
- **Trap-Instruction Debug Event**, page 546.
- **Internal-Debug Mode**, page 536.
- **External-Debug Mode**, page 536.

Simplified mnemonics defined for this instruction are described in **Trap Instructions**, page 832.

Pseudocode

```
if ((rA) <  EXTS(SIMM))∧ (TO0 = 1) then trap
if ((rA) >  EXTS(SIMM))∧ (TO1 = 1) then trap
if ((rA) =  EXTS(SIMM))∧ (TO2 = 1) then trap
if ((rA) <u EXTS(SIMM))∧ (TO3 = 1) then trap
if ((rA) >u EXTS(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 **r**S

X Instruction Form

31	rS	0 0 0 0 0 0 0 0 0 0	131	0
0	6	1 1	2 1	3 1

Description

This is a privileged instruction.

MSR[EE] is set to the value specified by bit 16 in register **r**S.

Pseudocode

MSR[EE] $\leftarrow$ (**r**S)$_{16}$

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	E	0 0 0 0	163	0
0	6	16	17	21	31

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	**r**A, **r**S, **r**B	(Rc=0)
xor.	**r**A, **r**S, **r**B	(Rc=1)

X Instruction Form

31	rS	rA	rB	316	Rc
0	6	11	16	21	31

Description

The contents of register **r**S are XORed with the contents of register **r**B and the result is loaded into register **r**A.

Pseudocode

(**r**A) ← (**r**S) ⊕ (**r**B)

Registers Altered

- **r**A.
- $CR[CR0]_{LT, GT, EQ, SO}$ 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.

xori

XOR Immediate

xori **r**A, **r**S, UIMM

D Instruction Form

26	rS	rA	UIMM
0	6	11	16 31

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the left. The contents of register **r**S are XORed with the extended UIMM field and the result is loaded into register **r**A.

Pseudocode

(**r**A) ← (**r**S) ⊕ (^{16}0 || UIMM)

Registers Altered

- **r**A.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

xoris

XOR Immediate Shifted

xoris **r**A, **r**S, UIMM

D Instruction Form

27	rS	rA	UIMM
0	6	11	16
			31

Description

The UIMM field is extended to 32 bits by concatenating 16 0-bits on the right. The contents of register **r**S are XORed with the extended UIMM field and the result is loaded into register **r**A.

Pseudocode

(**r**A) ← (**r**S) ⊕ (UIMM || $^{16}0$)

Registers Altered

- **r**A.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

XILINX®

Appendix A

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 360
CR	Condition Register	**Condition Register (CR)**, page 361
MSR	Machine-State Register	**Machine-State Register**, page 431
CCR0	Core-Configuration Register 0	**Core-Configuration Register**, page 459
CTR	Count Register	**Count Register (CTR)**, page 364
DAC1	Data Address-Compare 1	**Data Address-Compare Registers**, page 543
DAC2	Data Address-Compare 2	
DBCR0	Debug-Control Register 0	**Debug-Control Registers**, page 538
DBCR1	Debug-Control Register 1	
DBSR	Debug-Status Register	**Debug-Status Register**, page 541
DCCR	Data-Cache Cacheability Register	**Data-Cache Cacheability Register (DCCR)**, page 454
DCWR	Data-Cache Write-Through Register	**Data-Cache Write-Through Register (DCWR)**, page 453
DEAR	Data-Error Address Register	**Data Exception-Address Register**, page 502
DVC1	Data Value-Compare 1	**Data Value-Compare Registers**, page 543
DVC2	Data Value-Compare 2	
ESR	Exception-Syndrome Register	**Exception-Syndrome Register**, page 500
EVPR	Exception-Vector Prefix Register	**Exception-Vector Prefix Register**, page 500

Table A-1: **PPC405 Register Cross-Reference** *(Continued)*

Name	Descriptive Name	Cross Reference
IAC1	Instruction Address-Compare 1	**Instruction Address-Compare Registers**, page 542
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 454
ICDBDR	Instruction-Cache Debug-Data Register	**icread Instruction**, page 468
LR	Link Register	**Link Register (LR)**, page 363
PID	Process ID Register	**Process-ID Register**, page 474
PIT	Programmable-Interval Timer	**Programmable-Interval Timer Register**, page 527
PVR	Processor-Version Register	**Processor-Version Register**, page 433
SGR	Storage Guarded Register	**Storage Guarded Register (SGR)**, page 455
SLER	Storage Little-Endian Register	**Storage Little-Endian Register (SLER)**, page 455
SPRG0	SPR General-Purpose Register 0	**SPR General-Purpose Registers**, page 432
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 498
SRR1	Save/Restore Register 1	
SRR2	Save/Restore Register 2	**Save/Restore Registers 2 and 3**, page 499
SRR3	Save/Restore Register 3	
SU0R	Storage User-Defined 0 Register	**Storage User-Defined 0 Register (SU0R)**, page 455
TBL	Time-Base Lower	**Time Base**, page 524
TBU	Time-Base Upper	
TCR	Timer-Control Register	**Timer-Control Register**, page 528
TSR	Timer-Status Register	**Timer-Status Register**, page 529
USPRG0	User SPR General-Purpose Register 0	**User-SPR General-Purpose Register**, page 364
XER	Fixed-Point Exception Register	**Fixed-Point Exception Register (XER)**, page 363
ZPR	Zone-Protection Register	**Zone Protection**, page 482

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 772 lists the special-purpose registers sorted by SPRN and Table A-6, page 773 lists the special-purpose registers sorted by SPRF.

The following notes apply to the "Reset Value" column in these tables:

Notes:

1. 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.
2. 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
SU0R	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 770 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**	**crb**D		**crb**A			**crb**B			**257**				**0**
crandc	**19**	**crb**D		**crb**A			**crb**B			**129**				**0**
creqv	**19**	**crb**D		**crb**A			**crb**B			**289**				**0**
crnand	**19**	**crb**D		**crb**A			**crb**B			**225**				**0**
crnor	**19**	**crb**D		**crb**A			**crb**B			**33**				**0**
cror	**19**	**crb**D		**crb**A			**crb**B			**449**				**0**
crorc	**19**	**crb**D		**crb**A			**crb**B			**417**				**0**
crxor	**19**	**crb**D		**crb**A			**crb**B			**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
lhz	40	rD		rA			d							
lhzu	41	rD		rA			d							
lhzux	31	rD		rA			rB			311				0
lhzx	31	rD		rA			rB			279				0
lmw	46	rD		rA			d							
lswi	31	rD		rA			NB			597				0
lswx	31	rD		rA			rB			533				0
lwarx	31	rD		rA			rB			20				0
lwbrx	31	rD		rA			rB			534				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
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
mfdcr	31	rD		DCRF						323				0
mfmsr	31	rD		00000			00000			83				0
mfspr	31	rD		SPRF						339				0
mftb	31	rD		TBRF						371				0
mtcrf	31	rS		0	CRM				0	144				0
mtdcr	31	rS		DCRF						451				0
mtmsr	31	rS		00000			00000			146				0
mtspr	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
mulhhw	**4**	**rD**		**rA**			**rB**			**40**				**Rc**
mulhhwu	**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**
nmaclhw	**4**	**rD**		**rA**			**rB**			**OE**	**430**			**Rc**
nmaclhws	**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							
wrtee	**31**	rS		00000			00000			131				0
wrteei	**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**							
mulhhwu	**4**	rD		rA			rB			**8**				Rc
machhwu	**4**	rD		rA			rB			**OE**	**12**			Rc
mulhhw	**4**	rD		rA			rB			**40**				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
nmachhws	**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
mullhwu	**4**	rD		rA			rB			**392**				Rc
maclhwu	**4**	rD		rA			rB			**OE**	**396**			Rc
mullhw	**4**	rD		rA			rB			**424**				Rc
maclhw	**4**	rD		rA			rB			**OE**	**428**			Rc
nmaclhw	**4**	rD		rA			rB			**OE**	**430**			Rc
maclhwsu	**4**	rD		rA			rB			**OE**	**460**			Rc
maclhws	**4**	rD		rA			rB			**OE**	**492**			Rc
nmaclhws	**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
mfcr	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
wrtee	**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**
wrteei	**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 though 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**
mulhhw	**4**	**rD**	**rA**	**rB**		**40**	**Rc**
mulhhwu	**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**	**crf**D	**00**	**r**A	**r**B	**0**	**0**
cmpi	**11**	**crf**D	**00**	**r**A	**SIMM**		
cmpl	**31**	**crf**D	**00**	**r**A	**r**B	**32**	**0**
cmpli	**10**	**crf**D	**00**	**r**A	**SIMM**		

Table B-7: **Integer Logical Instructions**

	0	6	11	16	21	31
and	**31**	**r**S	**r**A	**r**B	**28**	**Rc**
andc	**31**	**r**S	**r**A	**r**B	**60**	**Rc**
andi.	**28**	**r**S	**r**A	**UIMM**		
andis.	**29**	**r**S	**r**A	**UIMM**		
cntlzw	**31**	**r**S	**r**A	**00000**	**26**	**Rc**
eqv	**31**	**r**S	**r**A	**r**B	**284**	**Rc**
extsb	**31**	**r**S	**r**A	**00000**	**954**	**Rc**
extsh	**31**	**r**S	**r**A	**00000**	**922**	**Rc**
nand	**31**	**r**S	**r**A	**r**B	**476**	**Rc**
nor	**31**	**r**S	**r**A	**r**B	**124**	**Rc**
or	**31**	**r**S	**r**A	**r**B	**444**	**Rc**
orc	**31**	**r**S	**r**A	**r**B	**412**	**Rc**
ori	**24**	**r**S	**r**A	**UIMM**		
oris	**25**	**r**S	**r**A	**UIMM**		
xor	**31**	**r**S	**r**A	**r**B	**316**	**Rc**
xori	**26**	**r**S	**r**A	**UIMM**		
xoris	**27**	**r**S	**r**A	**UIMM**		

Table B-8: **Integer Rotate Instructions**

	0	6	11	16	21	26	31
rlwimi	**20**	**r**S	**r**A	**SH**	**MB**	**ME**	**Rc**
rlwinm	**21**	**r**S	**r**A	**SH**	**MB**	**ME**	**Rc**
rlwnm	**23**	**r**S	**r**A	**r**B	**MB**	**ME**	**Rc**

Table B-9: **Integer Shift Instructions**

	0	6	11	16	21	31
slw	**31**	**r**S	**r**A	**r**B	**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
mfcr	31	rD		00000		00000			19	0
mfdcr	31	rD		DCRF					323	0
mfmsr	31	rD		00000		00000			83	0
mfspr	31	rD		SPRF					339	0
mftb	31	rD		TBRF					371	0
mtcrf	31	rS		0	CRM			0	144	0
mtdcr	31	rS		DCRF					451	0
mtmsr	31	rS		00000		00000			146	0
mtspr	31	rS		SPRF					467	0
wrtee	31	rS		00000		00000			131	0
wrteei	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 though 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**	
mulli	**7**	**rD**		**rA**	**SIMM**	
subfic	**8**	**rD**		**rA**	**SIMM**	
cmpli	**10**	**crfD**	**00**	**rA**	**SIMM**	

Table B-24: **D Form *(Continued)***

	0	6	9	11	16	31
cmpi	**11**	**crf**D	**00**	**r**A	**SIMM**	
addic	**12**	**r**D		**r**A	**SIMM**	
addic.	**13**	**r**D		**r**A	**SIMM**	
addi	**14**	**r**D		**r**A	**SIMM**	
addis	**15**	**r**D		**r**A	**SIMM**	
ori	**24**	**r**S		**r**A	**UIMM**	
oris	**25**	**r**S		**r**A	**UIMM**	
xori	**26**	**r**S		**r**A	**UIMM**	
xoris	**27**	**r**S		**r**A	**UIMM**	
andi.	**28**	**r**S		**r**A	**UIMM**	
andis.	**29**	**r**S		**r**A	**UIMM**	
lwz	**32**	**r**D		**r**A	**d**	
lwzu	**33**	**r**D		**r**A	**d**	
lbz	**34**	**r**D		**r**A	**d**	
lbzu	**35**	**r**D		**r**A	**d**	
stw	**36**	**r**S		**r**A	**d**	
stwu	**37**	**r**S		**r**A	**d**	
stb	**38**	**r**S		**r**A	**d**	
stbu	**39**	**r**S		**r**A	**d**	
lhz	**40**	**r**D		**r**A	**d**	
lhzu	**41**	**r**D		**r**A	**d**	
lha	**42**	**r**D		**r**A	**d**	
lhau	**43**	**r**D		**r**A	**d**	
sth	**44**	**r**S		**r**A	**d**	
sthu	**45**	**r**S		**r**A	**d**	
lmw	**46**	**r**D		**r**A	**d**	
stmw	**47**	**r**S		**r**A	**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**	**r**S	**r**A	**SH**	**MB**	**ME**	**Rc**
rlwinm	**21**	**r**S	**r**A	**SH**	**MB**	**ME**	**Rc**
rlwnm	**23**	**r**S	**r**A	**r**B	**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
mulhhwu	**4**	**rD**		**rA**	**rB**		**8**	**Rc**
mulhhw	**4**	**rD**		**rA**	**rB**		**40**	**Rc**
mulchwu	**4**	**rD**		**rA**	**rB**		**136**	**Rc**
mulchw	**4**	**rD**		**rA**	**rB**		**168**	**Rc**
mullhwu	**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**
mfcr	**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**
mfmsr	**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**
wrtee	**31**	**rS**		**00000**	**00000**		**131**	**0**
mtmsr	**31**	**rS**		**00000**	**00000**		**146**	**0**
stwcx.	**31**	**rS**		**rA**	**rB**		**150**	**1**
stwx	**31**	**rS**		**rA**	**rB**		**151**	**0**
wrteei	**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**
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**

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**
mfdcr	**31**	**rD**	**DCRF**			**323**	**0**
mfspr	**31**	**rD**	**SPRF**			**339**	**0**
mftb	**31**	**rD**	**TBRF**			**371**	**0**
mtdcr	**31**	**rS**	**DCRF**			**451**	**0**
mtspr	**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
nmachhws	**4**	**rD**	**rA**	**rB**	**OE**	**110**	**Rc**
macchwu	**4**	**rD**	**rA**	**rB**	**OE**	**140**	**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**
maclhwu	**4**	**rD**	**rA**	**rB**	**OE**	**396**	**Rc**
maclhw	**4**	**rD**	**rA**	**rB**	**OE**	**428**	**Rc**
nmaclhw	**4**	**rD**	**rA**	**rB**	**OE**	**430**	**Rc**
maclhwsu	**4**	**rD**	**rA**	**rB**	**OE**	**460**	**Rc**
maclhws	**4**	**rD**	**rA**	**rB**	**OE**	**492**	**Rc**
nmaclhws	**4**	**rD**	**rA**	**rB**	**OE**	**494**	**Rc**
subfc	**31**	**rD**	**rA**	**rB**	**OE**	**8**	**Rc**
addc	**31**	**rD**	**rA**	**rB**	**OE**	**10**	**Rc**
mulhwu	**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
maclhw				x	UISA			XO
maclhws				x	UISA			XO
maclhwsu				x	UISA			XO
maclhwu				x	UISA			XO
mcrf	x	x	x		UISA			XL
mcrxr	x	x	x		UISA			X
mfcr	x	x	x		UISA			X
mfdcr		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
mtdcr		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
nmaclhw				x	UISA			XO
nmaclhws				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
rfci		x	x		OEA	x		XL
rfi	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 572
add.	Add and Record		
addc	Add Carrying		page 573
addc.	Add Carrying and Record		
addco	Add Carrying with Overflow Enabled		
addco.	Add Carrying with Overflow Enabled and Record		
adde	Add Extended		page 574
adde.	Add Extended and Record		
addeo	Add Extended with Overflow Enabled		
addeo.	Add Extended with Overflow Enabled and Record		
addi	Add Immediate		page 575
addic	Add Immediate Carrying		page 576
addic.	Add Immediate Carrying and Record		page 577
addis	Add Immediate Shifted		page 578
addme	Add to Minus One Extended		page 579
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 572
addo.	Add with Overflow Enabled and Record		
addze	Add to Zero Extended		page 580
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 581
and.	AND and Record		
andc	AND with Complement		page 582
andc.	AND with Complement and Record		
andi.	AND Immediate and Record		page 583
andis.	AND Immediate Shifted and Record		page 584
b	Branch		page 585
ba	Branch Absolute		
bc	Branch Conditional		page 586
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 588
bcctrl	Branch Conditional to Count Register and Link		
bcl	Branch Conditional and Link		page 586
bcla	Branch Conditional Absolute and Link		
bclr	Branch Conditional to Link Register		page 590
bclrl	Branch Conditional to Link Register and Link		
bctr	Branch to Count Register	**bcctr**	page 823
bctrl	Branch to Count Register and Link	**bcctrl**	page 824
bdnz	Branch if Decremented CTR Not Zero	**bc**	page 822
bdnza	Branch if Decremented CTR Not Zero Absolute	**bca**	page 822
bdnzf	Branch if Decremented CTR Not Zero and Condition False	**bc**	page 822
bdnzfa	Branch if Decremented CTR Not Zero and Condition False Absolute	**bca**	page 822
bdnzfl	Branch if Decremented CTR Not Zero and Condition False and Link	**bcl**	page 823
bdnzfla	Branch if Decremented CTR Not Zero and Condition False Absolute and Link	**bcla**	page 823
bdnzflr	Branch if Decremented CTR Not Zero and Condition False to Link Register	**bclr**	page 823
bdnzflrl	Branch if Decremented CTR Not Zero and Condition False to Link Register and Link	**bclrl**	page 824
bdnzl	Branch if Decremented CTR Not Zero and Link	**bcl**	page 823
bdnzla	Branch if Decremented CTR Not Zero Absolute and Link	**bcla**	page 823
bdnzlr	Branch if Decremented CTR Not Zero to Link Register	**bclr**	page 823
bdnzlrl	Branch if Decremented CTR Not Zero to Link Register and Link	**bclrl**	page 824
bdnzt	Branch if Decremented CTR Not Zero and Condition True	**bc**	page 822
bdnzta	Branch if Decremented CTR Not Zero and Condition True Absolute	**bca**	page 822
bdnztl	Branch if Decremented CTR Not Zero and Condition True and Link	**bcl**	page 823
bdnztla	Branch if Decremented CTR Not Zero and Condition True Absolute and Link	**bcla**	page 823
bdnztlr	Branch if Decremented CTR Not Zero and Condition True to Link Register	**bclr**	page 823
bdnztlrl	Branch if Decremented CTR Not Zero and Condition True to Link Register and Link	**bclrl**	page 824
bdz	Branch if Decremented CTR Zero	**bc**	page 822
bdza	Branch if Decremented CTR Zero Absolute	**bca**	page 822
bdzf	Branch if Decremented CTR Zero and Condition False	**bc**	page 822
bdzfa	Branch if Decremented CTR Zero and Condition False Absolute	**bca**	page 822
bdzfl	Branch if Decremented CTR Zero and Condition False and Link	**bcl**	page 823
bdzfla	Branch if Decremented CTR Zero and Condition False Absolute and Link	**bcla**	page 823
bdzflr	Branch if Decremented CTR Zero and Condition False to Link Register	**bclr**	page 823
bdzflrl	Branch if Decremented CTR Zero and Condition False to Link Register and Link	**bclrl**	page 824
bdzl	Branch if Decremented CTR Zero and Link	**bcl**	page 823
bdzla	Branch if Decremented CTR Zero Absolute and Link	**bcla**	page 823

Table B-33: **Complete List of Instruction Mnemonics *(Continued)***

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
bdzlr	Branch if Decremented CTR Zero to Link Register	**bclr**	page 823
bdzlrl	Branch if Decremented CTR Zero to Link Register and Link	**bclrl**	page 824
bdzt	Branch if Decremented CTR Zero and Condition True	**bc**	page 822
bdzta	Branch if Decremented CTR Zero and Condition True Absolute	**bca**	page 822
bdztl	Branch if Decremented CTR Zero and Condition True and Link	**bcl**	page 823
bdztla	Branch if Decremented CTR Zero and Condition True Absolute and Link	**bcla**	page 823
bdztlr	Branch if Decremented CTR Zero and Condition True to Link Register	**bclr**	page 823
bdztlrl	Branch if Decremented CTR Zero and Condition True to Link Register and Link	**bclrl**	page 824
beq	Branch if Equal	**bc**	page 825
beqa	Branch if Equal Absolute	**bca**	page 825
beqctr	Branch if Equal to Count Register	**bcctr**	page 826
beqctrl	Branch if Equal to Count Register and Link	**bcctrl**	page 827
beql	Branch if Equal and Link	**bcl**	page 826
beqla	Branch if Equal Absolute and Link	**bcla**	page 826
beqlr	Branch if Equal to Link Register	**bclr**	page 826
beqlrl	Branch if Equal to Link Register and Link	**bclrl**	page 827
bf	Branch if Condition False	**bc**	page 822
bfa	Branch if Condition False Absolute	**bca**	page 822
bfctr	Branch if Condition False to Count Register	**bcctr**	page 823
bfctrl	Branch if Condition False to Count Register and Link	**bcctrl**	page 824
bfl	Branch if Condition False and Link	**bcl**	page 823
bfla	Branch if Condition False Absolute and Link	**bcla**	page 823
bflr	Branch if Condition False to Link Register	**bclr**	page 823
bflrl	Branch if Condition False to Link Register and Link	**bclrl**	page 824
bge	Branch if Greater Than or Equal	**bc**	page 825
bgea	Branch if Greater Than or Equal Absolute	**bca**	page 825
bgectr	Branch if Greater Than or Equal to Count Register	**bcctr**	page 826
bgectrl	Branch if Greater Than or Equal to Count Register and Link	**bcctrl**	page 827
bgel	Branch if Greater Than or Equal and Link	**bcl**	page 826
bgela	Branch if Greater Than or Equal Absolute and Link	**bcla**	page 826
bgelr	Branch if Greater Than or Equal to Link Register	**bclr**	page 826
bgelrl	Branch if Greater Than or Equal to Link Register and Link	**bclrl**	page 827
bgt	Branch if Greater Than	**bc**	page 825
bgta	Branch if Greater Than Absolute	**bca**	page 825
bgtctr	Branch if Greater Than to Count Register	**bcctr**	page 826
bgtctrl	Branch if Greater Than to Count Register and Link	**bcctrl**	page 827

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 826
bgtla	Branch if Greater Than Absolute and Link	**bcla**	page 826
bgtlr	Branch if Greater Than to Link Register	**bclr**	page 826
bgtlrl	Branch if Greater Than to Link Register and Link	**bclrl**	page 827
bl	Branch and Link		page 585
bla	Branch Absolute and Link		
ble	Branch if Less Than or Equal	**bc**	page 825
blea	Branch if Less Than or Equal Absolute	**bca**	page 825
blectr	Branch if Less Than or Equal to Count Register	**bcctr**	page 826
blectrl	Branch if Less Than or Equal to Count Register and Link	**bcctrl**	page 827
blel	Branch if Less Than or Equal and Link	**bcl**	page 826
blela	Branch if Less Than or Equal Absolute and Link	**bcla**	page 826
blelr	Branch if Less Than or Equal to Link Register	**bclr**	page 826
blelrl	Branch if Less Than or Equal to Link Register and Link	**bclrl**	page 827
blr	Branch to Link Register	**bclr**	page 823
blrl	Branch to Link Register and Link	**bclrl**	page 824
blt	Branch if Less Than	**bc**	page 825
blta	Branch if Less Than Absolute	**bca**	page 825
bltctr	Branch if Less Than to Count Register	**bcctr**	page 826
bltctrl	Branch if Less Than to Count Register and Link	**bcctrl**	page 827
bltl	Branch if Less Than and Link	**bcl**	page 826
bltla	Branch if Less Than Absolute and Link	**bcla**	page 826
bltlr	Branch if Less Than to Link Register	**bclr**	page 826
bltlrl	Branch if Less Than to Link Register and Link	**bclrl**	page 827
bne	Branch if Not Equal	**bc**	page 825
bnea	Branch if Not Equal Absolute	**bca**	page 825
bnectr	Branch if Not Equal to Count Register	**bcctr**	page 826
bnectrl	Branch if Not Equal to Count Register and Link	**bcctrl**	page 827
bnel	Branch if Not Equal and Link	**bcl**	page 826
bnela	Branch if Not Equal Absolute and Link	**bcla**	page 826
bnelr	Branch if Not Equal to Link Register	**bclr**	page 826
bnelrl	Branch if Not Equal to Link Register and Link	**bclrl**	page 827
bng	Branch if Not Greater Than	**bc**	page 825
bnga	Branch if Not Greater Than Absolute	**bca**	page 825
bngctr	Branch if Not Greater Than to Count Register	**bcctr**	page 826
bngctrl	Branch if Not Greater Than to Count Register and Link	**bcctrl**	page 827

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 826
bngla	Branch if Not Greater Than Absolute and Link	**bcla**	page 826
bnglr	Branch if Not Greater Than to Link Register	**bclr**	page 826
bnglrl	Branch if Not Greater Than to Link Register and Link	**bclrl**	page 827
bnl	Branch if Not Less Than	**bc**	page 825
bnla	Branch if Not Less Than Absolute	**bca**	page 825
bnlctr	Branch if Not Less Than to Count Register	**bcctr**	page 826
bnlctrl	Branch if Not Less Than to Count Register and Link	**bcctrl**	page 827
bnll	Branch if Not Less Than and Link	**bcl**	page 826
bnlla	Branch if Not Less Than Absolute and Link	**bcla**	page 826
bnllr	Branch if Not Less Than to Link Register	**bclr**	page 826
bnllrl	Branch if Not Less Than to Link Register and Link	**bclrl**	page 827
bns	Branch if Not Summary Overflow	**bc**	page 825
bnsa	Branch if Not Summary Overflow Absolute	**bca**	page 825
bnsctr	Branch if Not Summary Overflow to Count Register	**bcctr**	page 826
bnsctrl	Branch if Not Summary Overflow to Count Register and Link	**bcctrl**	page 827
bnsl	Branch if Not Summary Overflow and Link	**bcl**	page 826
bnsla	Branch if Not Summary Overflow Absolute and Link	**bcla**	page 826
bnslr	Branch if Not Summary Overflow to Link Register	**bclr**	page 826
bnslrl	Branch if Not Summary Overflow to Link Register and Link	**bclrl**	page 827
bso	Branch if Summary Overflow	**bc**	page 825
bsoa	Branch if Summary Overflow Absolute	**bca**	page 825
bsoctr	Branch if Summary Overflow to Count Register	**bcctr**	page 826
bsoctrl	Branch if Summary Overflow to Count Register and Link	**bcctrl**	page 827
bsol	Branch if Summary Overflow and Link	**bcl**	page 826
bsola	Branch if Summary Overflow Absolute and Link	**bcla**	page 826
bsolr	Branch if Summary Overflow to Link Register	**bclr**	page 826
bsolrl	Branch if Summary Overflow to Link Register and Link	**bclrl**	page 827
bt	Branch if Condition True	**bc**	page 822
bta	Branch if Condition True Absolute	**bca**	page 822
btctr	Branch if Condition True to Count Register	**bcctr**	page 823
btctrl	Branch if Condition True to Count Register and Link	**bcctrl**	page 824
btl	Branch if Condition True and Link	**bcl**	page 823
btla	Branch if Condition True Absolute and Link	**bcla**	page 823
btlr	Branch if Condition True to Link Register	**bclr**	page 823
btlrl	Branch if Condition True to Link Register and Link	**bclrl**	page 824

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 829
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 592
cmpi	Compare Immediate		page 593
cmpl	Compare Logical		page 594
cmpli	Compare Logical Immediate		page 595
cmplw	Compare Logical Word	**cmpl**	page 828
cmplwi	Compare Logical Word Immediate	**cmpli**	
cmpw	Compare Word	**cmp**	
cmpwi	Compare Word Immediate	**cmpi**	
cntlzw	Count Leading Zeros Word		page 596
cntlzw.	Count Leading Zeros Word and Record		
crand	Condition Register AND		page 597
crandc	Condition Register AND with Complement		page 598
crclr	Condition Register Clear	**crxor**	page 828
creqv	Condition Register Equivalent		page 599
crmove	Condition Register Move	**cror**	page 828
crnand	Condition Register NAND		page 600
crnor	Condition Register NOR		page 601
crnot	Condition Register Not	**crnor**	page 828
cror	Condition Register OR		page 602
crorc	Condition Register OR with Complement		page 603
crset	Condition Register Set	**creqv**	page 828
crxor	Condition Register XOR		page 604
dcba	Data Cache Block Allocate		page 605
dcbf	Data Cache Block Flush		page 607
dcbi	Data Cache Block Invalidate		page 609
dcbst	Data Cache Block Store		page 611
dcbt	Data Cache Block Touch		page 613
dcbtst	Data Cache Block Touch for Store		page 615
dcbz	Data Cache Block Clear to Zero		page 617
dccci	Data Cache Congruence Class Invalidate		page 619

Table B-33: **Complete List of Instruction Mnemonics *(Continued)***

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
dcread	Data Cache Read		page 621
divw	Divide Word		page 623
divw.	Divide Word and Record		
divwo	Divide Word with Overflow Enabled		
divwo.	Divide Word with Overflow Enabled and Record		
divwu	Divide Word Unsigned		page 625
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 626
eqv	Equivalent		page 627
eqv.	Equivalent and Record		
extlwi	Extract and Left Justify Immediate	**rlwinm**	page 829
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 628
extsb.	Extend Sign Byte and Record		
extsh	Extend Sign Halfword		page 629
extsh.	Extend Sign Halfword and Record		
icbi	Instruction Cache Block Invalidate		page 630
icbt	Instruction Cache Block Touch		page 632
iccci	Instruction Cache Congruence Class Invalidate		page 634
icread	Instruction Cache Read		page 635
inslwi	Insert from Left Immediate	**rlwimi**	page 829
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 637
la	Load Address	**addi**	page 834
lbz	Load Byte and Zero		page 638
lbzu	Load Byte and Zero with Update		page 639
lbzux	Load Byte and Zero with Update Indexed		page 640
lbzx	Load Byte and Zero Indexed		page 641
lha	Load Halfword Algebraic		page 642
lhau	Load Halfword Algebraic with Update		page 643

Table B-33: **Complete List of Instruction Mnemonics *(Continued)***

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
lhaux	Load Halfword Algebraic with Update Indexed		page 644
lhax	Load Halfword Algebraic Indexed		page 645
lhbrx	Load Halfword Byte-Reverse Indexed		page 646
lhz	Load Halfword and Zero		page 647
lhzu	Load Halfword and Zero with Update		page 648
lhzux	Load Halfword and Zero with Update Indexed		page 649
lhzx	Load Halfword and Zero Indexed		page 650
li	Load Immediate	**addi**	page 834
lis	Load Immediate Shifted	**addis**	page 834
lmw	Load Multiple Word		page 651
lswi	Load String Word Immediate		page 653
lswx	Load String Word Indexed		page 655
lwarx	Load Word and Reserve Indexed		page 657
lwbrx	Load Word Byte-Reverse Indexed		page 658
lwz	Load Word and Zero		page 659
lwzu	Load Word and Zero with Update		page 660
lwzux	Load Word and Zero with Update Indexed		page 661
lwzx	Load Word and Zero Indexed		page 662
macchw	Multiply Accumulate Cross Halfword to Word Modulo Signed		page 663
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 664
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 665
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 666
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 667
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 668
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 669
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 670
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 671
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 672
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 673
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 674
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 675
mcrxr	Move to Condition Register from XER		page 676
mfccr0	Move From Core-Configuration Register 0	**mfspr**	page 830
mfcr	Move from Condition Register		page 677
mfctr	Move From Count Register	**mfspr**	page 830
mfdac1	Move From Data Address-Compare 1		
mfdac2	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		
mfdccr	Move From Data-Cache Cachability Register		
mfdcr	Move from Device Control Register		page 678
mfdcwr	Move From Data-Cache Write-Through Register	**mfspr**	page 830
mfdear	Move From Data-Error Address Register		
mfdvc1	Move From Data Value-Compare 1		
mfdvc2	Move From Data Value-Compare 2		
mfesr	Move From Exception-Syndrome Register		
mfevpr	Move From Exception-Vector Prefix Register		
mfiac1	Move From Instruction Address-Compare 1		
mfiac2	Move From Instruction Address-Compare 2		
mfiac3	Move From Instruction Address-Compare 3		
mfiac4	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 679
mfpid	Move From Process ID Register	**mfspr**	page 830
mfpit	Move From Programmable-Interval Timer		
mfpvr	Move From Processor-Version Register		
mfsgr	Move From Storage Guarded Register		
mfsler	Move From Storage Little-Endian Register		
mfspr	Move from Special Purpose Register		page 680

Table B-33: **Complete List of Instruction Mnemonics *(Continued)***

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
mfsprg0	Move From SPR General-Purpose Register 0	**mfspr**	page 830
mfsprg1	Move From SPR General-Purpose Register 1		
mfsprg2	Move From SPR General-Purpose Register 2		
mfsprg3	Move From SPR General-Purpose Register 3		
mfsprg4	Move From SPR General-Purpose Register 4		
mfsprg5	Move From SPR General-Purpose Register 5		
mfsprg6	Move From SPR General-Purpose Register 6		
mfsprg7	Move From SPR General-Purpose Register 7		
mfsrr0	Move From Save/Restore Register 0		
mfsrr1	Move From Save/Restore Register 1		
mfsrr2	Move From Save/Restore Register 2		
mfsrr3	Move From Save/Restore Register 3		
mfsu0r	Move From Storage User-Defined 0 Register		
mftb	Move from Time Base Register		page 681
mftbl	Move From Time-Base Lower	**mfspr**	page 830
mftbu	Move From Time-Base Upper		
mftcr	Move From Timer-Control Register		
mftsr	Move From Timer-Status Register		
mfusprg0	Move From User SPR General-Purpose Register 0		
mfxer	Move From Fixed-Point Exception Register		
mfzpr	Move From Zone-Protection Register		
mr	Move Register	**or**	page 834
mr.	Move Register and Record	**or.**	
mtccr0	Move to Core-Configuration Register 0	**mtspr**	page 830
mtcr	Move to Condition Register	**mtcrf**	page 835
mtcrf	Move to Condition Register Fields		page 682
mtctr	Move to Count Register	**mtspr**	page 830
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 684

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 830
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		
mtevpr	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 685
mtpid	Move to Process ID Register	**mtspr**	page 830
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 686
mtsprg0	Move to SPR General-Purpose Register 0	**mtspr**	page 830
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 830
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 830
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 687
mulchw.	Multiply Cross Halfword to Word Signed and Record		
mulchwu	Multiply Cross Halfword to Word Unsigned		page 688
mulchwu.	Multiply Cross Halfword to Word Unsigned and Record		
mulhhw	Multiply High Halfword to Word Signed		page 689
mulhhw.	Multiply High Halfword to Word Signed and Record		
mulhhwu	Multiply High Halfword to Word Unsigned		page 690
mulhhwu.	Multiply High Halfword to Word Unsigned and Record		
mulhw	Multiply High Word		page 691
mulhw.	Multiply High Word and Record		
mulhwu	Multiply High Word Unsigned		page 692
mulhwu.	Multiply High Word Unsigned and Record		
mullhw	Multiply Low Halfword to Word Signed		page 693
mullhw.	Multiply Low Halfword to Word Signed and Record		
mullhwu	Multiply Low Halfword to Word Unsigned		page 694
mullhwu.	Multiply Low Halfword to Word Unsigned and Record		
mulli	Multiply Low Immediate		page 695
mullw	Multiply Low Word		page 696
mullw.	Multiply Low Word and Record		
mullwo	Multiply Low Word with Overflow Enabled		page 696
mullwo.	Multiply Low Word with Overflow Enabled and Record		
nand	NAND		page 697
nand.	NAND and Record		
neg	Negate		page 698
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 699
nmacchw.	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed and Record		
nmacchwo	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed with Overflow Enabled		
nmacchwo.	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 700
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 701
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 702
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 703
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 704
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 834
nor	NOR		page 705
nor.	NOR and Record		
not	Complement (Not) Register	**nor**	page 835
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 706
or.	OR and Record		
orc	OR with Complement		page 707
orc.	OR with Complement and Record		
ori	OR Immediate		page 708
oris	OR Immediate Shifted		page 709
rfci	Return from Critical Interrupt		page 710
rfi	Return from Interrupt		page 711
rlwimi	Rotate Left Word Immediate then Mask Insert		page 712
rlwimi.	Rotate Left Word Immediate then Mask Insert and Record		
rlwinm	Rotate Left Word Immediate then AND with Mask		page 713
rlwinm.	Rotate Left Word Immediate then AND with Mask and Record		
rlwnm	Rotate Left Word then AND with Mask		page 714
rlwnm.	Rotate Left Word then AND with Mask and Record		
rotlw	Rotate Left	**rlwinm**	page 829
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 715
slw	Shift Left Word		page 716
slw.	Shift Left Word and Record		
slwi	Shift Left Immediate	**rlwinm**	page 829
slwi.	Shift Left Immediate and Record	**rlwinm.**	
sraw	Shift Right Algebraic Word		page 717
sraw.	Shift Right Algebraic Word and Record		
srawi	Shift Right Algebraic Word Immediate		page 718
srawi.	Shift Right Algebraic Word Immediate and Record		
srw	Shift Right Word		page 719
srw.	Shift Right Word and Record		
srwi	Shift Right Immediate	**rlwinm**	page 829
srwi.	Shift Right Immediate and Record	**rlwinm.**	
stb	Store Byte		page 720
stbu	Store Byte with Update		page 721
stbux	Store Byte with Update Indexed		page 722

Table B-33: **Complete List of Instruction Mnemonics *(Continued)***

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
stbx	Store Byte Indexed		page 723
sth	Store Halfword		page 724
sthbrx	Store Halfword Byte-Reverse Indexed		page 725
sthu	Store Halfword with Update		page 726
sthux	Store Halfword with Update Indexed		page 727
sthx	Store Halfword Indexed		page 728
stmw	Store Multiple Word		page 729
stswi	Store String Word Immediate		page 730
stswx	Store String Word Indexed		page 732
stw	Store Word		page 734
stwbrx	Store Word Byte-Reverse Indexed		page 735
stwcx.	Store Word Conditional Indexed		page 737
stwu	Store Word with Update		page 739
stwux	Store Word with Update Indexed		page 740
stwx	Store Word Indexed		page 741
sub	Subtract	**subf**	page 832
sub.	Subtract and Record	**subf.**	
subc	Subtract Carrying	**subfc**	page 832
subc.	Subtract Carrying and Record	**subfc.**	
subco	Subtract Carrying with Overflow Enabled	**subfco**	page 832
subco.	Subtract Carrying with Overflow Enabled and Record	**subfco.**	
subf	Subtract from		page 742
subf.	Subtract from and Record		
subfc	Subtract from Carrying		page 743
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 744
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 745
subfme	Subtract from Minus One Extended		page 746
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 742
subfo.	Subtract from with Overflow Enabled and Record		
subfze	Subtract from Zero Extended		page 747
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 832
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 748
tlbia	TLB Invalidate All		page 749
tlbre	TLB Read Entry		page 750
tlbrehi	Read TLBHI Portion of TLB Entry	**tlbre**	page 832
tlbrelo	Read TLBLO Portion of TLB Entry		
tlbsx	TLB Search Indexed		page 752
tlbsx.	TLB Search Indexed and Record		
tlbsync	TLB Synchronize		page 754
tlbwe	TLB Write Entry		page 755
tlbwehi	Write TLBHI Portion of TLB Entry	**tlbwe**	page 832
tlbwelo	Write TLBLO Portion of TLB Entry		
trap	Trap if Unconditional	**tw**	page 833
tw	Trap Word		page 757
tweq	Trap if Equal	**tw**	page 833
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 759

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 833
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**	
wrtee	Write External Enable		page 761
wrteei	Write External Enable Immediate		page 762
xor	XOR		page 763
xor.	XOR and Record		
xori	XOR Immediate		page 764
xoris	XOR Immediate Shifted		page 765

XILINX®

Appendix C

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 822 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 825 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 ≠ 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**	—	—	**blrl**	**bctrl**
Branch if Condition True (CR_b=1)	**bt**	**bta**	**btlr**	**btctr**	**btl**	**btla**	**btlrl**	**btctrl**
Branch if Condition False (CR_b=0)	**bf**	**bfa**	**bflr**	**bfctr**	**bfl**	**bfla**	**bflrl**	**bfctrl**
Decrement CTR, Branch if CTR ≠ 0	**bdnz**	**bdnza**	**bdnzlr**	—	**bdnzl**	**bdnzla**	**bdnzlrl**	—
Decrement CTR, Branch if CTR ≠ 0 and Condition True (CR_b=1)	**bdnzt**	**bdnzta**	**bdnztlr**	—	**bdnztl**	**bdnztla**	**bdnztlrl**	—
Decrement CTR, Branch if CTR ≠ 0 and Condition False (CR_b=0)	**bdnzf**	**bdnzfa**	**bdnzflr**	—	**bdnzfl**	**bdnzfla**	**bdnzflrl**	—
Decrement CTR, Branch if CTR = 0	**bdz**	**bdza**	**bdzlr**	—	**bdzl**	**bdzla**	**bdzlrl**	—
Decrement CTR, Branch if CTR = 0 and Condition True (CR_b=1)	**bdzt**	**bdzta**	**bdztlr**	—	**bdztl**	**bdztla**	**bdztlrl**	—
Decrement CTR, Branch if CTR = 0 and Condition False (CR_b=0)	**bdzf**	**bdzfa**	**bdzflr**	—	**bdzfl**	**bdzfla**	**bdzflrl**	—

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** *b*, target	**bc** 12, *b*, target	**bta** *b*, target	**bca** 12, *b*, target
Branch if Condition False (CR_b=0)	**bf** *b*, target	**bc** 4, *b*, target	**bfa** *b*, target	**bca** 4, *b*, target
Decrement CTR, Branch if CTR ≠ 0	**bdnz** target	**bc** 16, 0, target	**bdnza** target	**bca** 16, 0, target
Decrement CTR, Branch if CTR ≠ 0 and Condition True (CR_b=1)	**bdnzt** *b*, target	**bc** 8, *b*, target	**bdnzta** *b*, target	**bca** 8, *b*, target
Decrement CTR, Branch if CTR ≠ 0 and Condition False (CR_b=0)	**bdnzf** *b*, target	**bc** 0, *b*, target	**bdnzfa** *b*, target	**bca** 0, *b*, 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** *b*, target	**bc** 10, *b*, target	**bdzta** *b*, target	**bca** 10, *b*, target
Decrement CTR, Branch if CTR = 0 and Condition False (CR_b=0)	**bdzf** *b*, target	**bc** 2, *b*, target	**bdzfa** *b*, target	**bca** 2, *b*, 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** *b*	**bclr** 12, *b*	**btctr** *b*	**bcctr** 12, *b*
Branch if Condition False (CR_b=0)	**bflr** *b*	**bclr** 4, *b*	**bfctr** *b*	**bcctr** 4, *b*
Decrement CTR, Branch if CTR ≠ 0	**bdnzlr**	**bclr** 16, 0	—	—
Decrement CTR, Branch if CTR ≠ 0 and Condition True (CR_b=1)	**bdnztlr** *b*	**bclr** 8, *b*	—	—
Decrement CTR, Branch if CTR ≠ 0 and Condition False (CR_b=0)	**bdnzflr** *b*	**bclr** 0, *b*	—	—
Decrement CTR, Branch if CTR = 0	**bdzlr**	**bclr** 18, 0	—	—
Decrement CTR, Branch if CTR = 0 and Condition True (CR_b=1)	**bdztlr** *b*	**bclr** 10, *b*	—	—
Decrement CTR, Branch if CTR = 0 and Condition False (CR_b=0)	**bdzflr** *b*	**bclr** 2, *b*	—	—

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** *b*, target	**bcl** 12, *b*, target	**btla** *b*, target	**bcla** 12, *b*, target
Branch if Condition False (CR_b=0)	**bfl** *b*, target	**bcl** 4, *b*, target	**bfla** *b*, target	**bcla** 4, *b*, target
Decrement CTR, Branch if CTR ≠ 0	**bdnzl** target	**bcl** 16, 0, target	**bdnzla** target	**bcla** 16, 0, target
Decrement CTR, Branch if CTR ≠ 0 and Condition True (CR_b=1)	**bdnztl** *b*, target	**bcl** 8, *b*, target	**bdnztla** *b*, target	**bcla** 8, *b*, target
Decrement CTR, Branch if CTR ≠ 0 and Condition False (CR_b=0)	**bdnzfl** *b*, target	**bcl** 0, *b*, target	**bdnzfla** *b*, target	**bcla** 0, *b*, 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** *b*, target	**bcl** 10, *b*, target	**bdztla** *b*, target	**bcla** 10, *b*, 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** *b*, target	**bcl** 2, *b*, target	**bdzfla** *b*, target	**bcla** 2, *b*, 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)	**btlrl** *b*	**bclrl** 12, *b*	**btctrl** *b*	**bcctrl** 12, *b*
Branch if Condition False (CR_b=0)	**bflrl** *b*	**bclrl** 4, *b*	**bfctrl** *b*	**bcctrl** 4, *b*
Decrement CTR, Branch if CTR ≠ 0	**bdnzlrl**	**bclrl** 16, 0	—	—
Decrement CTR, Branch if CTR ≠ 0 and Condition True (CR_b=1)	**bdnztlrl** *b*	**bclrl** 8, *b*	—	—
Decrement CTR, Branch if CTR ≠ 0 and Condition False (CR_b=0)	**bdnzflrl** *b*	**bclrl** 0, *b*	—	—
Decrement CTR, Branch if CTR = 0	**bdzlrl**	**bclrl** 18, 0	—	—
Decrement CTR, Branch if CTR = 0 and Condition True (CR_b=1)	**bdztlrl** *b*	**bclrl** 10, *b*	—	—
Decrement CTR, Branch if CTR = 0 and Condition False (CR_b=0)	**bdzflrl** *b*	**bclrl** 2, *b*	—	—

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 821.

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**	**bgelr**	**bgectr**	**bgel**	**bgela**	**bgelrl**	**bgectrl**
Branch if Greater Than	**bgt**	**bgta**	**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**	**bnectr**	**bnel**	**bnela**	**bnelrl**	**bnectrl**
Branch if Not Greater Than	**bng**	**bnga**	**bnglr**	**bngctr**	**bngl**	**bngla**	**bnglrl**	**bngctrl**
Branch if Summary Overflow	**bso**	**bsoa**	**bsolr**	**bsoctr**	**bsol**	**bsola**	**bsolrl**	**bsoctrl**
Branch if Not Summary Overflow	**bns**	**bnsa**	**bnslr**	**bnsctr**	**bnsl**	**bnsla**	**bnslrl**	**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×*n*+0, target	**blta** *n*, target	**bca** 12, 4×*n*+0, target
Branch if Less Than or Equal	**ble** *n*, target	**bc** 4, 4×*n*+1, target	**blea** *n*, target	**bca** 4, 4×*n*+1, target
Branch if Equal	**beq** *n*, target	**bc** 12, 4×*n*+2, target	**beqa** *n*, target	**bca** 12, 4×*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** *n*, target	**bc** 4, 4×*n*+0, target	**bgea** *n*, target	**bca** 4, 4×*n*+0, target
Branch if Greater Than	**bgt** *n*, target	**bc** 12, 4×*n*+1, target	**bgta** *n*, target	**bca** 12, 4×*n*+1, target
Branch if Not Less Than	**bnl** *n*, target	**bc** 4, 4×*n*+0, target	**bnla** *n*, target	**bca** 4, 4×*n*+0, target
Branch if Not Equal	**bne** *n*, target	**bc** 4, 4×*n*+2, target	**bnea** *n*, target	**bca** 4, 4×*n*+2, target
Branch if Not Greater Than	**bng** *n*, target	**bc** 4, 4×*n*+1, target	**bnga** *n*, target	**bca** 4, 4×*n*+1, target
Branch if Summary Overflow	**bso** *n*, target	**bc** 12, 4×*n*+3, target	**bsoa** *n*, target	**bca** 12, 4×*n*+3, target
Branch if Not Summary Overflow	**bns** *n*, target	**bc** 4, 4×*n*+3, target	**bnsa** *n*, target	**bca** 4, 4×*n*+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** *n*	**bclr** 12, 4×*n*+0	**bltctr** *n*	**bcctr** 12, 4×*n*+0
Branch if Less Than or Equal	**blelr** *n*	**bclr** 4, 4×*n*+1	**blectr** *n*	**bcctr** 4, 4×*n*+1
Branch if Equal	**beqlr** *n*	**bclr** 12, 4×*n*+2	**beqctr** *n*	**bcctr** 12, 4×*n*+2
Branch if Greater Than or Equal	**bgelr** *n*	**bclr** 4, 4×*n*+0	**bgectr** *n*	**bcctr** 4, 4×*n*+0
Branch if Greater Than	**bgtlr** *n*	**bclr** 12, 4×*n*+1	**bgtctr** *n*	**bcctr** 12, 4×*n*+1
Branch if Not Less Than	**bnllr** *n*	**bclr** 4, 4×*n*+0	**bnlctr** *n*	**bcctr** 4, 4×*n*+0
Branch if Not Equal	**bnelr** *n*	**bclr** 4, 4×*n*+2	**bnectr** *n*	**bcctr** 4, 4×*n*+2
Branch if Not Greater Than	**bnglr** *n*	**bclr** 4, 4×*n*+1	**bngctr** *n*	**bcctr** 4, 4×*n*+1
Branch if Summary Overflow	**bsolr** *n*	**bclr** 12, 4×*n*+3	**bsoctr** *n*	**bcctr** 12, 4×*n*+3
Branch if Not Summary Overflow	**bnslr** *n*	**bclr** 4, 4×*n*+3	**bnsctr** *n*	**bcctr** 4, 4×*n*+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** *n*, target	**bcl** 12, 4×*n*+0, target	**bltla** *n*, target	**bcla** 12, 4×*n*+0, target
Branch if Less Than or Equal	**blel** *n*, target	**bcl** 4, 4×*n*+1, target	**blela** *n*, target	**bcla** 4, 4×*n*+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** *n*, target	**bcl** 12, 4×*n*+2, target	**beqla** *n*, target	**bcla** 12, 4×*n*+2, target
Branch if Greater Than or Equal	**bgel** *n*, target	**bcl** 4, 4×*n*+0, target	**bgela** *n*, target	**bcla** 4, 4×*n*+0, target
Branch if Greater Than	**bgtl** *n*, target	**bcl** 12, 4×*n*+1, target	**bgtla** *n*, target	**bcla** 12, 4×*n*+1, target
Branch if Not Less Than	**bnll** *n*, target	**bcl** 4, 4×*n*+0, target	**bnlla** *n*, target	**bcla** 4, 4×*n*+0, target
Branch if Not Equal	**bnel** *n*, target	**bcl** 4, 4×*n*+2, target	**bnela** *n*, target	**bcla** 4, 4×*n*+2, target
Branch if Not Greater Than	**bngl** *n*, target	**bcl** 4, 4×*n*+1, target	**bngla** *n*, target	**bcla** 4, 4×*n*+1, target
Branch if Summary Overflow	**bsol** *n*, target	**bcl** 12, 4×*n*+3, target	**bsola** *n*, target	**bcla** 12, 4×*n*+3, target
Branch if Not Summary Overflow	**bnsl** *n*, target	**bcl** 4, 4×*n*+3, target	**bnsla** *n*, target	**bcla** 4, 4×*n*+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	**bltlrl** *n*	**bclrl** 12, 4×*n*+0	**bltctrl** *n*	**bcctrl** 12, 4×*n*+0
Branch if Less Than or Equal	**blelrl** *n*	**bclrl** 4, 4×*n*+1	**blectrl** *n*	**bcctrl** 4, 4×*n*+1
Branch if Equal	**beqlrl** *n*	**bclrl** 12, 4×*n*+2	**beqctrl** *n*	**bcctrl** 12, 4×*n*+2
Branch if Greater Than or Equal	**bgelrl** *n*	**bclrl** 4, 4×*n*+0	**bgectrl** *n*	**bcctrl** 4, 4×*n*+0
Branch if Greater Than	**bgtlrl** *n*	**bclrl** 12, 4×*n*+1	**bgtctrl** *n*	**bcctrl** 12, 4×*n*+1
Branch if Not Less Than	**bnllrl** *n*	**bclrl** 4, 4×*n*+0	**bnlctrl** *n*	**bcctrl** 4, 4×*n*+0
Branch if Not Equal	**bnelrl** *n*	**bclrl** 4, 4×*n*+2	**bnectrl** *n*	**bcctrl** 4, 4×*n*+2
Branch if Not Greater Than	**bnglrl** *n*	**bclrl** 4, 4×*n*+1	**bngctrl** *n*	**bcctrl** 4, 4×*n*+1
Branch if Summary Overflow	**bsolrl** *n*	**bclrl** 12, 4×*n*+3	**bsoctrl** *n*	**bcctrl** 12, 4×*n*+3
Branch if Not Summary Overflow	**bnslrl** *n*	**bclrl** 4, 4×*n*+3	**bnsctrl** *n*	**bcctrl** 4, 4×*n*+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 370 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 **crf**D 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 crf**D, **r**A, SIMM	**cmpi crf**D, 0, **r**A, SIMM
Compare Word	**cmpw crf**D, **r**A, **r**B	**cmp crf**D, 0, **r**A, **r**B
Compare Logical Word Immediate	**cmplwi crf**D, **r**A, UIMM	**cmpli crf**D, 0, **r**A, UIMM
Compare Logical Word	**cmplw crf**D, **r**A, **r**B	**cmpl crf**D, 0, **r**A, **r**B

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** *bx*	**crxor** *bx*, *bx*, *bx*
Condition Register Move	**crmove** *bx*, *by*	**cror** *bx*, *by*, *by*
Condition Register Not	**crnot** *bx*, *by*	**crnor** *bx*, *by*, *by*

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** rA, rS, *n*, *b* (*n* > 0)	**rlwinm** rA, rS, *b*, 0, *n*–1
	extlwi. rA, rS, *n*, *b* (*n* > 0)	**rlwinm.** rA, rS, *b*, 0, *n*–1
Extract and Right Justify Immediate	**extrwi** rA, rS, *n*, *b* (*n* > 0)	**rlwinm** rA, rS, *b*+*n*, 32–*n*, 31
	extrwi. rA, rS, *n*, *b* (*n* > 0)	**rlwinm.** rA, rS, *b*+*n*, 32–*n*, 31
Insert from Left Immediate	**inslwi** rA, rS, *n*, *b* (*n* > 0)	**rlwimi** rA, rS, 32–*b*, *b*, (*b*+*n*)–1
	inslwi. rA, rS, *n*, *b* (*n* > 0)	**rlwimi.** rA, rS, 32–*b*, *b*, (*b*+*n*)–1
Insert from Right Immediate	**insrwi** rA, rS, *n*, *b* (*n* > 0)	**rlwimi** rA, rS, 32–(*b*+*n*), *b*, (*b*+*n*)–1
	insrwi. rA, rS, *n*, *b* (*n* > 0)	**rlwimi.** rA, rS, 32–(*b*+*n*), *b*, (*b*+*n*)–1
Rotate Left Immediate	**rotlwi** rA, rS, *n*	**rlwinm** rA, rS, *n*, 0, 31
	rotlwi. rA, rS, *n*	**rlwinm.** rA, rS, *n*, 0, 31
Rotate Right Immediate	**rotrwi** rA, rS, *n*	**rlwinm** rA, rS, 32–*n*, 0, 31
	rotrwi. rA, rS, *n*	**rlwinm.** rA, rS, 32–*n*, 0, 31
Rotate Left	**rotlw** rA, rS, rB	**rlwnm** rA, rS, rB, 0, 31
	rotlw. rA, rS, rB	**rlwnm.** rA, rS, rB, 0, 31
Shift Left Immediate	**slwi** rA, rS, *n* (*n* < 32)	**rlwinm** rA, rS, *n*, 0, 31–*n*
	slwi. rA, rS, *n* (*n* < 32)	**rlwinm.** rA, rS, *n*, 0, 31–*n*
Shift Right Immediate	**srwi** rA, rS, *n* (*n* < 32)	**rlwinm** rA, rS, 32–*n*, *n*, 31
	srwi. rA, rS, *n* (*n* < 32)	**rlwinm.** rA, rS, 32–*n*, *n*, 31

Table C-15: **Simplified Mnemonics for Rotate and Shift Instructions *(Continued)***

Operation	Simplified Mnemonic	Equivalent Mnemonic
Clear Left Immediate	**clrlwi** **r**A, **r**S, *n* (*n* < 32)	**rlwinm** **r**A, **r**S, 0, *n*, 31
	clrlwi. **r**A, **r**S, *n* (*n* < 32)	**rlwinm.** **r**A, **r**S, 0, *n*, 31
Clear Right Immediate	**clrrwi** **r**A, **r**S, *n* (*n* < 32)	**rlwinm** **r**A, **r**S, 0, 0, 31–*n*
	clrrwi. **r**A, **r**S, *n* (*n* < 32)	**rlwinm.** **r**A, **r**S, 0, 0, 31–*n*
Clear Left and Shift Left Immediate	**clrlslwi** **r**A, **r**S, *b*, *n* (*n* ≤ *b* ≤ 31)	**rlwinm** **r**A, **r**S, *b*–*n*, 31–*n*
	clrlslwi. **r**A, **r**S, *b*, *n* (*n* ≤ *b* ≤ 31)	**rlwinm.** **r**A, **r**S, *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** **r**S	**mtspr** 947, **r**S	**mfccr0** **r**D	**mfspr** **r**D, 947
Count Register	**mtctr** **r**S	**mtspr** 9, **r**S	**mfctr** **r**D	**mfspr** **r**D, 9
Data Address-Compare 1	**mtdac1** **r**S	**mtspr** 1014, **r**S	**mfdac1** **r**D	**mfspr** **r**D, 1014
Data Address-Compare 2	**mtdac2** **r**S	**mtspr** 1015, **r**S	**mfdac2** **r**D	**mfspr** **r**D, 1015
Debug-Control Register 0	**mtdbcr0** **r**S	**mtspr** 1010, **r**S	**mfdbcr0** **r**D	**mfspr** **r**D, 1010
Debug-Control Register 1	**mtdbcr1** **r**S	**mtspr** 957, **r**S	**mfdbcr1** **r**D	**mfspr** **r**D, 957
Debug-Status Register	**mtdbsr** **r**S [1]	**mtspr** 1008, **r**S [1]	**mfdbsr** **r**D	**mfspr** **r**D, 1008
Data-Cache Cachability Register	**mtdccr** **r**S	**mtspr** 1018, **r**S	**mfdccr** **r**D	**mfspr** **r**D, 1018
Data-Cache Write-Through Register	**mtdcwr** **r**S	**mtspr** 954, **r**S	**mfdcwr** **r**D	**mfspr** **r**D, 954
Data-Error Address Register	**mtdear** **r**S	**mtspr** 981, **r**S	**mfdear** **r**D	**mfspr** **r**D, 981
Data Value-Compare 1	**mtdvc1** **r**S	**mtspr** 950, **r**S	**mfdvc1** **r**D	**mfspr** **r**D, 950
Data Value-Compare 2	**mtdvc2** **r**S	**mtspr** 951, **r**S	**mfdvc2** **r**D	**mfspr** **r**D, 951
Exception-Syndrome Register	**mtesr** **r**S	**mtspr** 980, **r**S	**mfesr** **r**D	**mfspr** **r**D, 980
Exception-Vector Prefix Register	**mtevpr** **r**S	**mtspr** 982, **r**S	**mfevpr** **r**D	**mfspr** **r**D, 982
Instruction Address-Compare 1	**mtiac1** **r**S	**mtspr** 1012, **r**S	**mfiac1** **r**D	**mfspr** **r**D, 1012
Instruction Address-Compare 2	**mtiac2** **r**S	**mtspr** 1013, **r**S	**mfiac2** **r**D	**mfspr** **r**D, 1013
Instruction Address-Compare 3	**mtiac3** **r**S	**mtspr** 948, **r**S	**mfiac3** **r**D	**mfspr** **r**D, 948
Instruction Address-Compare 4	**mtiac4** **r**S	**mtspr** 949, **r**S	**mfiac4** **r**D	**mfspr** **r**D, 949
Instruction-Cache Cachability Register	**mticcr** **r**S	**mtspr** 1019, **r**S	**mficcr** **r**D	**mfspr** **r**D, 1019
Instruction-Cache Debug-Data Register	—	—	**mficdbdr** **r**D	**mfspr** **r**D, 979
Link Register	**mtlr** **r**S	**mtspr** 8, **r**S	**mflr** **r**D	**mfspr** **r**D, 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	**mfspr** rD, 945
Programmable-Interval Timer	**mtpit** rS	**mtspr** 987, rS	**mfpit** rD	**mfspr** rD, 987
Processor-Version Register	—	—	**mfpvr** rD	**mfspr** rD, 287
Storage Guarded Register	**mtsgr** rS	**mtspr** 953, rS	**mfsgr** rD	**mfspr** rD, 953
Storage Little-Endian Register	**mtsler** rS	**mtspr** 955, rS	**mfsler** rD	**mfspr** rD, 955
SPR General-Purpose Register 0	**mtsprg0** rS	**mtspr** 272, rS	**mfsprg0** rD	**mfspr** rD, 272
SPR General-Purpose Register 1	**mtsprg1** rS	**mtspr** 273, rS	**mfsprg1** rD	**mfspr** rD, 273
SPR General-Purpose Register 2	**mtsprg2** rS	**mtspr** 274, rS	**mfsprg2** rD	**mfspr** rD, 274
SPR General-Purpose Register 3	**mtsprg3** rS	**mtspr** 275, rS	**mfsprg3** rD	**mfspr** rD, 275
SPR General-Purpose Register 4	—	—	**mfsprg4** rD	**mfspr** rD, 260
SPR General-Purpose Register 4	**mtsprg4** rS	**mtspr** 276, rS	—	—
SPR General-Purpose Register 5	—	—	**mfsprg5** rD	**mfspr** rD, 261
SPR General-Purpose Register 5	**mtsprg5** rS	**mtspr** 277, rS	—	—
SPR General-Purpose Register 6	—	—	**mfsprg6** rD	**mfspr** rD, 262
SPR General-Purpose Register 6	**mtsprg6** rS	**mtspr** 278, rS	—	—
SPR General-Purpose Register 7	—	—	**mfsprg7** rD	**mfspr** rD, 263
SPR General-Purpose Register 7	**mtsprg7** rS	**mtspr** 279, rS	—	—
Save/Restore Register 0	**mtsrr0** rS	**mtspr** 26, rS	**mfsrr0** rD	**mfspr** rD, 26
Save/Restore Register 1	**mtsrr1** rS	**mtspr** 27, rS	**mfsrr1** rD	**mfspr** rD, 27
Save/Restore Register 2	**mtsrr2** rS	**mtspr** 990, rS	**mfsrr2** rD	**mfspr** rD, 990
Save/Restore Register 3	**mtsrr3** rS	**mtspr** 991, rS	**mfsrr3** rD	**mfspr** rD, 991
Storage User-Defined 0 Register	**mtsu0r** rS	**mtspr** 956, rS	**mfsu0r** rD	**mfspr** 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	**mfspr** rD, 986
Timer-Status Register	**mttsr** rS [1]	**mtspr** 984, rS [1]	**mftsr** rD	**mfspr** rD, 984
User SPR General-Purpose Register 0	**mtusprg0** rS	**mtspr** 256, rS	**mfusprg0** rD	**mfspr** rD, 256
Fixed-Point Exception Register	**mtxer** rS	**mtspr** 1, rS	**mfxer** rD	**mfspr** rD, 1
Zone-Protection Register	**mtzpr** rS	**mtspr** 944, rS	**mfzpr** rD	**mfspr** rD, 944

Notes:
1. Performs a clear to zero operation.

Subtract Instructions

The subtract-from instructions subtract the second operand (**r**A) from the third operand (**r**B). 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 (**r**A – **r**B)	**sub** **r**D, **r**A, **r**B	**subf** **r**D, **r**B, **r**A
	sub. **r**D, **r**A, **r**B	**subf.** **r**D, **r**B, **r**A
	subo **r**D, **r**A, **r**B	**subfo** **r**D, **r**B, **r**A
	subo. **r**D, **r**A, **r**B	**subfo.** **r**D, **r**B, **r**A
Subtract Carrying (**r**A – **r**B)	**subc** **r**D, **r**A, **r**B	**subfc** **r**D, **r**B, **r**A
	subc. **r**D, **r**A, **r**B	**subfc.** **r**D, **r**B, **r**A
	subco **r**D, **r**A, **r**B	**subfco** **r**D, **r**B, **r**A
	subco. **r**D, **r**A, **r**B	**subfco.** **r**D, **r**B, **r**A
Subtract Immediate (**r**A – value)	**subi** **r**D, **r**A, value	**addi** **r**D, **r**A, –value
Subtract Immediate Shifted (**r**A – value || $16$0)	**subis** **r**D, **r**A, value	**addis** **r**D, **r**A, –value
Subtract Immediate Carrying (**r**A – value)	**subic** **r**D, **r**A, value	**addic** **r**D, **r**A, –value
Subtract Immediate Carrying and Record (**r**A – value)	**subic.** **r**D, **r**A, value	**addic.** **r**D, **r**A, –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** **r**D, **r**A	**tlbre** **r**D, **r**A, 0
Read TLBLO Portion of TLB Entry	**tlbrelo** **r**D, **r**A	**tlbre** **r**D, **r**A, 1
Write TLBHI Portion of TLB Entry	**tlbwehi** **r**D, **r**A	**tlbwe** **r**D, **r**A, 0
Write TLBLO Portion of TLB Entry	**tlbwelo** **r**D, **r**A	**tlbwe** **r**D, **r**A, 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	**nop**	**ori** 0, 0, 0

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	**li** **r**D, SIMM	**addi** **r**D, 0, SIMM
Load Immediate Shifted	**lis** **r**D, SIMM	**addis** **r**D, 0, SIMM

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 **r**A and d in the address syntax d(**r**A).

Table C-23: **Simplified Mnemonic for Load Address**

Operation	Simplified Mnemonic	Equivalent Mnemonic
Load Address	**la** **r**D, d(**r**A)	**addi** **r**D, **r**A, d
	la **r**D, *variable*	**addi** **r**D, **r**A, d (**r**A, d 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	**mr** **r**A, **r**S	**or** **r**A, **r**S, **r**S
	mr. **r**A, **r**S	**or.** **r**A, **r**S, **r**S

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

XILINX®

Appendix D

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 838 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 838 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 840 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 **r**3 and the data is loaded into **r**4.

```
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 **r**3, the new data is stored from **r**4, and the old data is loaded into **r**5.

```
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 **r**0, the address of the memory word to be incremented is in **r**3, the increment value is contained in **r**4, and the data to be incremented is loaded into **r**5.

```
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 **r**0, the address of the memory word to be ANDed is in **r**3, the AND value is contained in **r**4, and the data to be ANDed is loaded into **r**5.

```
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 **r**3, the new (nonzero) data is stored from **r**4, and the old data is loaded into **r**5.

```
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 **r**3, the compare value is contained in **r**4, the new data is stored from **r**5, and the old data is loaded into **r**6.

```
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 **r**4 and **r**6 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 **r**3). 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 838). This atomically updates the old value of the lock with the new value (1) contained in **r**4. The old value is returned in **r**5 (not shown in the following example). CR0[EQ] is updated by test_and_set to indicate whether the value returned in **r**5 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 **r**3, the new element address is in **r**4, 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
loopl: 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-   loopl   #(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 **r**2 and **r**3 (and **r**4 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 **r**6. For immediate shifts, the shift amount is assumed to be greater than zero. GPRs **r**0 and **r**31 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 ($CR_{28:31}$). 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.

```
cror 2, 28, 29
cror 2, 2, 30
cror 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 370 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 511, 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

Notes:

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 446 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.

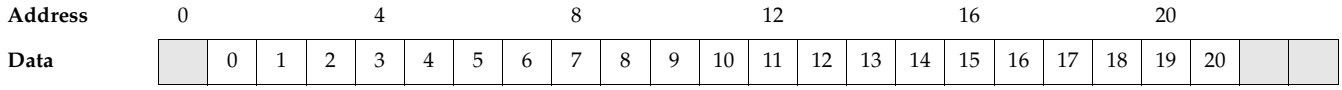


Figure D-1: **String Access Example**

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.

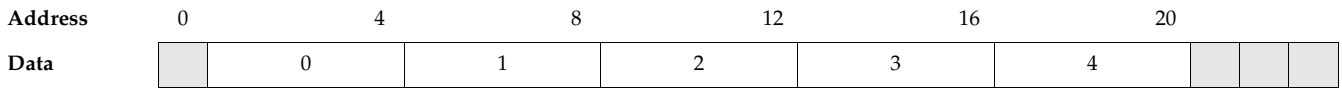


Figure D-2: **Multiple-Word Access Example**

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.

Appendix E

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*n*	Data address-compare registers	Debugging
DBCR*n*	Debug-control registers	
DBSR	Debug-status register	
DVC*n*	Data value-compare registers	
IAC*n*	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*n*	Hardware implementation registers	Processor configuration
DBAT*n*	Data BATs	Memory management
IBAT*n*	Instruction BATs	
SDR1	Page table base address	
SR*n*	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*n*	Monitor control registers	Performance monitoring
PMC*n*	Performance counters	
SIA	Sampled instruction address	
UMMCR*n*	Monitor control registers (user mode)	
UPMC*n*	Performance counters (user mode)	
USIA	Sampled instruction address (user mode)	
ICTC	Instruction cache throttling control register	Cache control
L2CR	L2 cache control register	
THRM*n*	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** **tlbre** **tlbsx[.]** **tlbsync** **tlbwe**	**tlbia** **tlbie** **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
dccci	Invalidates individual data-cache congruence classes.	
dcread	Data-cache debug function controlled by CCR0 register.	Data-cache debug function controlled by CDBCR register.
icbt	Instruction-cache block touch, executable from user mode.	Instruction-cache block touch, executable from privileged mode only.
iccci	Invalidates the entire instruction cache.	Invalidates individual instruction-cache congruence classes.
icread	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 851 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: 0x000*n_nnnn* or 0xFFF*n_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 decrementer 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 798, 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.

XILINX®

Appendix F

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*n*[1]	Data-cache normal victim register	Cache control
DTV*n*[1]	Data-cache transient victim register	
DVLIM[1]	Data-cache victim limit	
INV*n*[1]	Instruction-cache normal victim register	
ITV*n*[1]	Instruction-cache transient victim register	
IVLIM[1]	Instruction-cache victim limit	
DBCR2	Debug-control register 2	Debugging
DCDBTRH[1] DCDBTRL[1]	Data-cache debug tag registers	
ICDBTRH[1] ICDBTRL[1]	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: • 8-bit PID • 32-bit effective address	97 bits: • 1-bit instruction or data address-space (from the MSR) • 32-bit PID • 64-bit effective address
Page size	1KB to 16MB	1KB to 1TB (terabyte)
TLB instructions	**tlbia** **tlbre** **tlbsx[.]** **tlbsync** **tlbwe**	**tlbivax** **tlbre** **tlbsx[.]** **tlbsync** **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
dccci	This instruction is implementation dependent. On some PowerPC Book-E processors, this instruction invalidates the entire data cache.
dcread	This instruction is implementation dependent. On some PowerPC Book-E processors, the format of data returned by this instruction is different.
icbt	The opcode differs from the opcode recognized by PowerPC 40x processors.
iccci	This instruction is implementation dependent. On some PowerPC Book-E processors, this instruction invalidates the entire instruction cache.
icread	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 decrementer (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 decrementer interrupt.
- The timer-control register (TCR) controls different FIT and watchdog time-out intervals, and it controls the decrementer instead of the PIT.
- The timer-status register (TSR) describes decrementer 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 798, 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*n* registers. Debug status is reported by the DBSR register.

Index

E

F

G

H

I

R

S

T

U

V

W

X

Y

Z

Volume 2(b): PPC405 Processor Block Manual

Virtex-II Pro™ Platform FPGA Documentation

March 2002 Release

XILINX®

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

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

Virtex-II Pro Advance Product Specification

March 2002 Release

Revision History

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

Date	Version	Revision
02/07/02	1.0	Initial Xilinx release.

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 2-1 lists the general notational conventions used throughout this document.

Table 2-1: **General Notational Conventions**

Convention	Definition
mnemonic	Instruction mnemonics are shown in lower-case bold.
variable	Variable items are shown in italic.
$\overline{\text{ActiveLow}}$	An overbar indicates an active-low signal.

Table 2-1: **General Notational Conventions *(Continued)***

Convention	Definition
n	A decimal number.
0x*n*	A hexadecimal number.
0b*n*	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).
$OBJECT_{b,b,\ldots}$	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 2-2 lists the PPC405x3 registers used in this document and their descriptive names.

Table 2-2: **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.

assert	As applied to signals, this term indicates a signal is driven to its active state.
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.
Book-E	An version of the PowerPC architecture designed specifically for embedded applications.
cache block	Synonym for *cache line*.
cache line	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.
cache set	Synonym for *congruence class*.
clear	To write a bit value of 0.
clock	Unless otherwise specified, this term refers to the PPC405x3 processor clock.
congruence class	A collection of cache lines with the same index.
cycle	The time between two successive rising edges of the associated clock.
dead cycle	A cycle in which no useful activity occurs on the associated interface.
deassert	As applied to signals, this term indicates a signal is driven to its inactive state.
dirty	An indication that cache information is more recent than the copy in memory.
doubleword	Eight bytes, or 64 bits.
effective address	The untranslated memory address as seen by a program.
exception	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 *interrupt* to occur.
fill buffer	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.
flush	A cache operation that involves writing back a modified entry to memory, followed by an invalidation of the entry.
GB	Gigabyte, or one-billion bytes.
halfword	Two bytes, or 16 bits.
hit	An indication that requested information exists in the accessed cache array, the associated fill buffer, or on the corresponding OCM interface.

inactive	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 *active-high* signal drives a logic 0 when inactive. An *active-low* signal drives a logic 1 when inactive.
interrupt	The process of stopping the currently executing program so that an exception can be handled.
invalidate	A cache or TLB operation that causes an entry to be marked as invalid. An invalid entry can be subsequently replaced.
KB	Kilobyte, or one-thousand bytes.
line buffer	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.
line fill	A transfer of the contents of the instruction or data line buffer into the appropriate cache.
line transfer	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).
little endian	A memory byte ordering where the address of an item corresponds to the least-significant byte.
logical address	Synonym for *effective address*.
MB	Megabyte, or one-million bytes.
memory	Collectively, cache memory and system memory.
miss	An indication that requested information does not exist in the accessed cache array, the associated fill buffer, or on the corresponding OCM interface.
OEA	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.
on chip	In system-on-chip implementations, this indicates on the same FPGA chip as the processor core, but external to the processor core.
pending	As applied to interrupts, this indicates that an exception occurred, but the interrupt is disabled. The interrupt occurs when it is later enabled.
physical address	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.
PLB	Processor local bus.
privileged mode	The operating mode typically used by system software. Privileged operations are allowed and software can access all registers and memory.

problem state	Synonym for *user mode*.
process	A program (or portion of a program) and any data required for the program to run.
real address	Synonym for *physical address*.
scalar	Individual data objects and instructions. Scalars are of arbitrary size.
set	To write a bit value of 1.
sleep	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.
sticky	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.
string	A sequence of consecutive bytes.
supervisor state	Synonym for *privileged mode*.
system memory	Physical memory installed in a computer system external to the processor core, such RAM, ROM, and flash.
tag	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.
UISA	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.
user mode	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.
VEA	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.
virtual address	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.
wake up	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.
word	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*

XILINX®

Chapter 1

Introduction to the PowerPC® 405 Processor

The PPC405x3 is a 32-bit implementation of the *PowerPC™ embedded-environment architecture* that is derived from the PowerPC architecture. Specifically, the PPC405x3 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 PPC405x3 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)
• 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 PPC405x3 are compatible with other PowerPC implementations. Privileged software generally is not compatible. The migration of privileged software from the PowerPC architecture to the PPC405x3 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 PPC405x3.

PowerPC Embedded-Environment Architecture

The PPC405x3 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	• 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

Table 1-2: **OEA Features of the PowerPC Embedded-Environment Architecture *(Continued)***

Operating Environment	Features
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 decrementer (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

PPC405x3 Software Features

The PPC405x3 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 PPC405x3 VEA and OEA support is also available in implementations of the PowerPC Book-E architecture. Key software features of the PPC405x3 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 PPC405x3. 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 PPC405x3 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 PPC405x3 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 PPC405x3 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

PPC405x3 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 PPC405x3 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 PPC405x3. Descriptions of the registers are in the following sections.

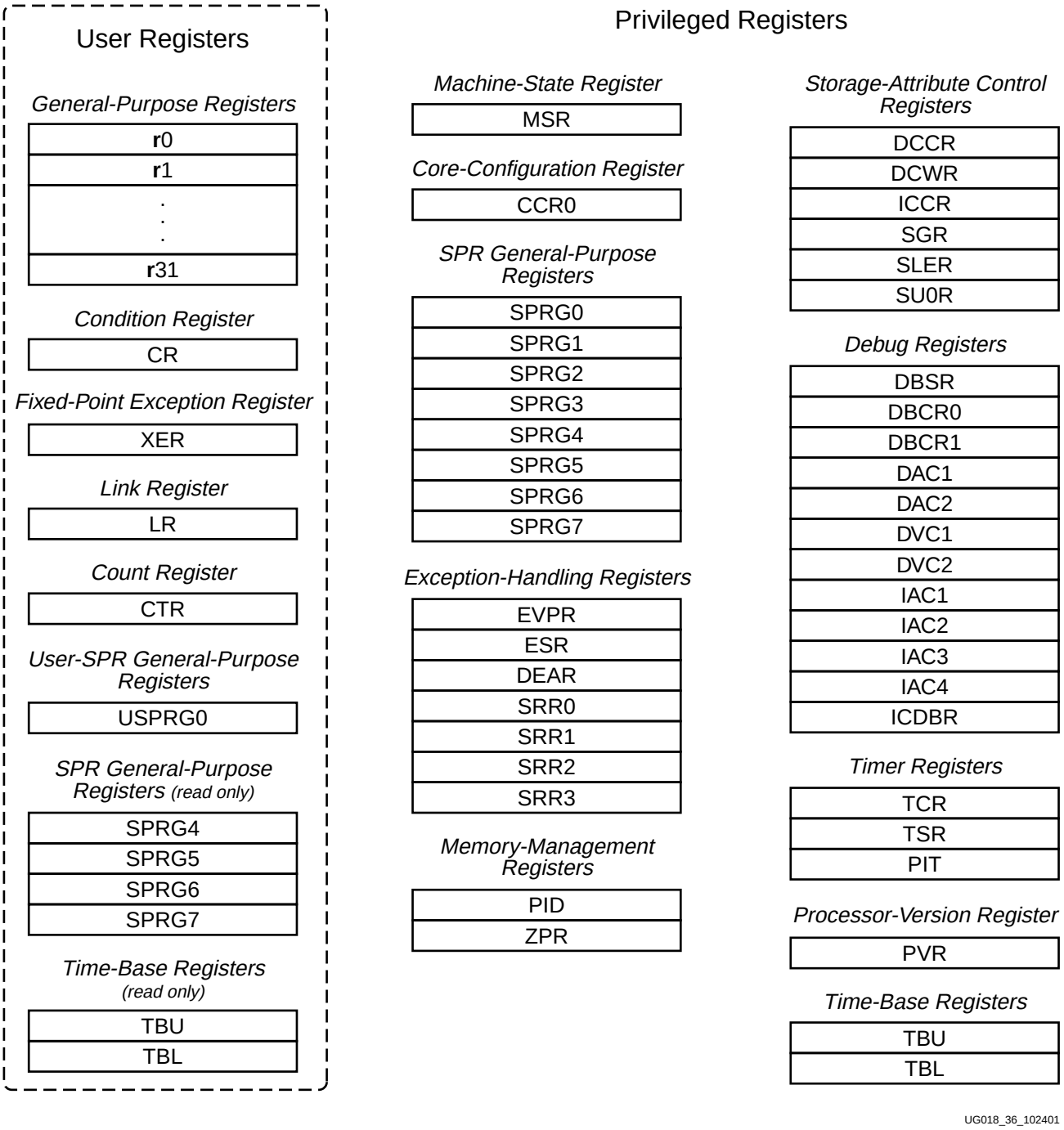


Figure 1-1: PPC405x3 **Registers**

General-Purpose Registers

The processor contains thirty-two 32-bit *general-purpose registers* (GPRs), identified as **r**0 through **r**31. 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 PPC405x3 processor. The OCM controllers are examples of devices that contain DCRs. Although the DCRs are not part of the PPC405x3 implementation, they are accessed using the **mtdcr** and **mfdcr** instructions. The DCRs can be accessed only by privileged software.

PPC405x3 Hardware Organization

As shown in Figure 1-2, the PPC405x3 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.

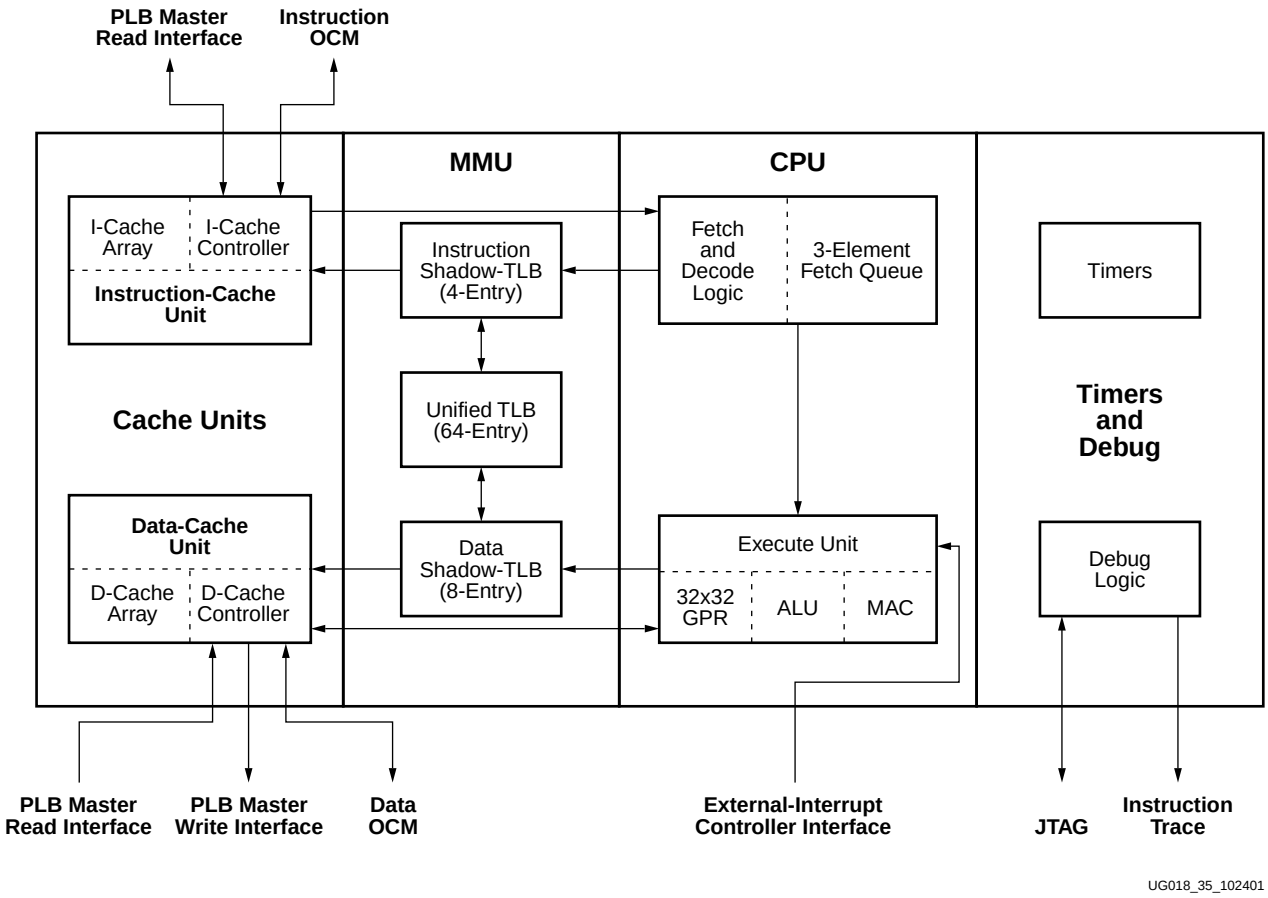


Figure 1-2: PPC405x3 **Organization**

Central-Processing Unit

The PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 to handle critical interrupts independently of noncritical interrupts.

Memory Management Unit

The PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 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.

PPC405x3 Interfaces

The PPC405x3 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.

PPC405x3 Performance

The PPC405x3 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 PPC405x3.

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: **PPC405x3 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: **PPC405x3 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

XILINX®

Chapter 2

Input/Output Interfaces

The processor block (PPC405x3, 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 897
- **CPU Control Interface**, page 901
- **Reset Interface**, page 903
- **Instruction-Side Processor Local Bus Interface**, page 907
- **Data-Side Processor Local Bus Interface**, page 929
- **Device-Control Register Interface**, page 958
- **External Interrupt Controller Interface**, page 968
- **JTAG Interface**, page 970
- **Debug Interface**, page 975
- **Trace Interface**, page 978

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 l/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:

PREFIX1PREFIX2SIGNAME1[SIGNAME1][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.
- SIGNAME1 is an uppercase name identifying the primary function of the signal.
- SIGNAME1 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(1)
EIC	External interrupt controller	Outside
ISOCM	Instruction-side on-chip memory (ISOCM)	Outside(1)
JTG	JTAG	Inside
PLB	Processor local bus	Inside
RST	Reset	Inside
TIE	TIE (signal tied statically to GND or V_{DD})	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 PPC405x3 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 PPC405x3. 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 PPC405x3 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 PPC405x3 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 PPC405x3 clock, CPMC405CLOCK. Instead of using the global-local enables to prevent the clock signal from propagating through a zone, CPM logic can stop the PPC405x3 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 PPC405x3 from sleep mode when any of the following occurs:

- A timer interrupt or timer reset is asserted by the PPC405x3.
- A chip-reset or system-reset request is asserted (this request comes from a source other than the PPC405x3).
- An external interrupt or critical interrupt input is asserted and the corresponding interrupt is enabled by the appropriate machine-state register (MSR) bit.
- The DBGC405DEBUGHALT chip-input signal (if provided) is asserted. Assertion of this signal indicates that an external debug tool wants to control the PPC405x3 processor. See page 976 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.

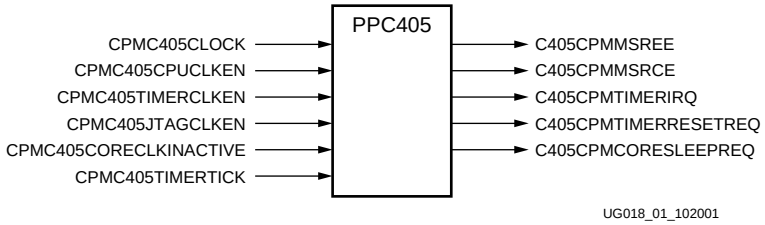


Figure 2-1: **CPM Interface Block Symbol**

Table 2-2: **CPM Interface I/O Signals**

Signal	I/O Type	If Unused	Function
CPMC405CLOCK	I	Required	PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 register (JDSR). An external debug tool (such as RISCWatch) can read this register and determine that the PPC405x3 is in sleep mode. This signal should be asserted by the CPM when it places the PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3. It is also used to report the detection of a machine check condition within the PPC405x3.

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.

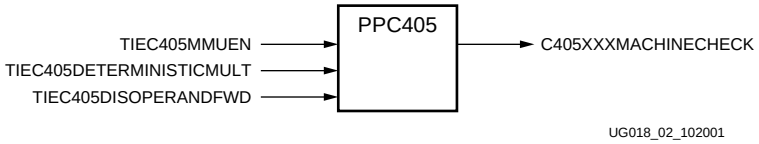


Figure 2-2: **CPU Control Interface Block Symbol**

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 PPC405x3.

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 PPC405x3 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 PPC405x3.

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 × Halfword (32-bit result)	1	2
Halfword × Word (48-bit result)	2	3
Word × Word (64-bit result)	4	5

Notes:

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 PPC405x3.

C405XXXMACHINECHECK (output)

When asserted, this signal indicates the PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 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, RSTC405RESETCHIP, 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[1]
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] following reset	Previous DBSR[MRR]	0b01	0b10	0b11	0b11

Notes:
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.

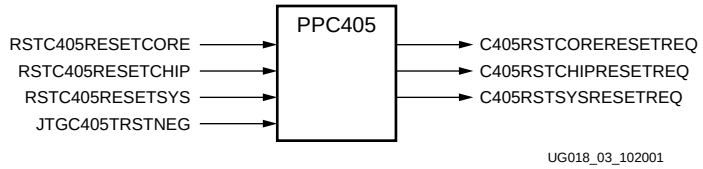


Figure 2-3: **Reset Interface Block Symbol**

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 PPC405x3 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 PPC405x3 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 PPC405x3 core logic, data cache, instruction cache, and the on-chip memory controller (OCM). The PPC405x3 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 904 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 PPC405x3 core logic, data cache, instruction cache, and the OCM). The signal does not reset logic in the processor block. The PPC405x3 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 PPC405x3 does not record a chip reset type in DBSR[MRR] when this signal is deasserted.

Table 2-5, page 904 shows the valid combinations of the RSTC405RESETCORE, RSTC405RESETCHIP, 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 PPC405x3 core logic, data cache, instruction cache, and the OCM). This signal resets the logic in the PPC405x3 JTAG unit, but it does not reset any other processor block logic. The PPC405x3 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, RSTC405RESETCHIP, 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 PPC405x3 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 904 shows the valid combinations of the RSTC405RESETCORE, RSTC405RESETCHIP, 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 PPC405x3 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 PPC405x3 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 911).
- The target address of the instruction to be fetched is specified by the address bus, C405PLBICUABUS[0:29] (page 911). 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 912). 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 912). Cacheable transfers are always performed with an eight-word transfer size.
- The user-defined storage attribute is indicated by C405PLBICUU0ATTR (page 912).
- The request priority is indicated by C405PLBICUPRIORITY[0:1] (page 913). 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 913). This can occur when a branch instruction is executed or when an interrupt occurs.

Fetched 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 913).
- Instructions sent from the PLB slave to the ICU during a line transfer are indicated as valid using PLBC405ICURDDACK (page 914).
- The PLB-slave bus width, or size (32-bit or 64-bit), is specified by PLBC405ICUSSIZE1 (page 914). 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 915). 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 915).

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.

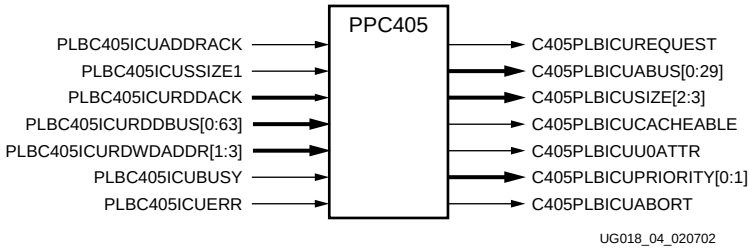


Figure 2-4: **Instruction-Side PLB Interface Block Symbol**

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.
C405PLBICUSIZE[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.
PLBC405ICUSSIZE1	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 _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 982 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 PPC405x3 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 PLBC405ICUSSIZE1 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.

PLBC405ICUSSIZE1 (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 915). 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 916 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 PLBC405ICUSSIZE1.
- The line-transfer size specified by C405PLBICUSIZE[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.

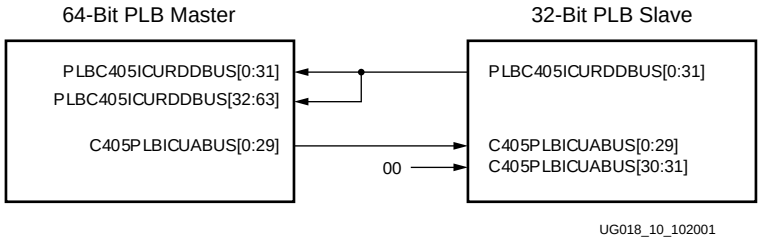


Figure 2-5: **Attachment of ISPLB Between 32-Bit Slave and 64-Bit Master**

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 915.

Table 2-10: **Contents of ICU Read-Data Bus During Line Transfer**

PLB-Slave Size	Line-Transfer Size	Transfer Order[1]	ICU Read-Data Bus [0:31][2]	ICU Read-Data Bus [32:63][2]
32-Bit	Four Words	x00	Instruction 0	*Instruction 0*
		x01	*Instruction 1*	Instruction 1
		x10	Instruction 2	*Instruction 2*
		x11	*Instruction 3*	Instruction 3
	Eight Words	000	Instruction 0	*Instruction 0*
		001	*Instruction 1*	Instruction 1
		010	Instruction 2	*Instruction 2*
		011	*Instruction 3*	Instruction 3
		100	Instruction 4	*Instruction 4*
		101	*Instruction 5*	Instruction 5
		110	Instruction 6	*Instruction 6*
		111	*Instruction 7*	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:
1. "x" indicates a don't-care value in PLBC405ICURDWDADDR[1:3].
2. 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 PPC405x3 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 PLBC405ICUSSIZE1 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[1]	Description	Where Used	
rl#	Fetch-request identifier	Request Request acknowledge Read-data acknowledge	(C405PLBICUREQUEST) (PLBC405ICUADDRACK) (PLBC405ICURDDACK)
adr#	Fetch-request address	Request address	(C405PLBICUABUS[0:29])
d#$_{\#}$	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 PPC405x3 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 ICU read-data bus ICU forward (bypass)	(PLBC405ICURDDACK) (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 (rl1) 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 (rl2) 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).

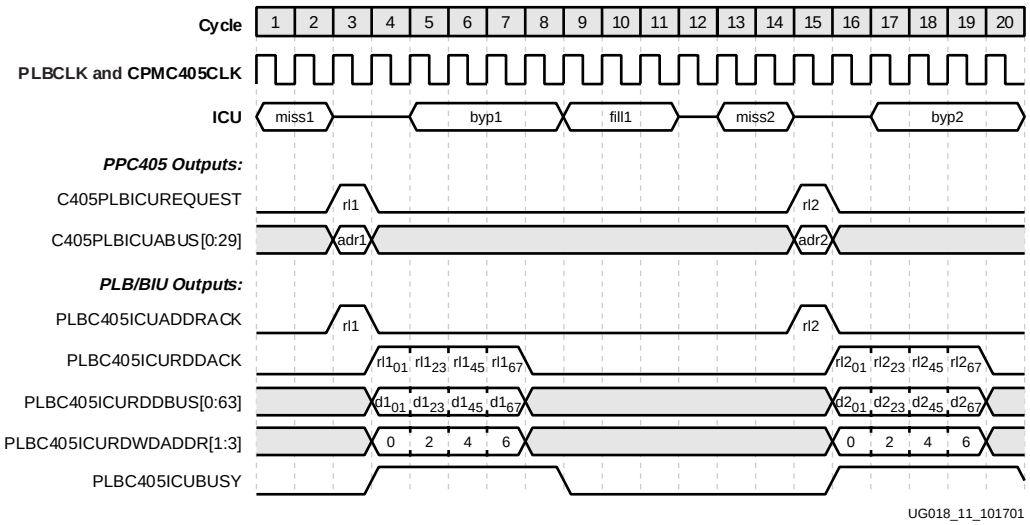


Figure 2-6: **ISPLB Non-Pipelined Cacheable Sequential Fetch (Case 1)**

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 (rl1) 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 (rl2) 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).

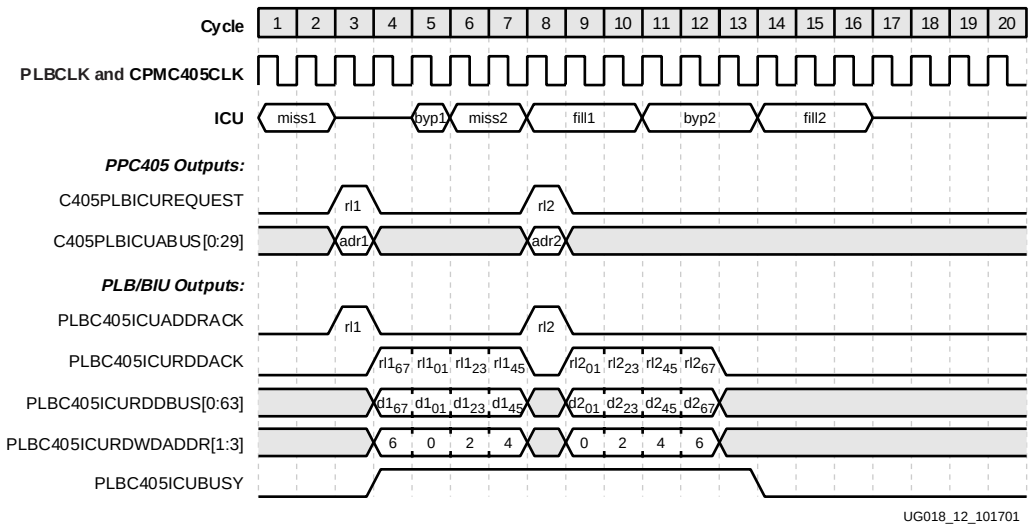


Figure 2-7: **ISPLB Non-Pipelined Cacheable Sequential Fetch (Case 2)**

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 (rl1) 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 (rl2) 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.

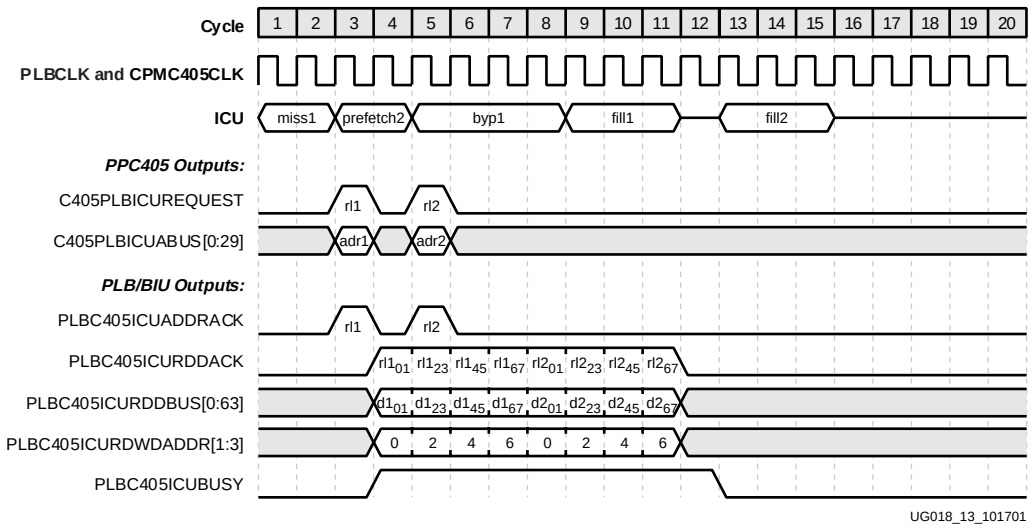


Figure 2-8: **ISPLB Pipelined Cacheable Sequential Fetch (Case 1)**

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 (rl1) 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 (rl2) 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).

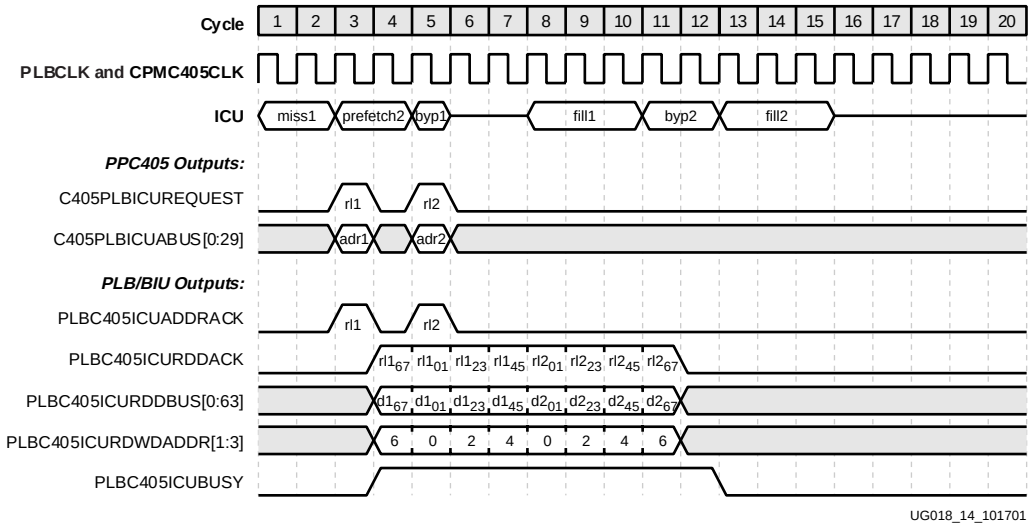


Figure 2-9: **ISPLB Pipelined Cacheable Sequential Fetch (Case 2)**

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 (rl1) 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 (rl2) 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.

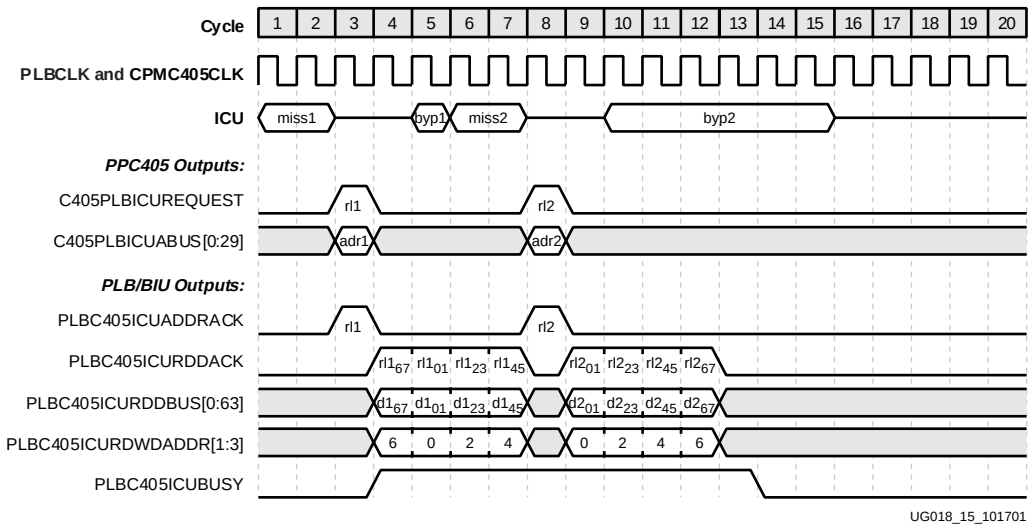


Figure 2-10: **ISPLB Non-Pipelined Non-Cacheable Sequential Fetch**

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 (rl1) 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 (rl2) 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.

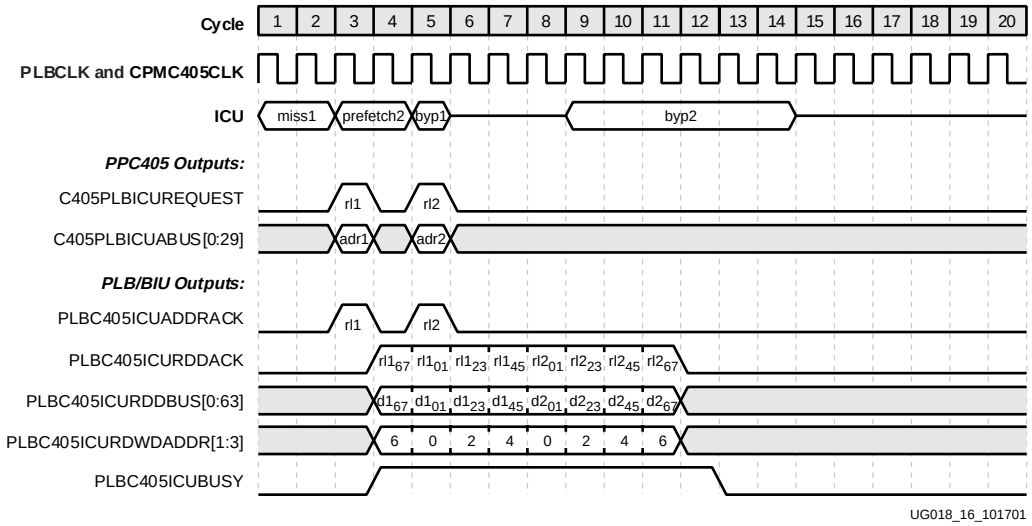


Figure 2-11: **ISPLB Pipelined Non-Cacheable Sequential Fetch**

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 PPC405x3 clock.

The line read (rl1) is requested by the ICU in PLB cycle 2, which corresponds to PPC405x3 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 (PPC405x3 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).

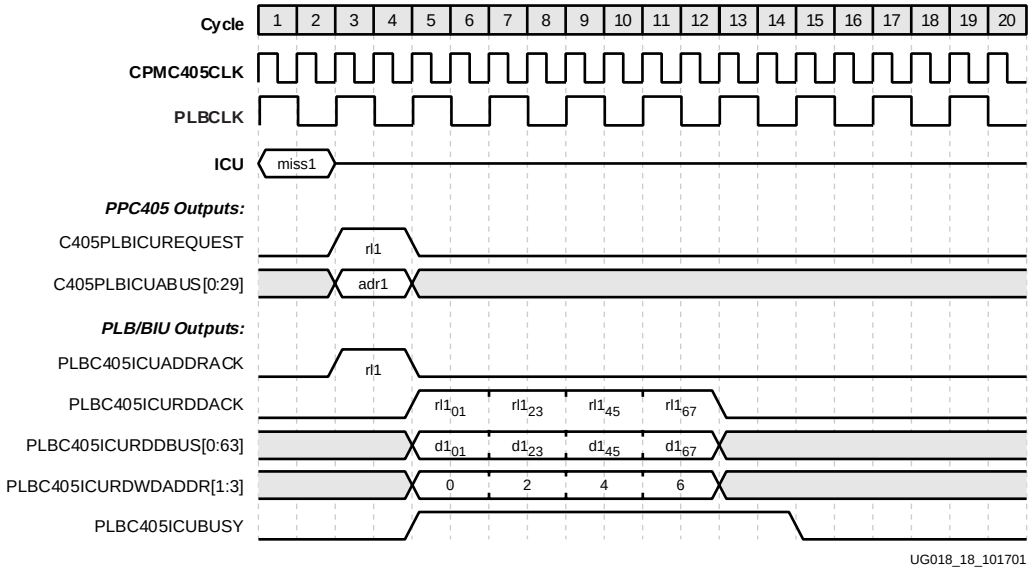


Figure 2-12: **ISPLB 2:1 Core-to-PLB Line Fetch**

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 PPC405x3 clock.

The line read (rl1) is requested by the ICU in PLB cycle 2, which corresponds to PPC405x3 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 (PPC405x3 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).

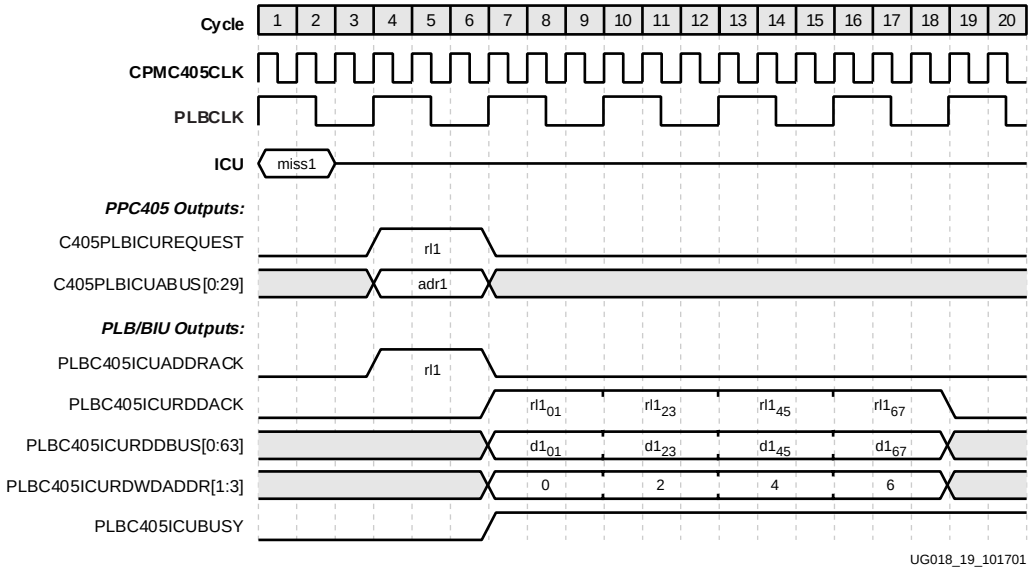


Figure 2-13: **ISPLB 3:1 Core-to-PLB Line Fetch**

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 (rl1) 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 (rl2) 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.

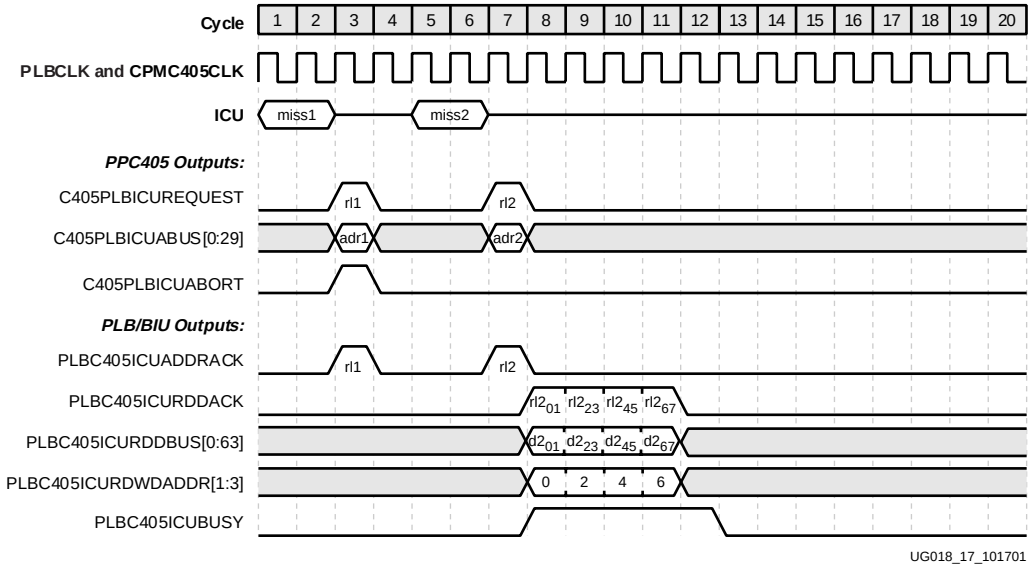


Figure 2-14: **ISPLB Aborted Fetch Request**

Data-Side Processor Local Bus Interface

The data-side processor local bus (DSPLB) interface enables the PPC405x3 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 PPC405x3 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 933).
- The type of request (read or write) is indicated by C405PLBDCURNW (page 934).
- The target address of the data to be accessed is specified by the address bus, C405PLBDCUABUS[0:31] (page 934).
- The transfer size is specified as a single word or as eight words (cache line) using C405PLBDCUSIZE2 (page 934). 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 936). 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 934). Cacheable transfers are performed using word or line transfer sizes.
- The write-through storage attribute is indicated by C405PLBDCUWRITETHRU (page 935).
- The guarded storage attribute is indicated by C405PLBDCUGUARDED (page 935).
- The user-defined storage attribute is indicated by C405PLBDCUU0ATTR (page 935).
- The request priority is indicated by C405PLBDCUPRIORITY[0:1] (page 937). 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 937). 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 939).
- 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 941). 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 942).

- The PLB-slave bus width, or size (32-bit or 64-bit), is specified by PLBC405DCUSSIZE1 (page 940). 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 941). Data writes are transferred from the DCU to the PLB slave over the DCU write-data bus, PLBC405DCUWRDBUS[0:63] (page 938). 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 936). 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 941). 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 **dcbtst** 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.

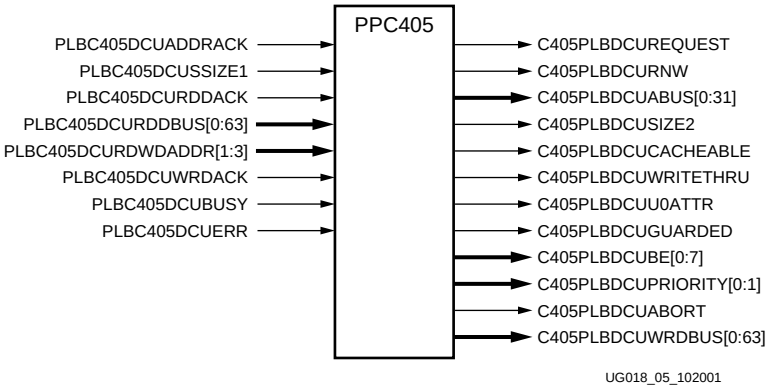


Figure 2-15: **Data-Side PLB Interface Block Symbol**

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.
C405PLBDCUSIZE2	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.
PLBC405DCUSSIZE1	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 _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 982 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 PPC405x3 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.

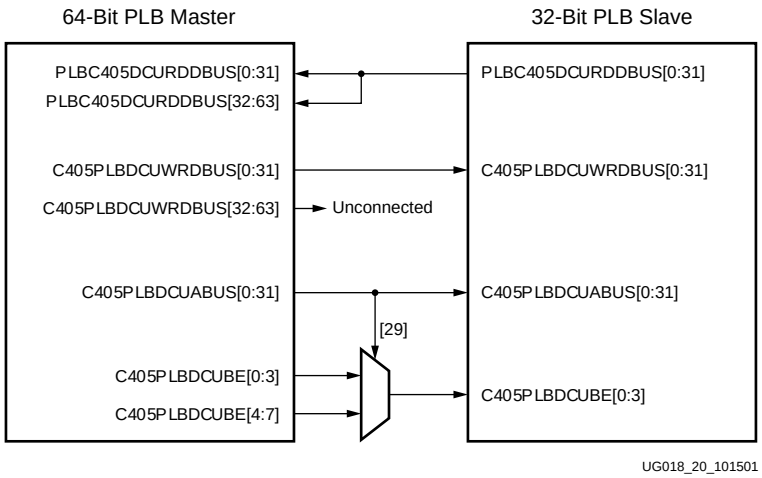


Figure 2-16: **Attachment of DSPLB Between 32-Bit Slave and 64-Bit Master**

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 PLBC405DCUSSIZE1 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).

PLBC405DCUSSIZE1 (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 936. 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 916, 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 939, 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 916, 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 939, 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 PLBC405DCUSSIZE1) 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 936). 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 936 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 936 is assumed.

Table 2-16: **Contents of DCU Read-Data Bus During Eight-Word Line Transfer**

PLB-Slave Size	Transfer Order[1]	DCU Read-Data Bus [0:31][2]	DCU Read-Data Bus [32:63][2]
32-Bit	000	Word 0	*Word 0*
	001	*Word 1*	Word 1
	010	Word 2	*Word 2*
	011	*Word 3*	Word 3
	100	Word 4	*Word 4*
	101	*Word 5*	Word 5
	110	Word 6	*Word 6*
	111	*Word 7*	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 PLBC405DCUSSIZE1 and the line-transfer size (specified by C405PLBDCUSIZE2). 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 PPC405x3 can queue up to two flush operations.
- The BIU size (bus width) is 64 bits, so PLBC405DCUSSIZE1 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[1]	Description	Where Used	
rl#, wl#	Eight-word line read-request or write-request identifier, respectively	Request Request acknowledge Read-data acknowledge Write-data acknowledge	(C405PLBDCUREQUEST) (PLBC405DCUADDRACK) (PLBC405DCURDDACK) (PLBC405DCUWRDACK)
rw#, ww#	Single word read-request or write-request identifier, respectively	Request Request acknowledge Read-data acknowledge Write-data acknowledge	(C405PLBDCUREQUEST) (PLBC405DCUADDRACK) (PLBC405DCURDDACK) (PLBC405DCUWRDACK)
adr#	Data-access request address	Request address	(C405PLBDCUABUS[0:31])
$d\#_{\#}$	A doubleword (eight data bytes) transferred as a result of an eight-word line transfer request	DCU read-data bus DCU write-data bus	(PLBC405DCURDDBUS[0:63]) (PLBC405DCUWRDBUS[0:63])
d#	A word (four data bytes) transferred as a result of a single word transfer request	DCU read-data bus DCU write-data bus	(PLBC405DCURDDBUS[0:63]) (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 DCU read-data bus Write-data acknowledge DCU write-data bus	(PLBC405DCURDDACK) (PLBC405DCURDDBUS[0:63]) (PLBC405DCUWRDACK) (PLBC405DCUWRDBUS[0:63])
#	Used to identify the order doublewords are sent to the DCU	Transfer order	(PLBC405DCURDWDADDR[1:3])

Notes:
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 (rl1) 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 (rl2) 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). 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 (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 responds to the rl3 request after it has completed all transactions associated with the second request (rl2). 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.

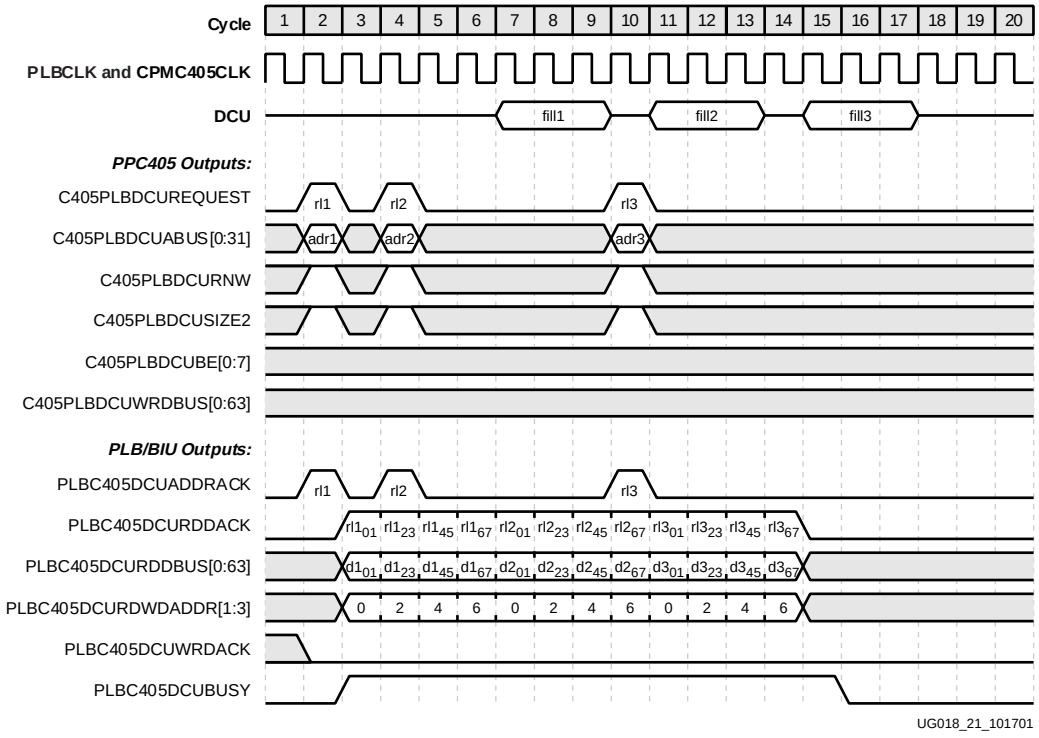


Figure 2-17: **DSPLB Three Consecutive Line Reads**

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.

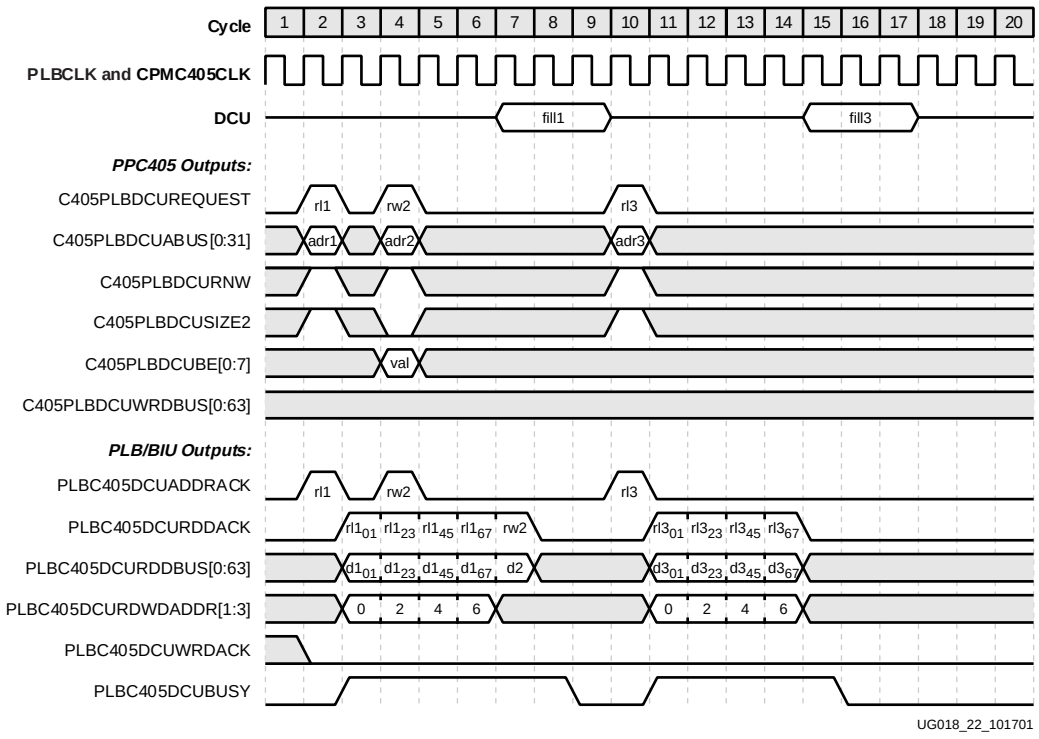


Figure 2-18: **DSPLB Line Read/Word Read/Line Read**

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 PPC405x3 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.

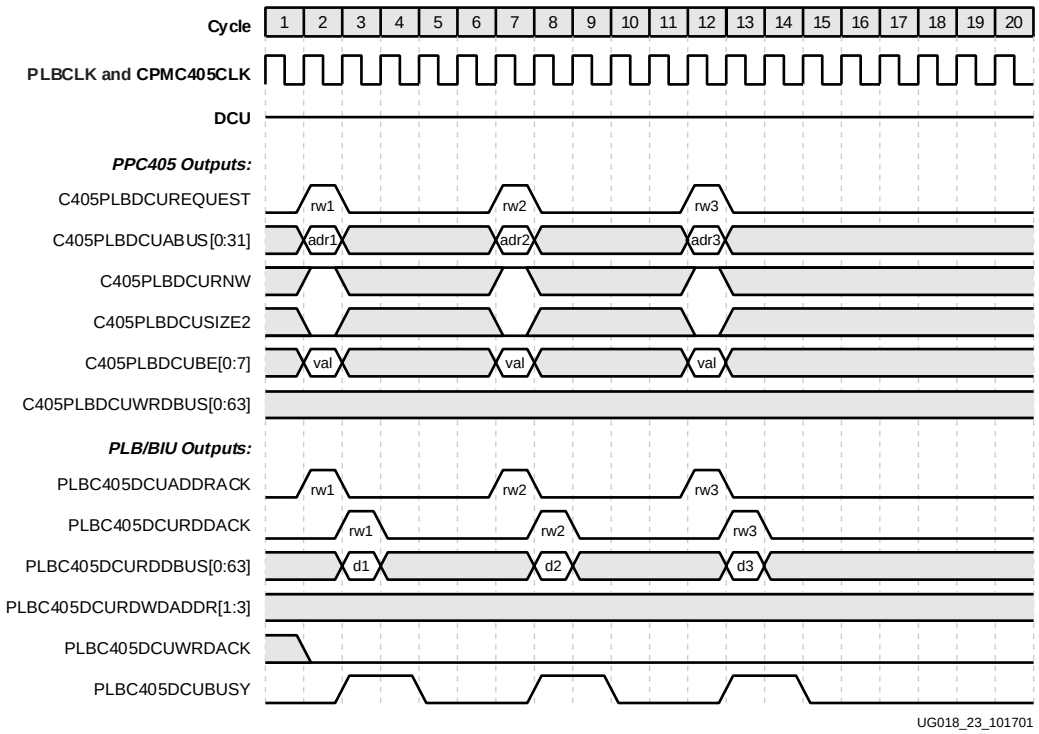


Figure 2-19: **DSPLB Three Consecutive Word Reads**

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.

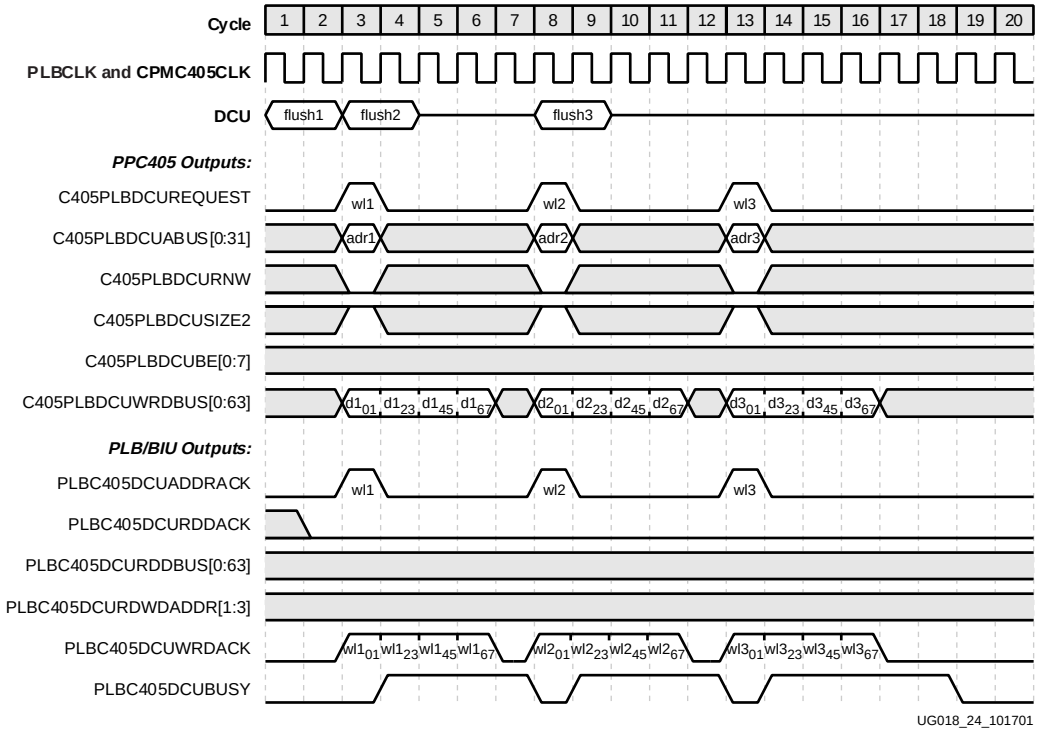


Figure 2-20: **DSPLB Three Consecutive Line Writes**

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.

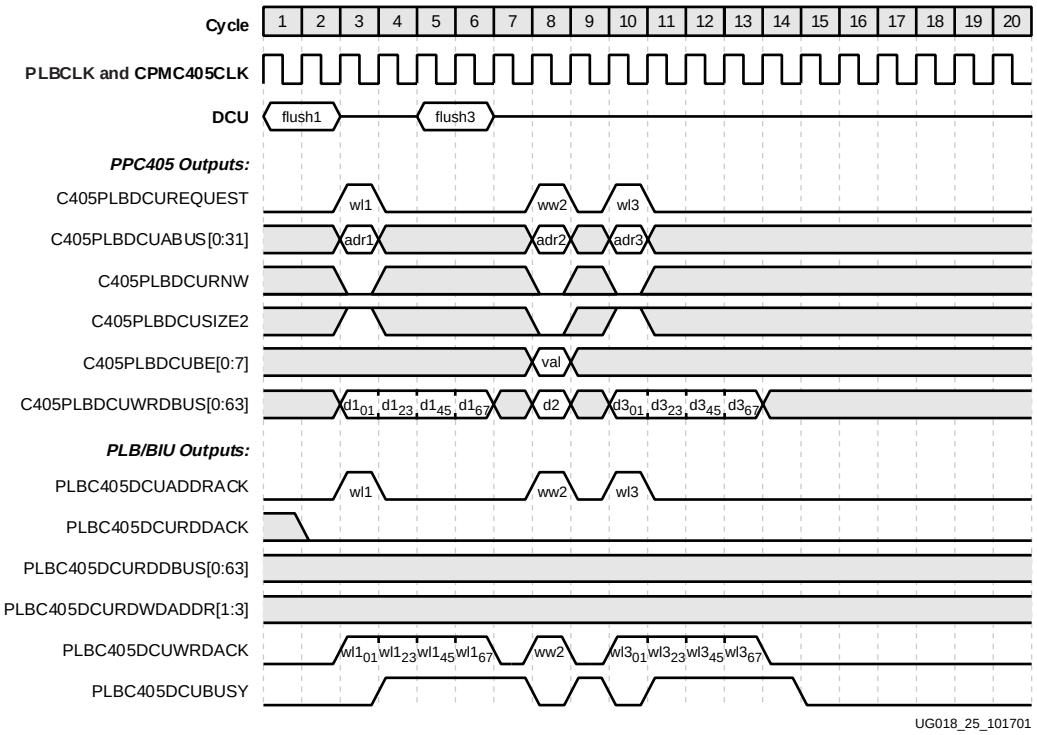


Figure 2-21: **DSPLB Line Write/Word Write/Line Write**

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.

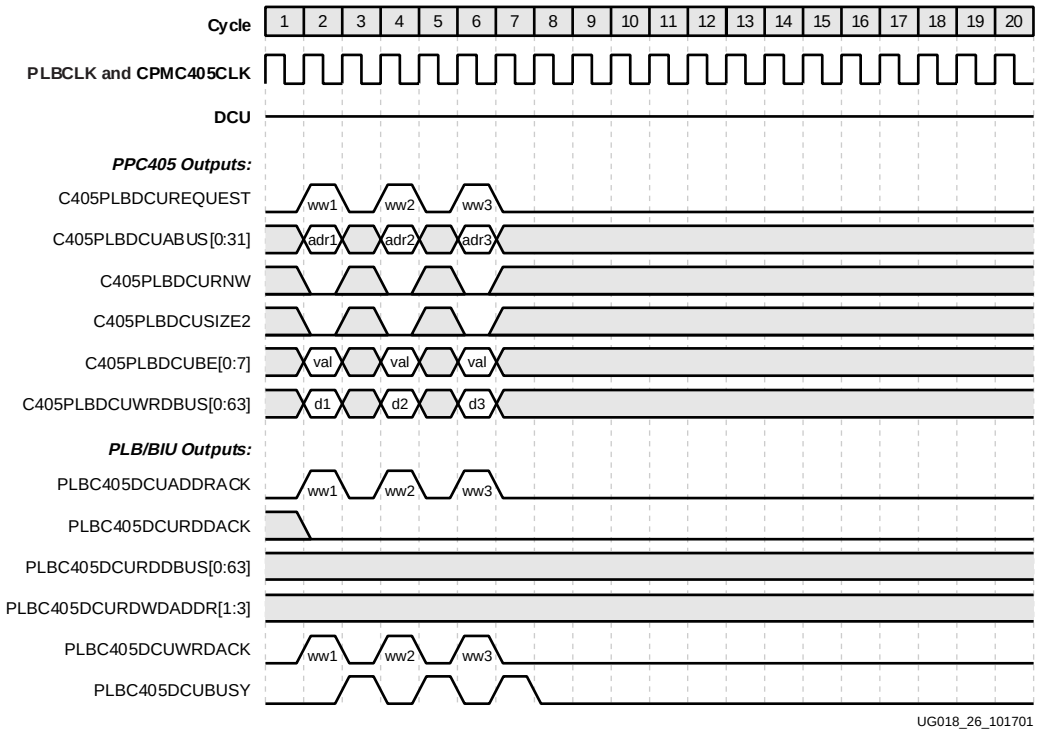


Figure 2-22: **DSPLB Three Consecutive Word Writes**

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).

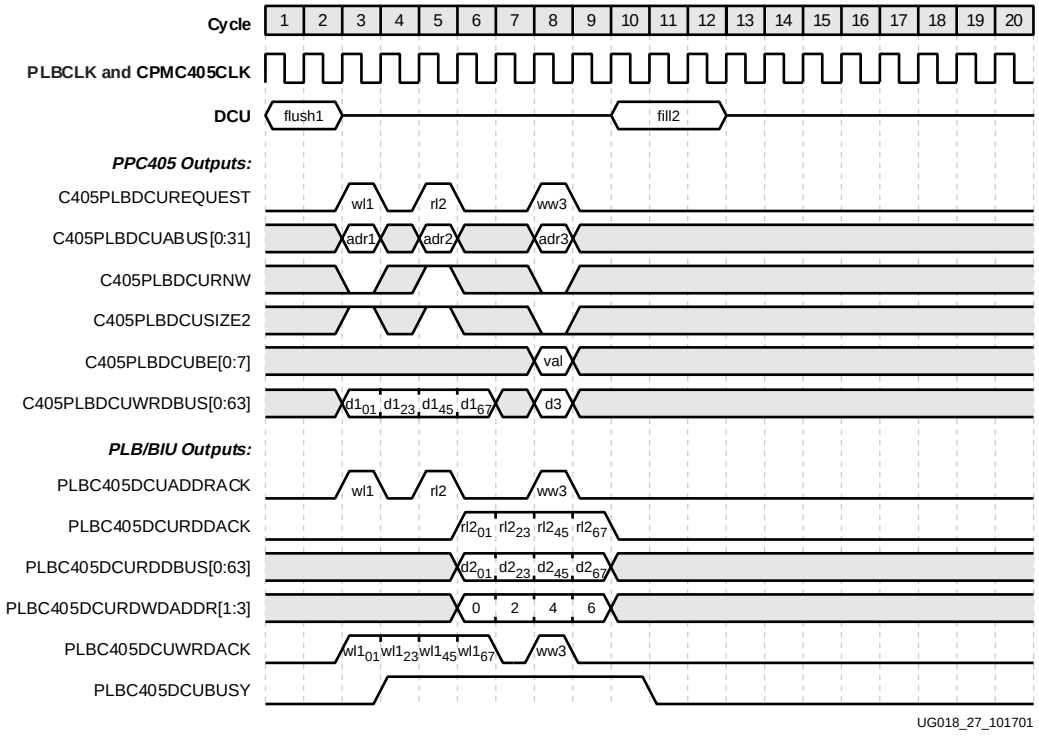


Figure 2-23: **DSPLB Line Write/Line Read/Word Write**

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.

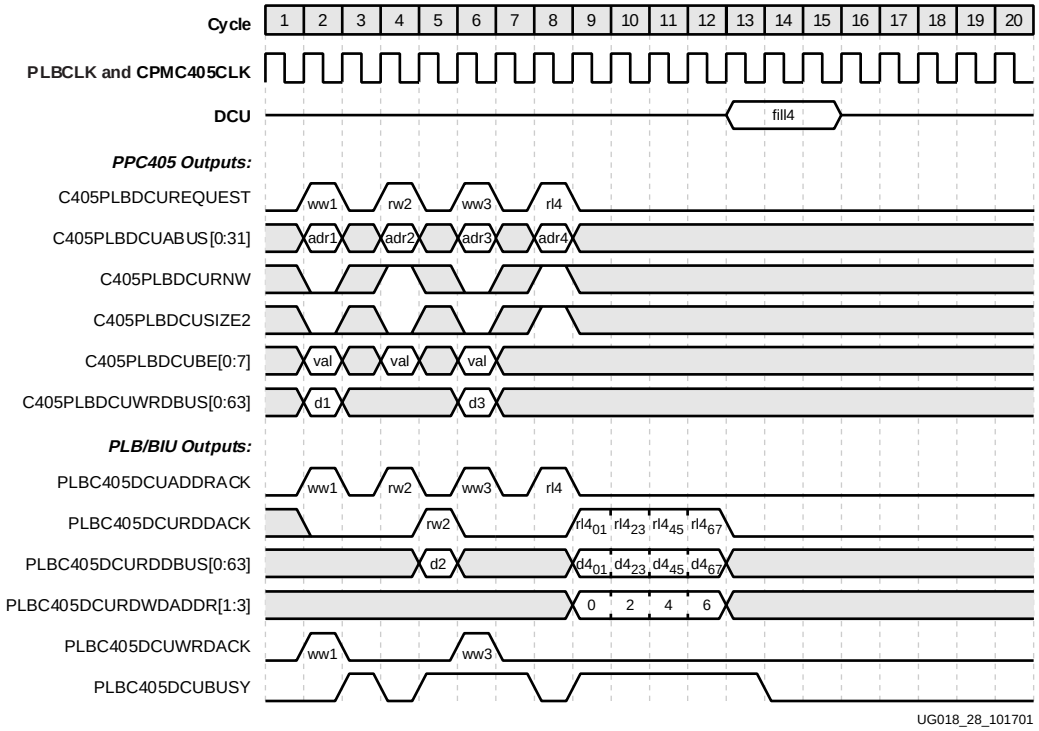


Figure 2-24: **DSPLB Word Write/Word Read/Word Write/Line Read**

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).

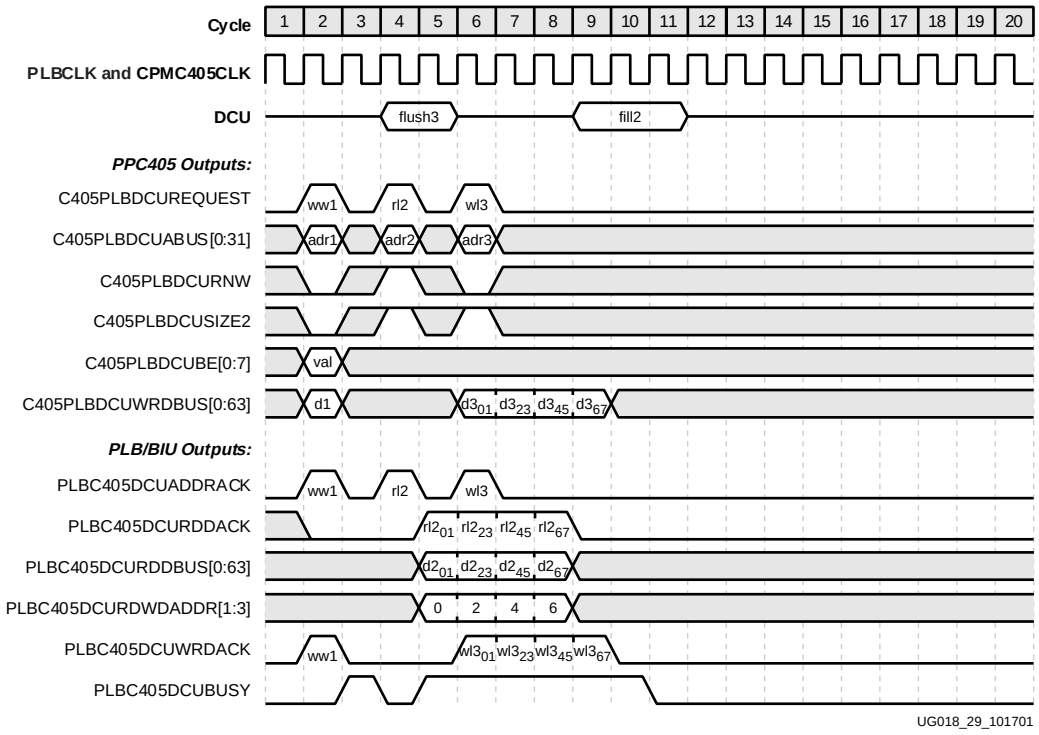


Figure 2-25: **DSPLB Word Write/Line Read/Line Write**

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 PPC405x3 clock.

The line read (rl1) is requested by the DCU in PLB cycle 2, which corresponds to PPC405x3 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 (PPC405x3 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 PPC405x3 cycles 13 through 15.

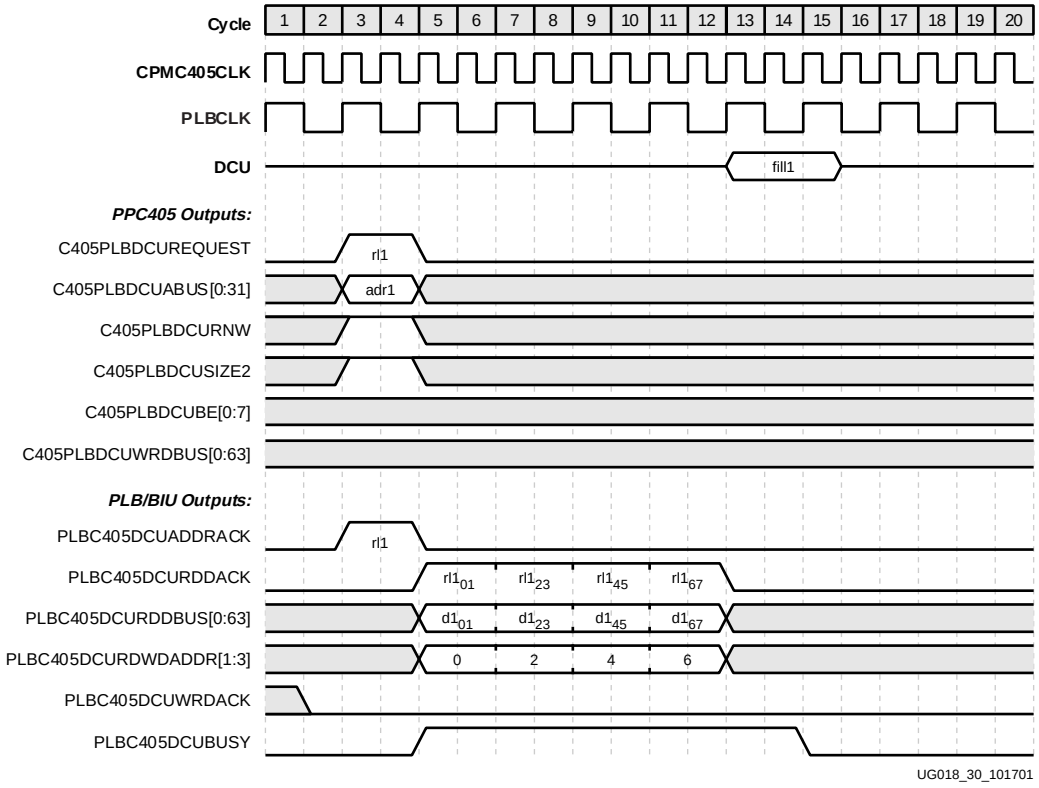


Figure 2-26: **DSPLB 2:1 Core-to-PLB Line Read**

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 PPC405x3 clock.

The line write (wl1) is requested by the DCU in PLB cycle 2, which corresponds to PPC405x3 cycle 4. The BIU responds in the same cycle. The request is made in response to a flush in PPC405x3 cycles 1 and 2 (flush1). Data is sent from the DCU to the BIU in PLB cycles 2 through 5 (PPC405x3 cycles 4 through 15).

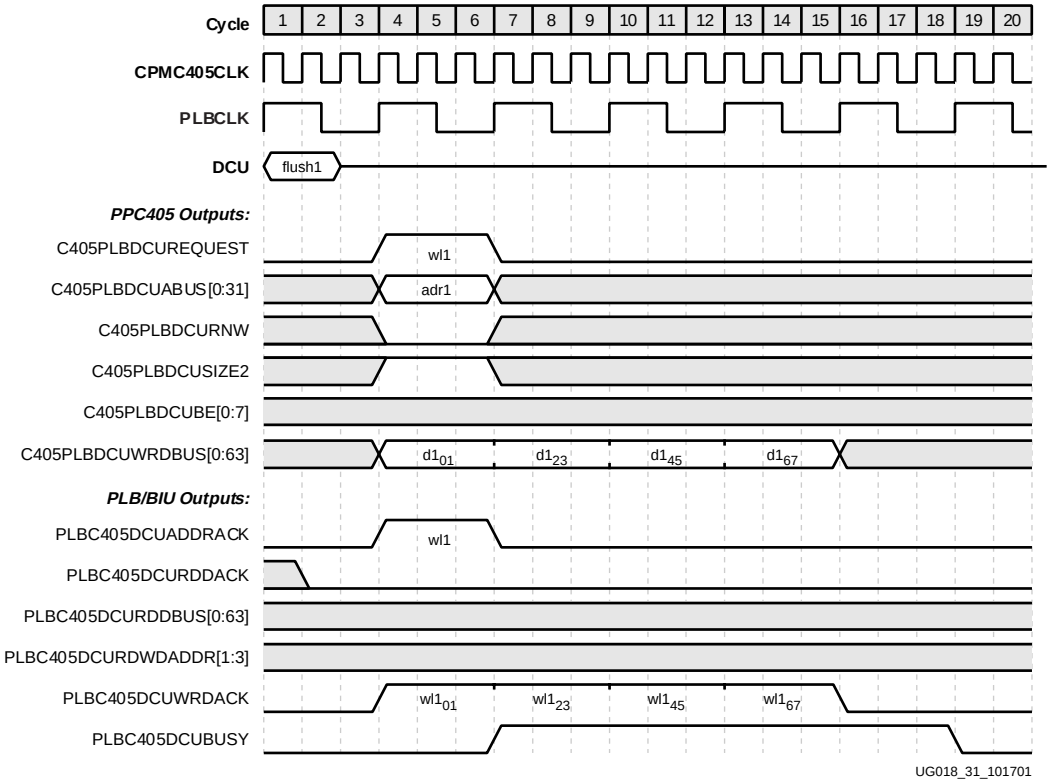


Figure 2-27: **DSPLB 3:1 Core-to-PLB Line Write**

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.

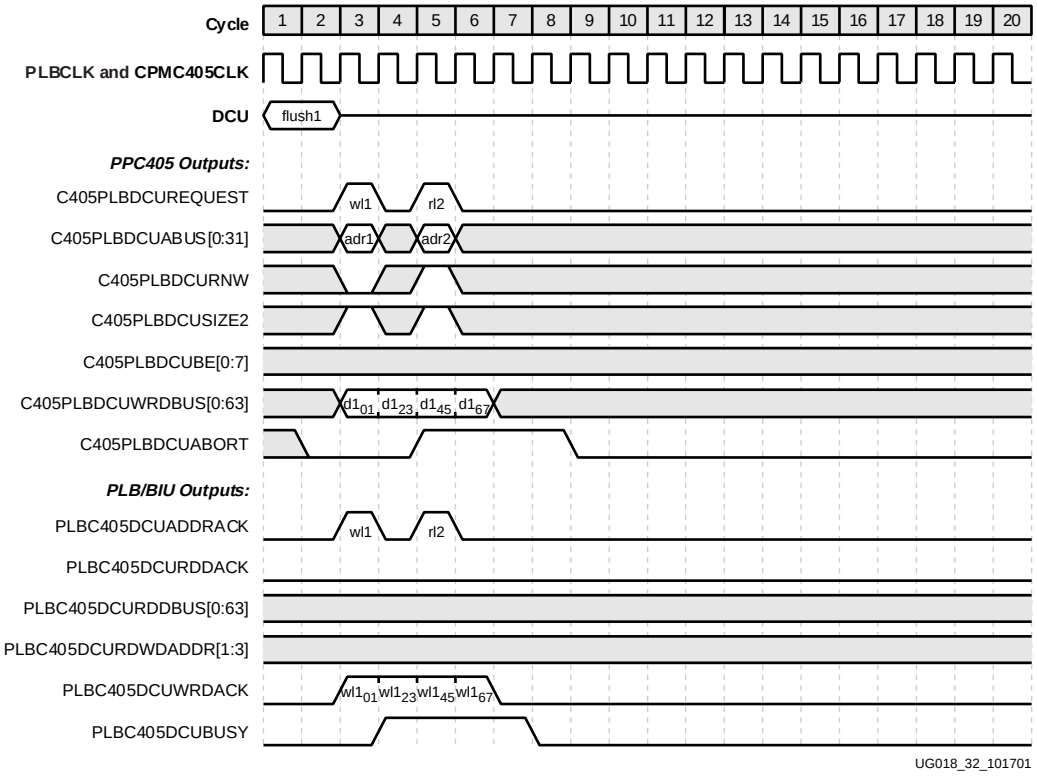


Figure 2-28: **DSPLB Aborted Data-Access Request**

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 959 shows an example DCR chain implementation in an FPGA chip. The acknowledge signal in this example is formed using combinatorial logic (OR gate).

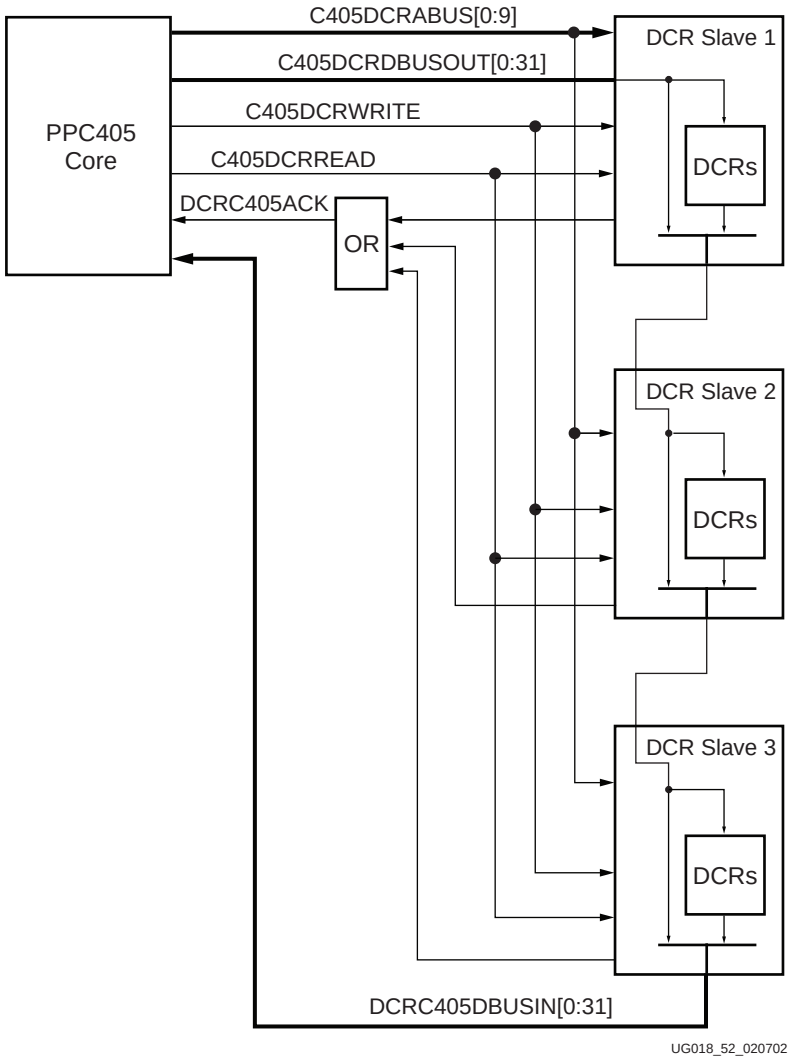


Figure 2-29: **DCR Chain Block Diagram**

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 PPC405x3. A DCR access takes at least three PPC405x3 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 PPC405x3 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 PPC405x3 must be derived from a common source. The common source frequency is multiplied or divided before being sent to the PPC405x3 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 PPC405x3 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 PPC405x3. 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.

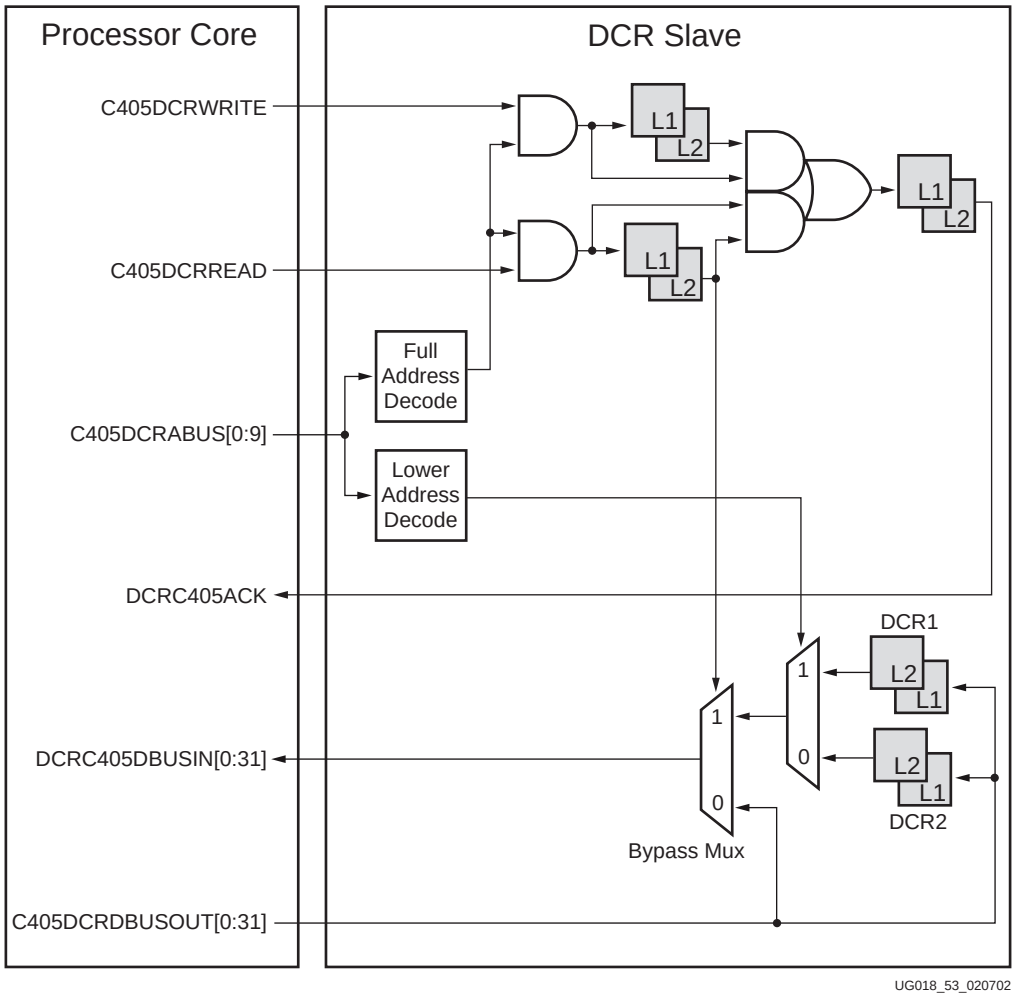


Figure 2-30: **DCR Bus Implementation**

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.

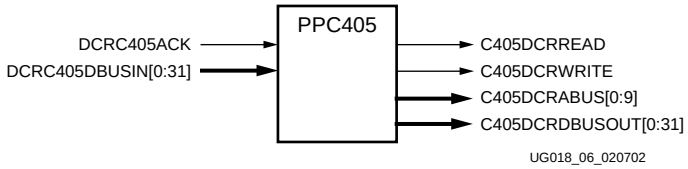


Figure 2-31: **DCR Interface Block Symbol**

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 (**mfdcr**). 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 (**mtdcr**). 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 **mtdcr** 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 **mtdcr** 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 **mtdcr** or **mfdcr** instruction is executed. Its contents are valid only when a DCR write-request or DCR read-request is asserted. When a **mtdcr** 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 PPC405x3 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 PPC405x3 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 PPC405x3 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 960.
- 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.

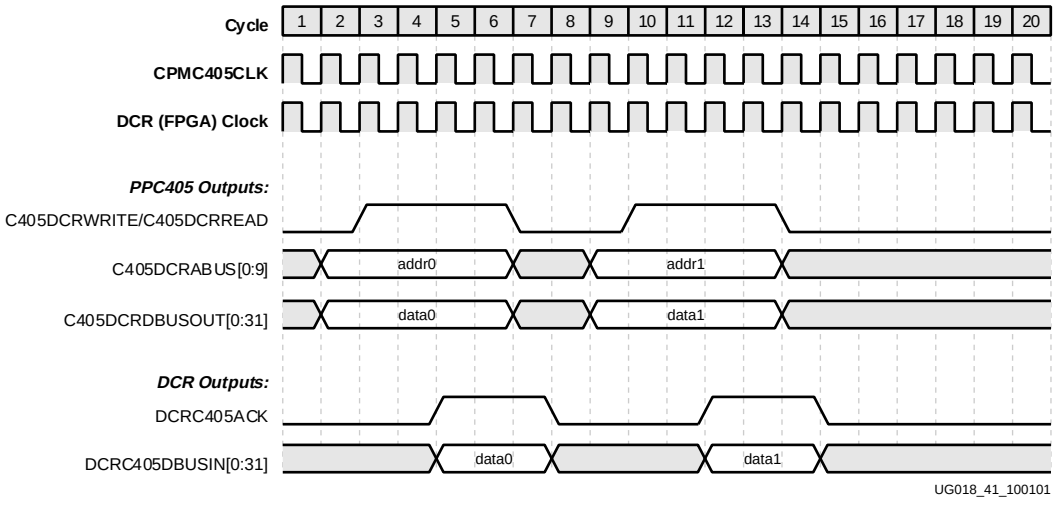


Figure 2-32: **DCR Interface Mode 0, 1:1 Clocking, Latched Acknowledge**

DCR Interface Mode 0, 1:1 Clocking, Combinatorial Acknowledge

The example in Figure 2-33 assumes the following:

- The PPC405x3 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 959.
- 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.

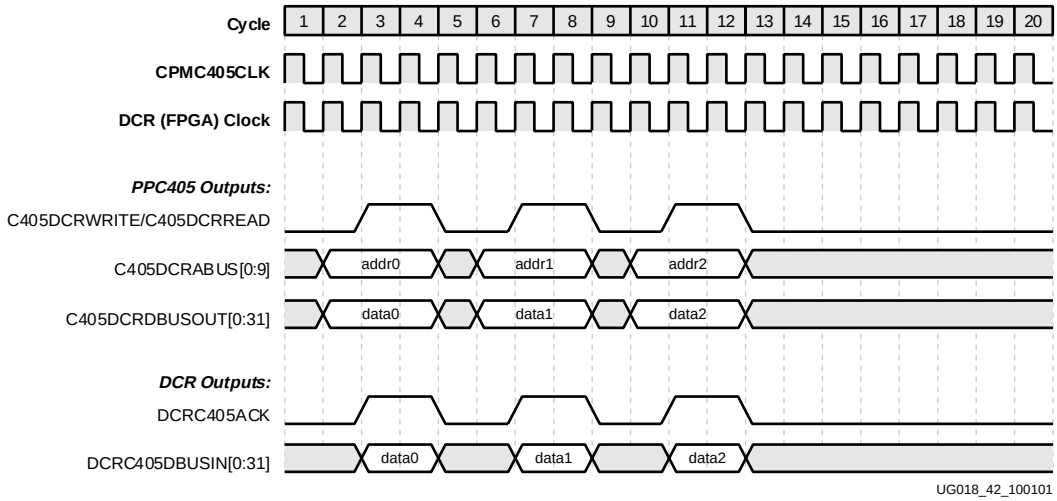


Figure 2-33: **DCR Interface Mode 0, 1:1 Clocking, Combinatorial Acknowledge**

DCR Interface Mode 0, 2:1 Clocking, Latched Acknowledge

The example in Figure 2-34 assumes the following:

- The PPC405x3 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 960.
- 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 PPC405x3 cycles. The implementation of the acknowledge signal causes it to be asserted two DCR cycles (four PPC405x3 cycles) after the access request. It is deasserted in the DCR cycle after the access request is deasserted.

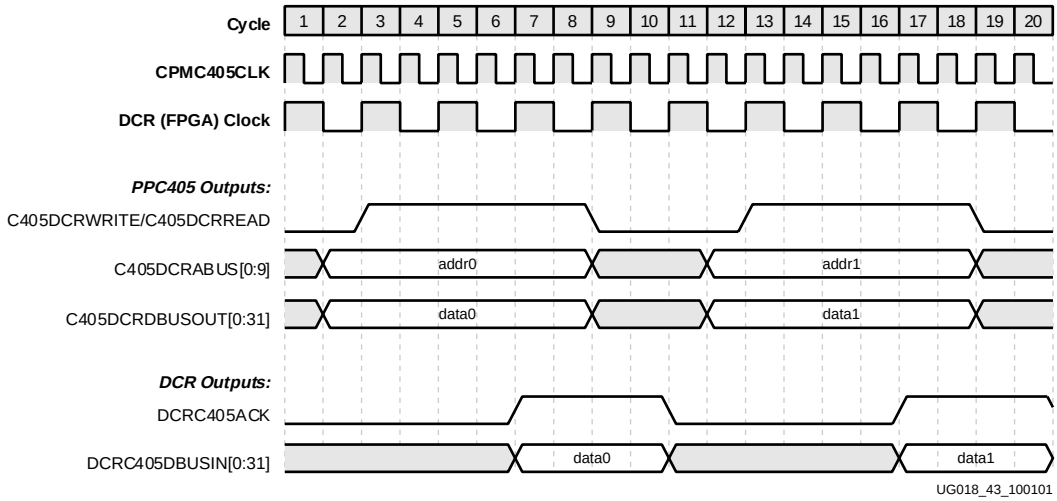


Figure 2-34: **DCR Interface Mode 0, 2:1 Clocking, Latched Acknowledge**

DCR Interface Mode 0, 1:2 Clocking, Latched Acknowledge

The example in Figure 2-35 assumes the following:

- The PPC405x3 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 960.
- 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 PPC405x3 (twelve DCR) cycles. The implementation of the acknowledge signal causes it to be asserted two DCR cycles (one PPC405x3 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 PPC405x3 and DCR clocks are not interlocked (the rising edges do not coincide) when the acknowledge is deasserted. The clocks interlock one-half a PPC405x3 cycle later, but the processor block must wait an entire additional cycle before it can present a new DCR address.

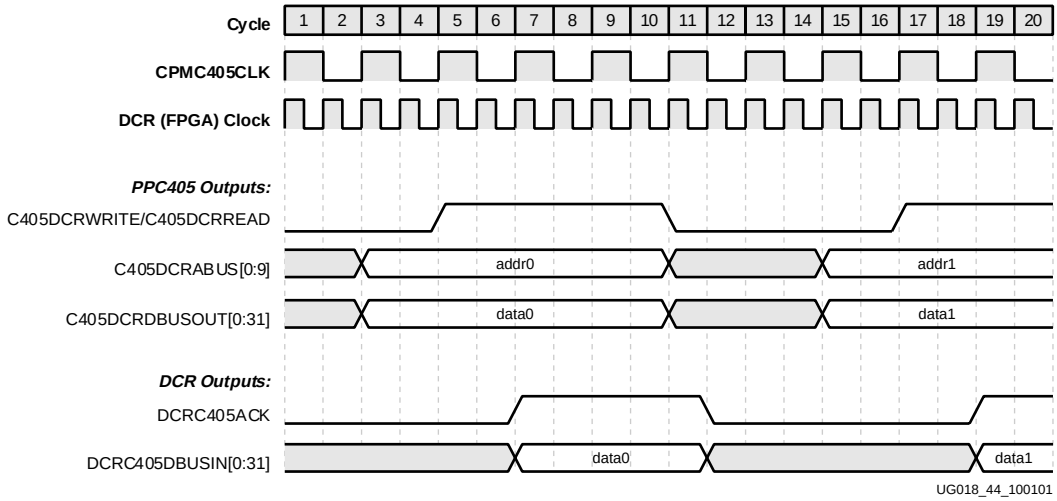


Figure 2-35: **DCR Interface Mode 0, 1:2 Clocking, Latched Acknowledge**

DCR Interface Mode 1, 1:1 Clocking, Combinatorial Acknowledge

The example in Figure 2-36 assumes the following:

- The PPC405x3 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 959.
- 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.

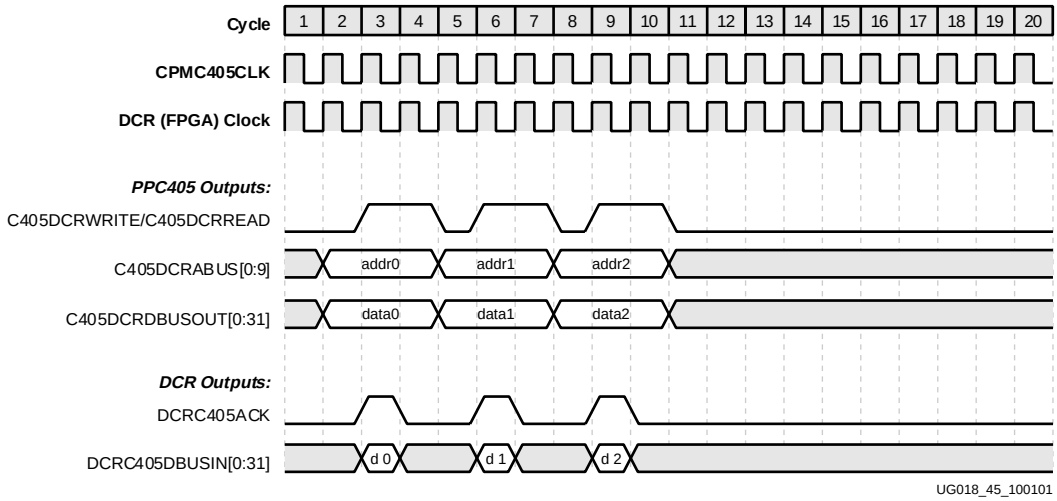


Figure 2-36: **DCR Interface Mode 1, 1:1 Clocking, Combinatorial Acknowledge**

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 897, 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.

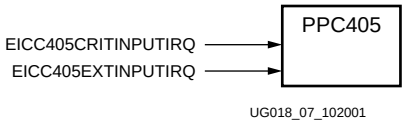


Figure 2-37: **EIC Interface Block Symbol**

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 is 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 PPC405x3 and facilitates portability of debugging tools.

The 7-bit instruction code supported by PPC405x3 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 PPC405x3 IR and the least significant three bits are in the second IR.

Table 2-20: **PPC405x3 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 0111_100 1001_100 1010_100 1011_100 1100_100 1101_100 1110_100	PPC405x3 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 976) 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.

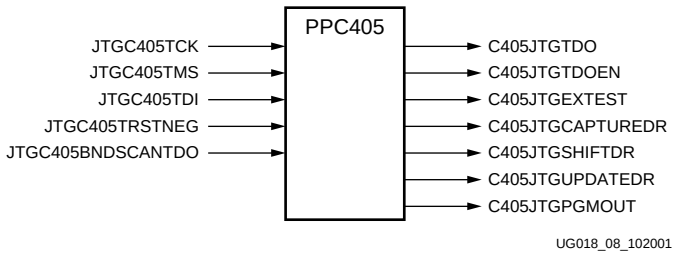


Figure 2-38: **JTAG Interface Block Symbol**

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.
C405JTGTDO	O	No Connect	JTAG TDO (test-data out).
C405JTGTDOEN	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.
C405JTGUPDATEDR	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 PPC405x3 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 PPC405x3 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 PPC405x3 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.

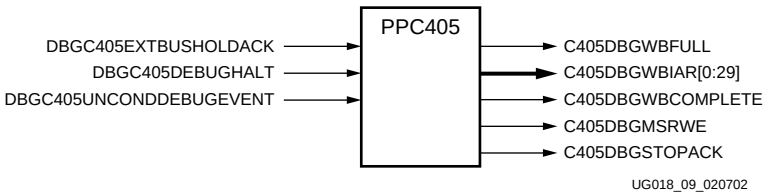


Figure 2-39: **Debug Interface Block Symbol**

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.
DBGC405UNCONDDEBUGEVENT	I	0	Indicates the external debug logic is causing an unconditional debug event.
C405DBGWBFULL	O	No Connect	Indicates the PPC405x3 writeback pipeline stage is full.
C405DBGWBIAR[0:29]	O	No Connect	The address of the current instruction in the PPC405x3 writeback pipeline stage.
C405DBGWBCOMPLETE	O	No Connect	Indicates the current instruction in the PPC405x3 writeback pipeline stage is completing.
C405DBGMSRWE	O	No Connect	Indicates the value of MSR[WE].
C405DBGSTOPACK	O	No Connect	Indicates the PPC405x3 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 PPC405x3 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.

DBGC405DEBUGHALT (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 PPC405x3.

DBGC405UNCONDDEBUGEVENT (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 PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 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.

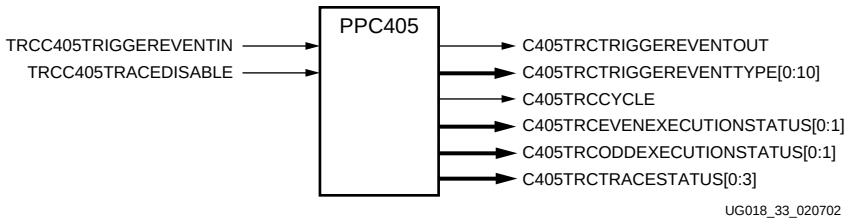


Figure 2-40: **Trace Interface Block Symbol**

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 PPC405x3 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 PPC405x3 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 PPC405x3 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 PPC405x3 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.

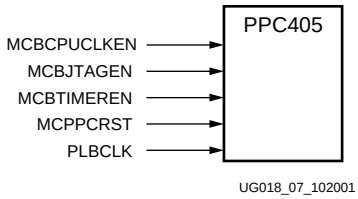


Figure 2-41: **FPGA-Specific Interface Block Symbol**

Table 2-25: **Additional FPGA I/O Signals**

Signal	I/O Type	If Unused	Function
MCBCPUCLKEN	I	1	Indicates the PPC405x3 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.

XILINX®

Chapter 3

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 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 hardware using second port. See the **Application Example** section on page 995.

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 Read Only and Write Only addresses to ISBRAM
- 8-bit DCR registers: ISCNTL, ISARC
- 32-bit DCR registers: ISINIT, ISFILL
- Two alternatives to write into ISBRAM: 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 992.

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 986, shows an example of a DSOCM-to-BRAM interface.

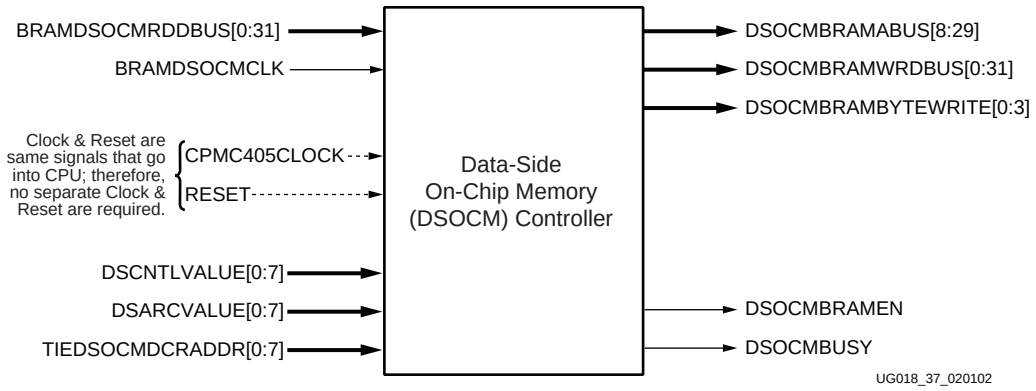


Figure 3-1: **DSOCM Block**

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.

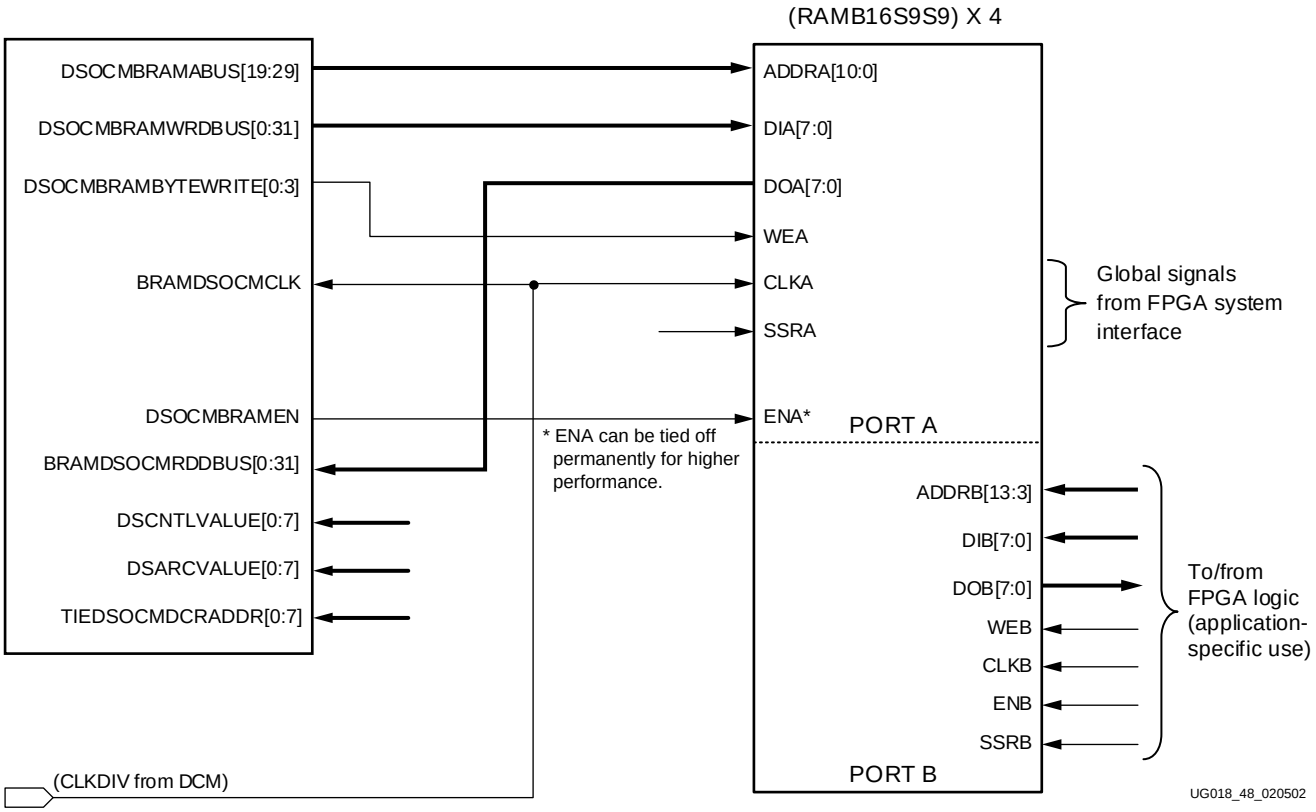


Figure 3-2: **DSOCM to BRAM Interface: 8K Example**

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 992, 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 992, 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 988, shows an example of an ISOCM-to-BRAM interface.

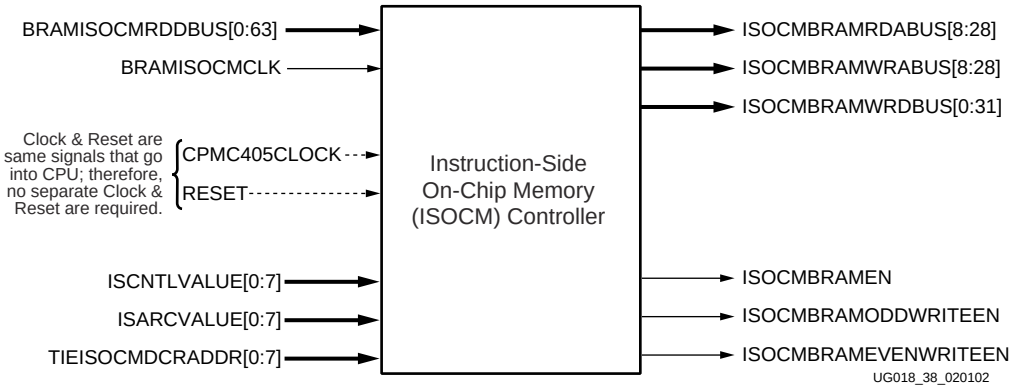


Figure 3-3: **ISOCM Block**

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.

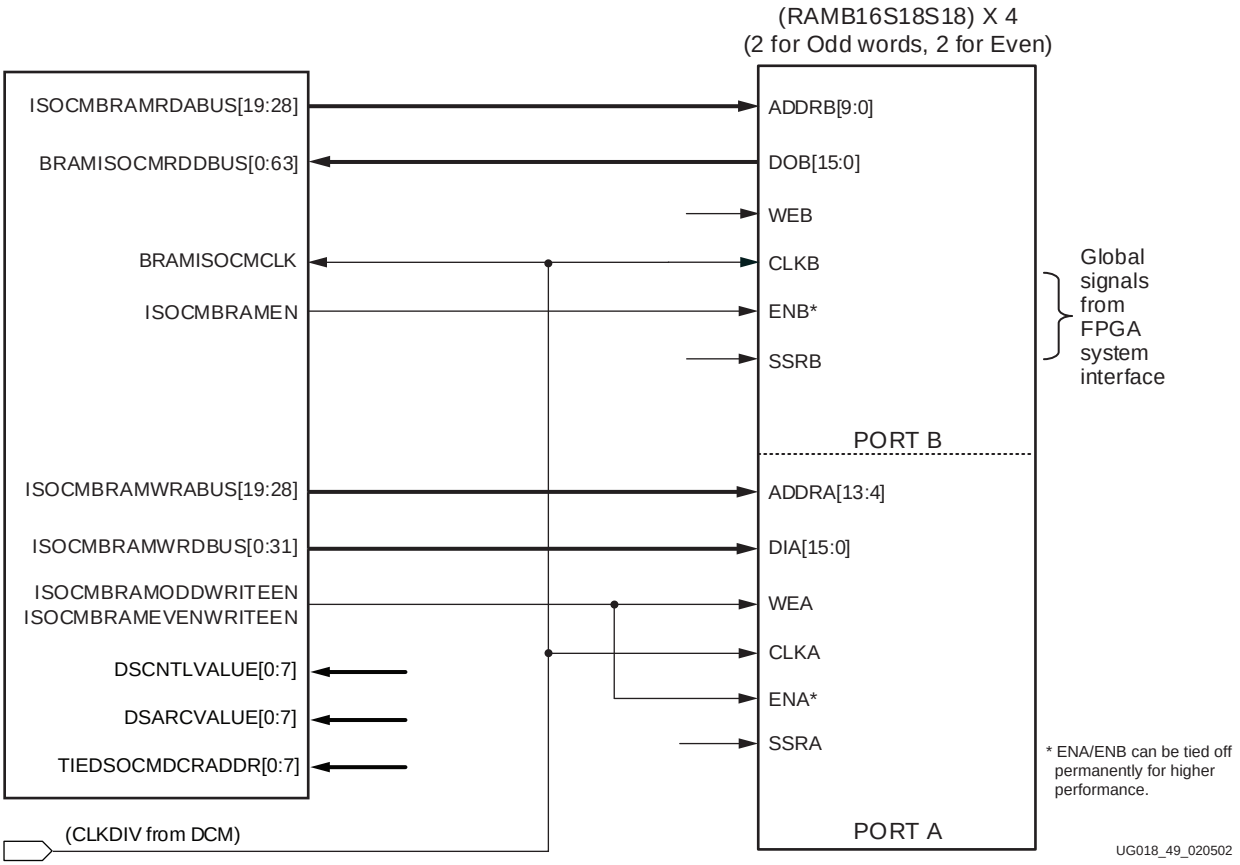


Figure 3-4: **ISOCM to BRAM Interface: 8K Example**

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 993, 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 993, 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 1100 = 0x02C
address of ISFILL = 00 0010 1101 = 0x02D
address of ISARC = 00 0010 1110 = 0x02E
address of ISCNTL = 00 0010 1111 = 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.

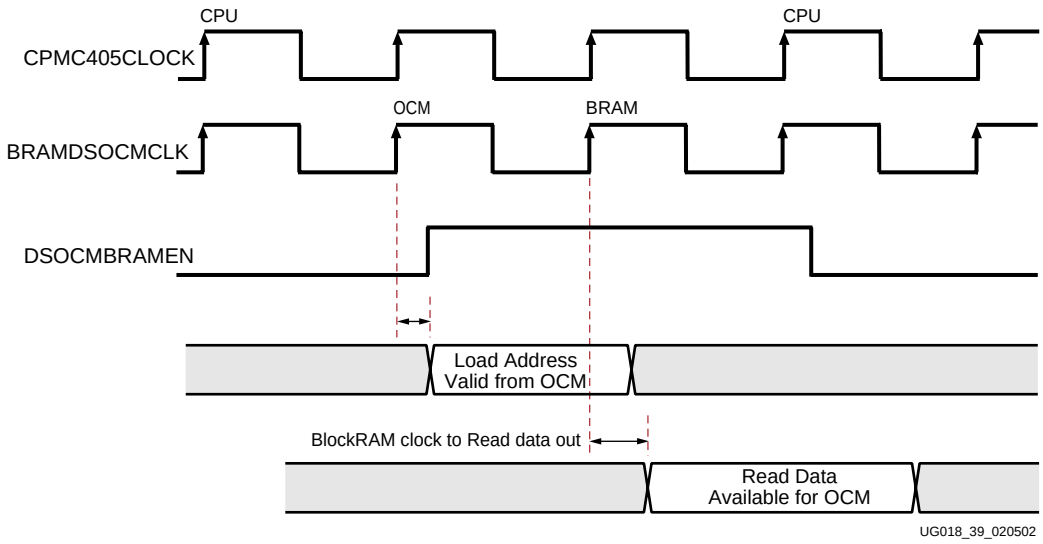


Figure 3-5: **Single-Cycle Static Performance Analysis of DSOCM (Load Operation)**

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.

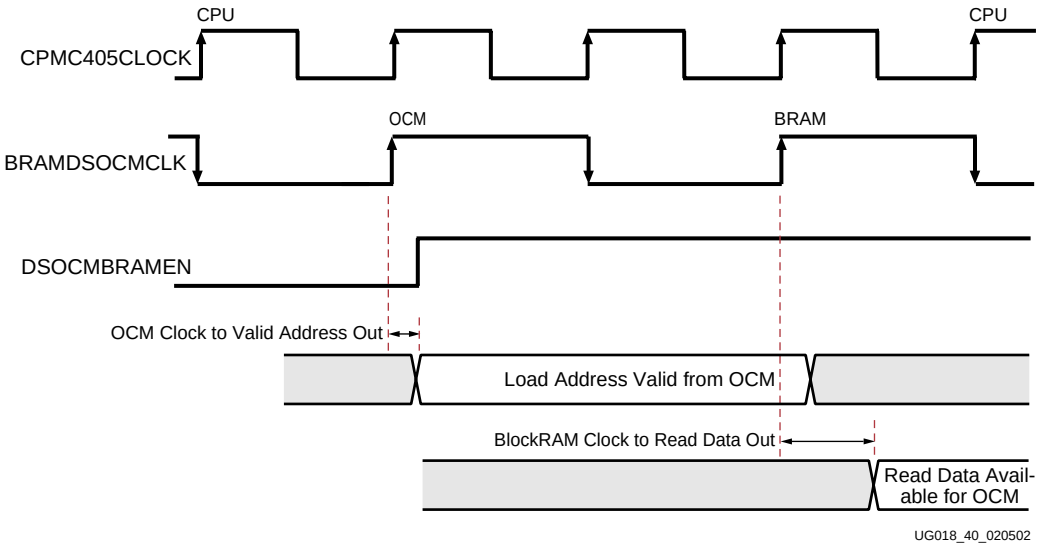


Figure 3-6: **Multi-Cycle Static Performance Analysis of DSOCM (Load Operation)**

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.

Reserved(1)

DSOCMBUSY(2)

DISABLEOPERANDFWD(3)

DSOCMEN(4)

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 ISARCVALUE 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 ISCNTLVALUE inputs to the processor block.

Reserved(1)

ISOCMEN(2)

ISOCMMCM[0:2]	CPMC405CLOCK: BRAMISOCMCLK 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

2*n* – 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. ISOCMEN:
   Enables the DSOCM address decoder.

ISINIT (ISOCM Initalization 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, BRAMDSOCMCLK, BRAMISOCMCLK, CPMC405CLOCK, RST,
         DSOCMBRAMABUS, DSOCMBRAMEN, DSOCMBRAMWRDBUS, BRAMDSOCMRDDBUS,
           DSOCMBRAMBYTEWRITE, ADDRB, 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        BRAMDSOCMCLK;       // 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]  ADDRB; // 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         BRAMISOCMCLK;          // 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 BRAMDSOCMCLK = OCMCLK;
assign BRAMISOCMCLK = 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               ( ADDRB ),
     .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 ),         .O ( CLK ));
BUFG   buf2 (.I ( clk2x_i ),       .O ( CLK2X ));
BUFG   buf3 (.I ( clkmult_i ),     .O ( 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,
                     BRAMDSOCMCLK,
                     DSOCMBRAMWRDBUS,
                     BRAMDSOCMRDDBUS,
                     DSOCMBRAMBYTEWRITE,
                     ADDRB,
                     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        BRAMDSOCMCLK;       // 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   ( ADDRB[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   ( ADDRB[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
```

Appendix A

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.

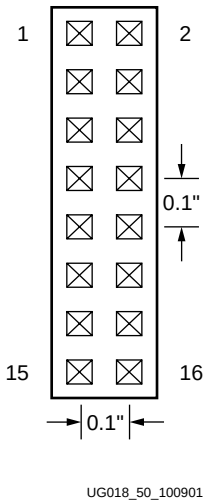


Figure A-1: **JTAG-Connector Physical Layout**

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[1]	JTAG test-data in.
4	Output	$\overline{\text{TRST}}$	JTAG test reset.
5	No Connect	Reserved	
6	Output	+Power[2]	Processor power OK
7	Output	TCK[3]	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 10KΩ 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 (1KΩ or less) should be used to provide short-circuit current-limiting protection.
3. A 10KΩ 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[1]	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
DBGC405DEBUGHALT[2]	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.

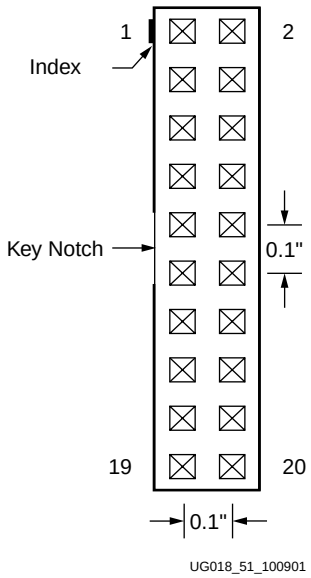


Figure A-2: **Trace-Connector Physical Layout**

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

Appendix B

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 896.

Table B-1: **PPC405x3 Interface Signals in Alphabetical Order**

Signal	I/O Type	Interface	If Unused	Function
C405CPMCORESLEEPREQ	O	CPM, page 899	No Connect	Indicates the core is requesting to be put into sleep mode.
C405CPMMSRCE	O	CPM, page 899	No Connect	Indicates the value of MSR[CE].
C405CPMMSREE	O	CPM, page 899	No Connect	Indicates the value of MSR[EE].
C405CPMTIMERIRQ	O	CPM, page 899	No Connect	Indicates a timer-interrupt request occurred.
C405CPMTIMERRESETREQ	O	CPM, page 899	No Connect	Indicates a watchdog-timer reset request occurred.
C405DBGMSRWE	O	Debug, page 977	No Connect	Indicates the value of MSR[WE].
C405DBGSTOPACK	O	Debug, page 977	No Connect	Indicates the PPC405x3 is in debug halt mode.
C405DBGWBCOMPLETE	O	Debug, page 976	No Connect	Indicates the current instruction in the PPC405x3 writeback pipeline stage is completing.
C405DBGWBFULL	O	Debug, page 976	No Connect	Indicates the PPC405x3 writeback pipeline stage is full.
C405DBGWBIAR[0:29]	O	Debug, page 976	No Connect	The address of the current instruction in the PPC405x3 writeback pipeline stage.
C405DCRABUS[0:9]	O	DCR, page 962	No Connect	Specifies the address of the DCR access request.
C405DCRDBUSOUT[0:31]	O	DCR, page 962	No Connect or attach to input bus	The 32-bit DCR write-data bus.
C405DCRREAD	O	DCR, page 961	No Connect	Indicates a DCR read request occurred.
C405DCRWRITE	O	DCR, page 961	No Connect	Indicates a DCR write request occurred.
C405JTGCAPTUREDR	O	JTAG, page 973	No Connect	Indicates the TAP controller is in the capture-DR state.
C405JTGEXTEST	O	JTAG, page 973	No Connect	Indicates the JTAG EXTEST instruction is selected.
C405JTGPGMOUT	O	JTAG, page 974	No Connect	Indicates the state of a general purpose program bit in the JTAG debug control register (JDCR).
C405JTGSHIFTDR	O	JTAG, page 973	No Connect	Indicates the TAP controller is in the shift-DR state.
C405JTGTDO	O	JTAG, page 973	No Connect	JTAG TDO (test-data out).
C405JTGTDOEN	O	JTAG, page 973	No Connect	Indicates the JTAG TDO signal is enabled.

Table B-1: **PPC405x3 Interface Signals in Alphabetical Order *(Continued)***

Signal	I/O Type	Interface	If Unused	Function
C405JTGUPDATEDR	O	JTAG, page 974	No Connect	Indicates the TAP controller is in the update-DR state.
C405PLBDCUABORT	O	DSPLB, page 937	No Connect	Indicates the DCU is aborting an unacknowledged data-access request.
C405PLBDCUABUS[0:31]	O	DSPLB, page 934	No Connect	Specifies the memory address of the data-access request.
C405PLBDCUBE[0:7]	O	DSPLB, page 936	No Connect	Specifies which bytes are transferred during single-word transfers.
C405PLBDCUCACHEABLE	O	DSPLB, page 934	No Connect	Indicates the value of the cacheability storage attribute for the target address.
C405PLBDCUGUARDED	O	DSPLB, page 935	No Connect	Indicates the value of the guarded storage attribute for the target address.
C405PLBDCUPRIORITY[0:1]	O	DSPLB, page 937	No Connect	Indicates the priority of the data-access request.
C405PLBDCUREQUEST	O	DSPLB, page 933	No Connect	Indicates the DCU is making a data-access request.
C405PLBDCURNW	O	DSPLB, page 934	No Connect	Specifies whether the data-access request is a read or a write.
C405PLBDCUSIZE2	O	DSPLB, page 934	No Connect	Specifies a single word or eight-word transfer size.
C405PLBDCUU0ATTR	O	DSPLB, page 935	No Connect	Indicates the value of the user-defined storage attribute for the target address.
C405PLBDCUWRDBUS[0:63]	O	DSPLB, page 938	No Connect	The DCU write-data bus used to transfer data from the DCU to the PLB slave.
C405PLBDCUWRITETHRU	O	DSPLB, page 935	No Connect	Indicates the value of the write-through storage attribute for the target address.
C405PLBICUABORT	O	ISPLB, page 913	No Connect	Indicates the ICU is aborting an unacknowledged fetch request.
C405PLBICUABUS[0:29]	O	ISPLB, page 911	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, page 912	No Connect	Indicates the value of the cacheability storage attribute for the target address.
C405PLBICUPRIORITY[0:1]	O	ISPLB, page 913	No Connect	Indicates the priority of the ICU fetch request.
C405PLBICUREQUEST	O	ISPLB, page 911	No Connect	Indicates the ICU is making an instruction-fetch request.
C405PLBICUSIZE[2:3]	O	ISPLB, page 911	No Connect	Specifies a four word or eight word line-transfer size.
C405PLBICUU0ATTR	O	ISPLB, page 912	No Connect	Indicates the value of the user-defined storage attribute for the target address.
C405RSTCHIPRESETREQ	O	Reset, page 905	Required	Indicates a chip-reset request occurred.
C405RSTCORERESETREQ	O	Reset, page 904	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, page 905	Required	Indicates a system-reset request occurred.
C405TRCCYCLE	O	Trace, page 979	No Connect	Specifies the trace cycle.
C405TRCEVENEXECUTIONSTATUS[0:1]	O	Trace, page 980	No Connect	Specifies the execution status collected during the first of two processor cycles.
C405TRCODDEXECUTIONSTATUS[0:1]	O	Trace, page 980	No Connect	Specifies the execution status collected during the second of two processor cycles.
C405TRCTRACESTATUS[0:3]	O	Trace, page 980	No Connect	Specifies the trace status.
C405TRCTRIGGEREVENTOUT	O	Trace, page 979	Wrap to Trigger Event In	Indicates a trigger event occurred.
C405TRCTRIGGEREVENTTYPE[0:10]	O	Trace, page 979	No Connect	Specifies which debug event caused the trigger event.
C405XXXMACHINECHECK	O	Control, page 902	No Connect	Indicates a machine-check error has been detected by the PPC405x3.
CPMC405CLOCK	I	CPM, page 898	Required	PPC405x3 clock input (for all non-JTAG logic, including timers).
CPMC405CORECLKINACTIVE	I	CPM, page 898	0	Indicates the CPM logic disabled the clocks to the core.
CPMC405CPUCLKEN	I	CPM, page 898	1	Enables the core clock zone.
CPMC405JTAGCLKEN	I	CPM, page 898	1	Enables the JTAG clock zone.
CPMC405TIMERCLKEN	I	CPM, page 898	1	Enables the timer clock zone.
CPMC405TIMERTICK	I	CPM, page 899	1	Increments or decrements the PPC405x3 timers every time it is active with the CPMC405CLOCK.
DBGC405DEBUGHALT	I	Debug, page 976	0	Indicates the external debug logic is placing the processor in debug halt mode.
DBGC405EXTBUSHOLDACK	I	Debug, page 975	0	Indicates the bus controller has given control of the bus to an external master.
DBGC405UNCONDDEBUGEVENT	I	Debug, page 976	0	Indicates the external debug logic is causing an unconditional debug event.
DCRC405ACK	I	DCR, page 962	0	Indicates a DCR access has been completed by a peripheral.
DCRC405DBUSIN[0:31]	I	DCR, page 962	0x0000_0000 or attach to output bus	The 32-bit DCR read-data bus.
EICC405CRITINPUTIRQ	I	EIC, page 969	0	Indicates an external critical interrupt occurred.
EICC405EXTINPUTIRQ	I	EIC, page 969	0	Indicates an external noncritical interrupt occurred.
JTGC405BNDSCANTDO	I	JTAG, page 973	0	JTAG boundary scan input from the previous boundary scan element TDO output.
JTGC405TCK	I	JTAG, page 972	See IEEE 1149.1	JTAG TCK (test clock).
JTGC405TDI	I	JTAG, page 972	1	JTAG TDI (test-data in).
JTGC405TMS	I	JTAG, page 972	1	JTAG TMS (test-mode select).
JTGC405TRSTNEG	I	Reset, page 906	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, page 973	Required	JTAG $\overline{\text{TRST}}$ (test reset).
MCBCPUCLKEN	I	FPGA, page 981	1	Indicates the PPC405x3 clock enable should follow GWE during a partial reconfiguration.
MCBJTAGEN	I	FPGA, page 981	1	Indicates the JTAG clock enable should follow GWE during a partial reconfiguration.
MCBTIMEREN	I	FPGA, page 981	1	Indicates the timer clock enable should follow GWE during a partial reconfiguration.
MCPPCRST	I	FPGA, page 981	1	Indicates the PPC405x3 should be reset when GSR is asserted during a partial reconfiguration.
PLBC405DCUADDRACK	I	DSPLB, page 939	0	Indicates a PLB slave acknowledges the current data-access request.
PLBC405DCUBUSY	I	DSPLB, page 942	0	Indicates the PLB slave is busy performing an operation requested by the DCU.
PLBC405DCUERR	I	DSPLB, page 943	0	Indicates an error was detected by the PLB slave during the transfer of data to or from the DCU.
PLBC405DCURDDACK	I	DSPLB, page 941	0	Indicates the DCU read-data bus contains valid data for transfer to the DCU.
PLBC405DCURDDBUS[0:63]	I	DSPLB, page 941	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, page 941	0b000	Indicates which word or doubleword of an eight-word line transfer is present on the DCU read-data bus.
PLBC405DCUSSIZE1	I	DSPLB, page 940	0	Specifies the bus width (size) of the PLB slave that accepted the request.
PLBC405DCUWRDACK	I	DSPLB, page 942	0	Indicates the data on the DCU write-data bus is being accepted by the PLB slave.
PLBC405ICUADDRACK	I	ISPLB, page 913	0	Indicates a PLB slave acknowledges the current ICU fetch request.
PLBC405ICUBUSY	I	ISPLB, page 916	0	Indicates the PLB slave is busy performing an operation requested by the ICU.
PLBC405ICUERR	I	ISPLB, page 917	0	Indicates an error was detected by the PLB slave during the transfer of instructions to the ICU.
PLBC405ICURDDACK	I	ISPLB, page 914	0	Indicates the ICU read-data bus contains valid instructions for transfer to the ICU.
PLBC405ICURDDBUS[0:63]	I	ISPLB, page 915	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, page 915	0b000	Indicates which word or doubleword of a four-word or eight-word line transfer is present on the ICU read-data bus.
PLBC405ICUSSIZE1	I	ISPLB, page 914	0	Specifies the bus width (size) of the PLB slave that accepted the request.
PLBCLK	I	FPGA, page 982	Required	PLB clock.
RSTC405RESETCHIP	I	Reset, page 905	Required	Indicates a chip-reset occurred.
RSTC405RESETCORE	I	Reset, page 905	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, page 906	Required	Indicates a system-reset occurred. Resets the logic in the PPC405x3 JTAG unit.
TIEC405DETERMINISTICMULT	I	Control, page 901	Required	Specifies whether all multiply operations complete in a fixed number of cycles or have an early-out capability.
TIEC405DISOPERANDFWD	I	Control, page 902	Required	Disables operand forwarding for load instructions.
TIEC405MMUEN	I	Control, page 901	Required	Enables the memory-management unit (MMU)
TRCC405TRACEDISABLE	I	Trace, page 980	0	Disables trace collection and broadcast.
TRCC405TRIGGEREVENTIN	I	Trace, page 980	Wrap to Trigger Event Out	Indicates a trigger event occurred and that trace status is to be generated.

Index

S

T

U

V

W

Volume 3: Rocket I/O™ Transceiver User Guide

Virtex-II Pro™ Platform FPGA Documentation

March 2002 Release

XILINX®

XILINX®

Chapter 1

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**.

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 a "PowerPC+Software+Devlopment". 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."

Chapter 2

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)(1)	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

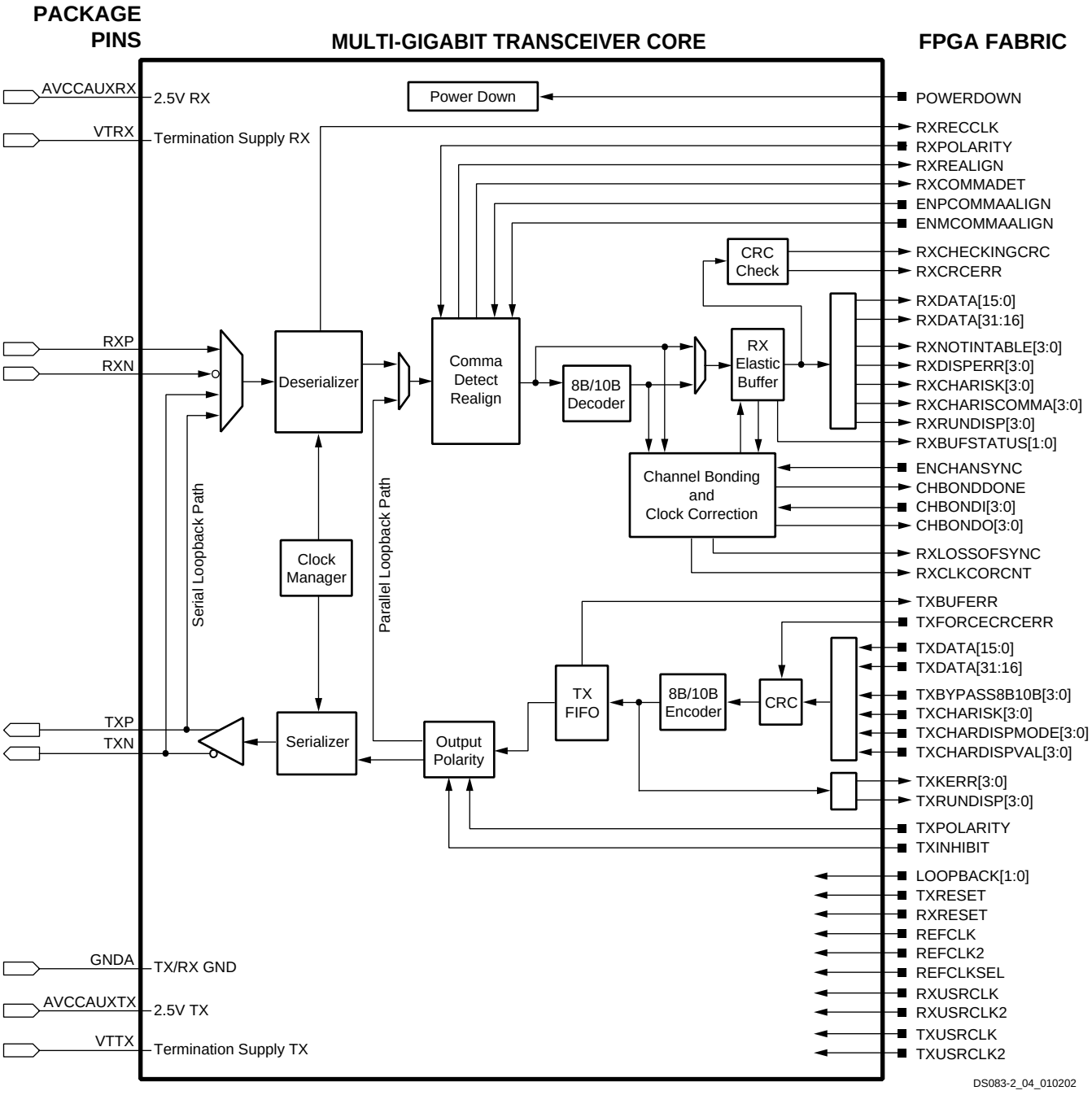


Figure 2-1: **Rocket I/O Transceiver Block Diagram**

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 ±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, 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 recived 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.

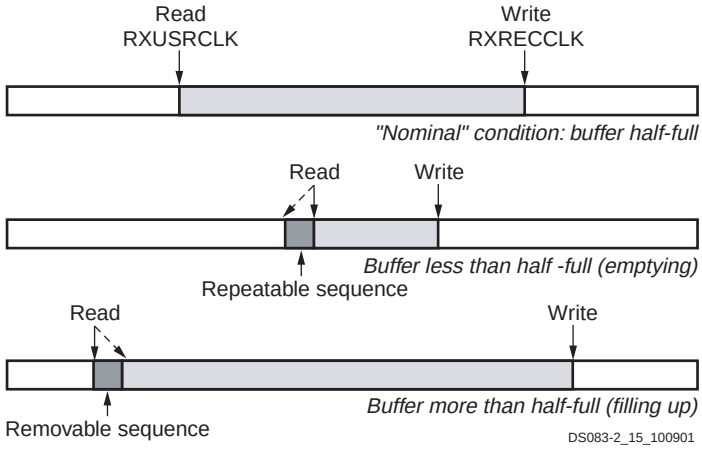


Figure 2-2: **Clock Correction in Receiver**

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.

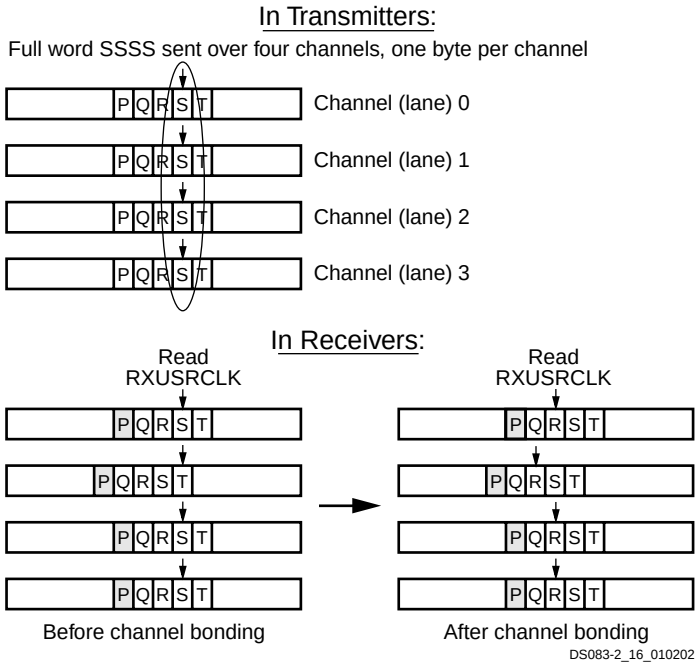


Figure 2-3: **Channel Bonding (Alignment)**

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

XILINX®

Chapter 3

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(1), GT_AURORA, GT_FIBRE_CHAN(2), GT_ETHERNET(2), GT_INFINIBAND, and GT_XAUI Primitive Ports**

Port	I/O	Port Size(3)	Definition
CHBONDDONE(2)	O	1	Indicates a receiver has successfully completed channel bonding when asserted High.
CHBONDI(2)	I	4	The channel bonding control that is used only by "slaves" which is driven by a transceiver's CHBONDO port.
CHBONDO(2)	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(2)	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[1], GT_AURORA, GT_FIBRE_CHAN[2], GT_ETHERNET[2], GT_INFINIBAND, and GT_XAUI Primitive Ports *(Continued)***

Port	I/O	Port Size[3]	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 ± 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 **Clock Correction Count**, 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) If RX_LOSS_OF_SYNC_FSM = TRUE, this outputs the state of the FSM. Bit 1 signals a loss of sync. Bit 0 indicates a resync state. 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[4]	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[4]	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[1], GT_AURORA, GT_FIBRE_CHAN[2], GT_ETHERNET[2], GT_INFINIBAND, and GT_XAUI Primitive Ports *(Continued)***

Port	I/O	Port Size[3]	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(1), GT_AURORA, GT_FIBRE_CHAN(2), GT_ETHERNET(2), GT_INFINIBAND, and GT_XAUI Primitive Ports *(Continued)***

Port	I/O	Port Size(3)	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(4)	O	1	Transmit differential port (FPGA external)
TXP(4)	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 an 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	True/False controls the alignment of detected commas within the transceivers 2-byte wide data path. **False:** 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. **True:** Aligns comma with 20-bit alignment range. As a result aligns on the RXDATA[15:8] byte. **NOTE:** 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.
CHAN_BOND_LIMIT	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.
CHAN_BOND_MODE	STRING OFF, MASTER, SLAVE_1_HOP, SLAVE_2_HOPS **OFF:** No channel bonding involving this transceiver. **MASTER:** This transceiver is master for channel bonding. Its CHBONDO port directly drives CHBONDI ports on one or more SLAVE_1_HOP transceivers. **SLAVE_1_HOP:** 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. **SLAVE_2_HOPS:** This transceiver is a slave for channel bonding. SLAVE_2_HOPS CHBONDI is directly driven by a SLAVE_1_HOP CHBONDO port.
CHAN_BOND_OFFSET	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. 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. 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. For optimal performance of the elastic buffer, CHAN_BOND_WAIT and CHAN_BOND_OFFSET should be set to the same value (typically 8).

Table 3-2: **Rocket I/O Transceiver Attributes *(Continued)***

Attribute	Description
CHAN_BOND_ONE_SHOT	True/False that controls repeated execution of channel bonding. **False:** 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. **True:** 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. Slave transceivers should always have CHAN_BOND_ONE_SHOT set to False.
CHAN_BOND_SEQ_*_*	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 **Receiving Vitesse Channel Bonding Sequence**, page 1103, for format.
CHAN_BOND_SEQ_2_USE	Controls use of second channel bonding sequence. **False:** Channel bonding uses only one channel bonding sequence defined by CHAN_BOND_SEQ_1_1..4. **True:** 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.
CHAN_BOND_SEQ_LEN	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.
CHAN_BOND_WAIT	Integer 1-15 that defines the length of wait (in bytes) after seeing channel bonding sequence before executing channel bonding.
CLK_COR_INSERT_IDLE_FLAG	True/False controls whether RXRUNDISP input status denotes running disparity or inserted-idle flag. **False:** RXRUNDISP denotes running disparity when RXDATA is decoded data. **True:** RXRUNDISP is raised for the first byte of each inserted (repeated) clock correction ("Idle") sequence (when RXDATA is decoded data).
CLK_COR_KEEP_IDLE	True/False controls whether or not the final byte stream must retain at least one clock correction sequence. **False:** Transceiver can remove all clock correction sequences to further re-center the elastic buffer during clock correction. **True:** In the final RXDATA stream, the transceiver must leave at least one clock correction sequence per continuous stream of clock correction sequences.
CLK_COR_REPEAT_WAIT	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.

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. **False:** Clock correction uses only one clock correction sequence defined by CLK_COR_SEQ_1_1..4. **True:** Clock correction uses two clock correction sequences defined by: CLK_COR_SEQ_1_1..4 and CLK_COR_SEQ_2_1..4 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. **False:** Permanently disable execution of clock correction (rate matching). Clock RXUSRCLK must be frequency-locked with RXRECCLK in this case. **True:** 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	True/False controls the raising of RXCHARISCOMMA on an invalid comma.
	False: Raise RXCHARISCOMMA on:
	`0011111xxx`
	(if DEC_PCOMMA_DETECT is TRUE)
	and/or on:
	`1100000xxx`
	(if DEC_MCOMMA_DETECT is TRUE)
	regardless of the settings of the `xxx` bits.
	True: Raise RXCHARISCOMMA only on valid characters that are in the 8B/10B translation.

MCOMMA_10B_VALUE	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.
MCOMMA_DETECT	True/False indicates whether to raise or not raise the RXCOMMADET when minus-comma is detected.
PCOMMA_10B_VALUE	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.
PCOMMA_DETECT	True/False indicates whether to raise or not raise the RXCOMMADET when plus-comma is detected.
RX_BUFFER_USE	Always set to True.
RX_CRC_USE, TX_CRC_USE	True/False determines if CRC is used or not.
RX_DATA_WIDTH, TX_DATA_WIDTH	Integer (1, 2, or 4). Relates to the data width of the FPGA fabric interface.
RX_DECODE_USE	This determines if the 8B/10B decoding is bypassed. False denotes that it is bypassed.
RX_LOS_INVALID_INCR	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.
RX_LOS_THRESHOLD	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.
RX_LOSS_OF_SYNC_FSM	True/False denotes the nature of RXLOSSOFSYNC output. **True:** RXLOSSOFSYNC outputs the state of the FSM bit. See **RXLOSSOFSYNC**, page 1070, for details.
SERDES_10B	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.

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 Table 4-2, page 1120.

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(1)	Default GT_ETHERNET
ALIGN_COMMA_MSB	False	False	False
CHAN_BOND_LIMIT	16	16	1
CHAN_BOND_MODE	OFF(2)	OFF	OFF
CHAN_BOND_OFFSET	8	8	0
CHAN_BOND_ONE_SHOT	Falso(2)	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(2)	False	False(2)
CLK_COR_KEEP_IDLE	False(2)	False	False(2)
CLK_COR_REPEAT_WAIT	1(2)	1	1(2)
CLK_COR_SEQ_1_1	00100011100	00000000000	00110111100
CLK_COR_SEQ_1_2	00100011100(4)	00000000000	00001010000
CLK_COR_SEQ_1_3	00100011100(5)	00000000000	00000000000
CLK_COR_SEQ_1_4	00100011100(5)	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 GT_AURORA	Default GT_CUSTOM(1)	Default GT_ETHERNET
CLK_COR_SEQ_LEN	N(3)	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(2)	False	False(2)
RX_DATA_WIDTH	N(3)	2	N(3)
RX_DECODE_USE	True	True	True
RX_LOS_INVALID_INCR	1(2)	1	1(2)
RX_LOS_THRESHOLD	4(2)	4	4(2)
RX_LOSS_OF_SYNC_FSM	True(2)	True	True(2)
SERDES_10B	False(2)	False	False(2)
TERMINATION_IMP	50(2)	50	50(2)
TX_BUFFER_USE	True	True	True
TX_CRC_FORCE_VALUE	11010110(2)	11010110	11010110(2)
TX_CRC_USE	False(2)	False	False(2)
TX_DATA_WIDTH	N(3)	2	N(3)
TX_DIFF_CTRL	500(2)	500	500(2)
TX_PREEMPHASIS	0(2)	0	0(2)

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 GT_FIBRE_CHAN	Default GT_INFINIBAND	Default GT_XAUI
ALIGN_COMMA_MSB	False	False	False
CHAN_BOND_LIMIT	1	16	16
CHAN_BOND_MODE	OFF	OFF(1)	OFF(1)
CHAN_BOND_OFFSET	0	8	8
CHAN_BOND_ONE_SHOT	True	False(1)	False(1)
CHAN_BOND_SEQ_1_1	00000000000	00110111100	00101111100
CHAN_BOND_SEQ_1_2	00000000000	Lane ID (Modify with Lane ID)	00000000000
CHAN_BOND_SEQ_1_3	00000000000	00001001010	00000000000
CHAN_BOND_SEQ_1_4	00000000000	00001001010	00000000000
CHAN_BOND_SEQ_2_1	00000000000	00110111100	00000000000
CHAN_BOND_SEQ_2_2	00000000000	Lane ID (Modify with Lane ID)	00000000000
CHAN_BOND_SEQ_2_3	00000000000	00001000101	00000000000
CHAN_BOND_SEQ_2_4	00000000000	00001000101	00000000000
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(1)	False(1)	False(1)
CLK_COR_KEEP_IDLE	False(1)	False(1)	False(1)
CLK_COR_REPEAT_WAIT	2(1)	1(1)	1(1)
CLK_COR_SEQ_1_1	00110111100	00100011100	00100011100
CLK_COR_SEQ_1_2	00010010101	00000000000	00000000000
CLK_COR_SEQ_1_3	00010110101	00000000000	00000000000
CLK_COR_SEQ_1_4	00010110101	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
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(1)

Table 3-4: **Default Attribute Values: GT_FIBRE_CHAN, GT_INFINIBAND, and GT_XAUI *(Continued)***

Attribute	Default GT_FIBRE_CHAN	Default GT_INFINIBAND	Default GT_XAUI
CRC_FORMAT	FIBRE_CHAN	INFINIBAND	USER_MODE(1)
CRC_START_OF_PKT	K27_7	Note (3)	K27_7(1)
DEC_MCOMMA_DETECT	True	True	True
DEC_PCOMMA_DETECT	True	True	True
DEC_VALID_COMMA_ONLY	True	True	True
Lane ID(INFINBAND ONLY)	NA	00000000000(1)	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(1)	False(1)	False(1)
RX_DATA_WIDTH	N(2)	N(2)	N(2)
RX_DECODE_USE	True	True	True
RX_LOS_INVALID_INCR	1(1)	1(1)	1(1)
RX_LOS_THRESHOLD	4(1)	4(1)	4(1)
RX_LOSS_OF_SYNC_FSM	True(1)	True(1)	True(1)
SERDES_10B	False(1)	False(1)	False(1)
TERMINATION_IMP	50(1)	50(1)	50(1)
TX_BUFFER_USE	True	True	True
TX_CRC_FORCE_VALUE	11010110(1)	11010110(1)	11010110(1)
TX_CRC_USE	False(1)	False(1)	False(1)
TX_DATA_WIDTH	N(2)	N(2)	N(2)
TX_DIFF_CTRL	500(1)	500(1)	500(1)
TX_PREEMPHASIS	0(1)	0(1)	0(1)

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(1)
2 byte	1:1
4 byte	2:1(1)

Notes:
1. 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 RX_DATA_WIDTH	REFCLK	TXUSRCLK RXUSRCLK	TXUSRCLK2 RXUSRCLK2
False	1	CLKIN	CLK0	CLK2X180
False	2	CLKIN	CLK0	CLK0
False	4	CLKIN	CLK180(1)	CLKDV (divide by 2)
True	1	CLKIN	CLKDV (divide by 2)	CLK180(1)
True	2	CLKIN	CLKDV (divide by 2)	CLKDV (divide by 2)
True	4	CLKIN	CLKFX180 (divide by 2)	CLKDV (divide by 4)

Notes:
1. 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.

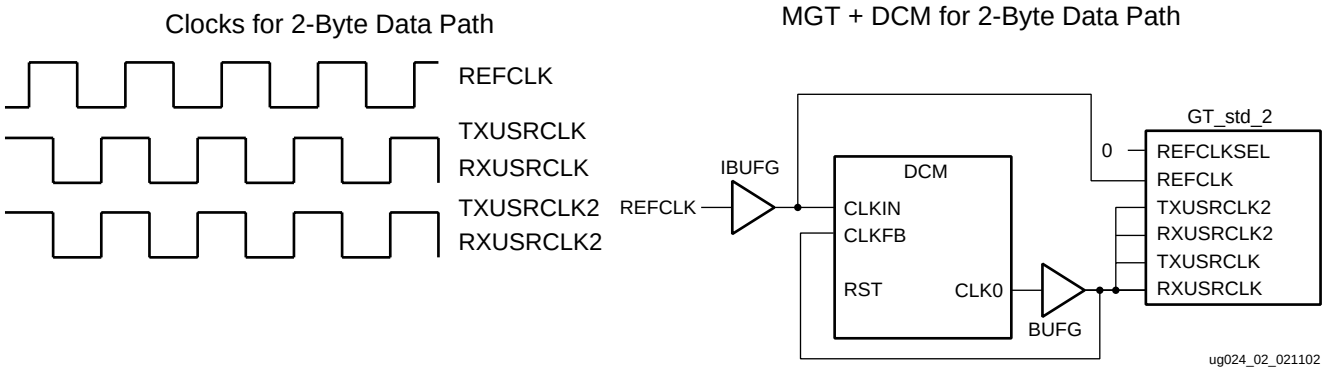


Figure 3-1: **Two-Byte Clock**

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;
--
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 ),
         .PSCLK      ( 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 ),
```

```
                .O ( 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.

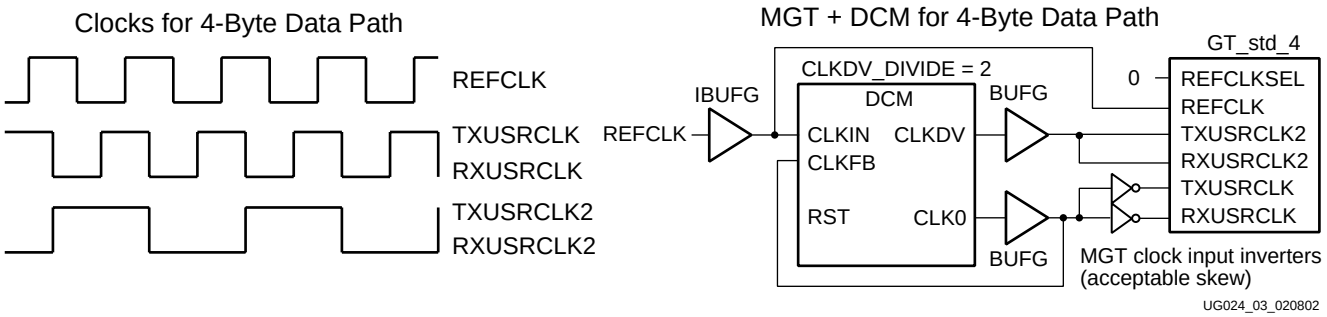


Figure 3-2: **Four-Byte Clock**

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;
```

```
      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 CLKDV_W   : std_logic;

begin

GND   <= '0';
-- DCM Instantiation
U_DCM: DCM
  port map(
         CLKIN     =>   REFCLK,   CLKFB     =>   CLK0_W,
         DSSEN     =>   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,
```

```
          O => REFCLK    );

  U2_BUFG: BUFG
    port map(
          I => CLK0_W,
          O => USRCLK_M
          );

  U3_BUFG: 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.

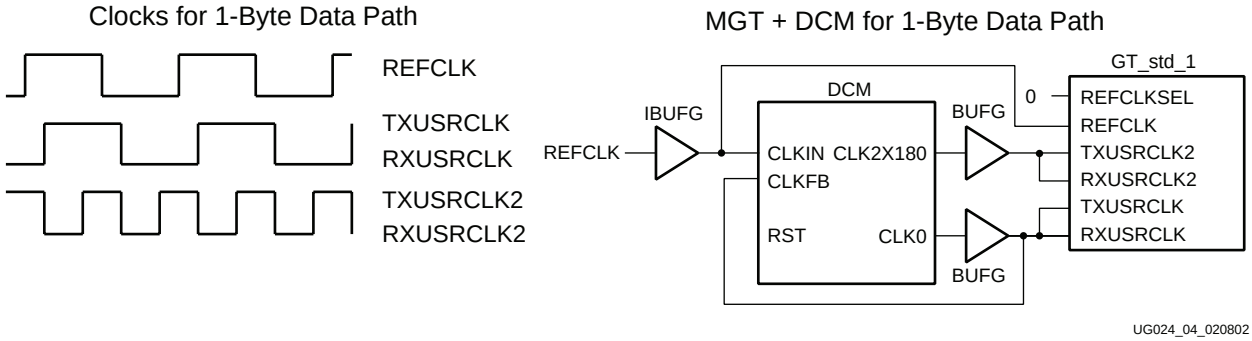


Figure 3-3: **One-Byte Clock**

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      ( 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   ( clk2x_180 ),
                .CLKDV      (  ),
                .CLKFX      (  ),
                .CLKFX180   (  ),
                .LOCKED     ( DCM_LOCKED ),
                .PSDONE     (  ),
                .STATUS     (  )
                );


                 BUFG buf1 (
                           .I ( clk2x_180 ),
                           .O ( USRCLK2_M )
                           );

                 BUFG buf2 (
                           .I ( clk_i ),
                           .O ( USRCLK_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.

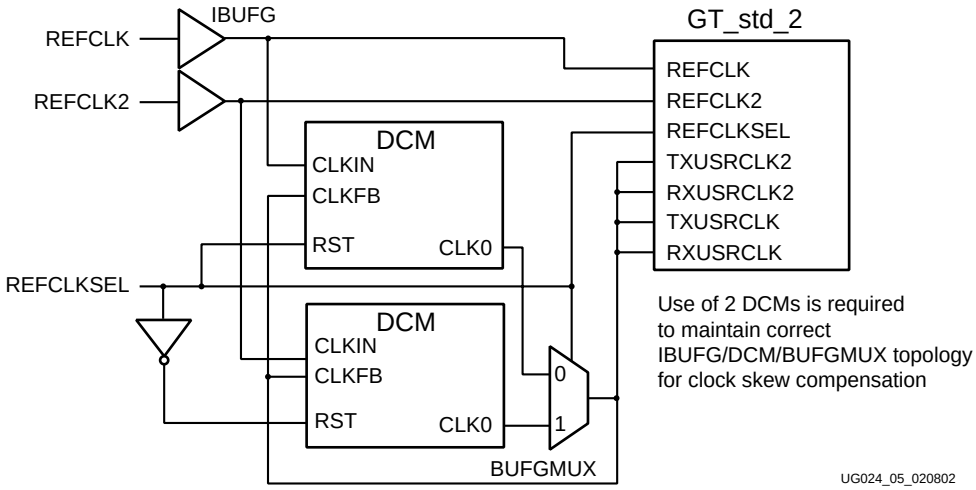


Figure 3-4: **Multiplexed REFCLK**

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
Setup/Hold:		
$T_{GCCK_}$CHBI/$T_{GCKC_}$CHBI	Control inputs	CHBONDI[3:0]
Clock to Out:		
$T_{GCKCO_}$CHBO	Control outputs	CHBONDO[3:0]
Clock:		
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
Setup/Hold:		
$T_{GCCK_}$RRST/$T_{GCKC_}$RRST	Control input	RXRESET
$T_{GCCK_}$RPOL/$T_{GCKC_}$RPOL	Control input	RXPOLARITY
$T_{GCCK_}$ECSY/$T_{GCKC_}$ECSY	Control input	ENCHANSYNC
Clock to Out:		

Table 3-10: **Parameters Relative to the RX User Clock2 (RXUSRCLK2) *(Continued)***

Parameter	Function	Signals
T_{GCKST}_RNIT	Status outputs	RXNOTINTABLE[3:0]
T_{GCKST}_RDERR	Status outputs	RXDISPERR[3:0]
T_{GCKST}_RCMCH	Status outputs	RXCHARISCOMMA[3:0]
T_{GCKST}_ALIGN	Status output	RXREALIGN
T_{GCKST}_CMDT	Status output	RXCOMMADET
T_{GCKST}_RLOS	Status outputs	RXLOSSOFSYNC[1:0]
T_{GCKST}_RCCCNT	Status outputs	RXCLKCORCNT[2:0]
T_{GCKST}_RBSTA	Status outputs	RXBUFSTATUS[1:0]
T_{GCKST}_RCCRC	Status output	RXCHECKINGCRC
T_{GCKST}_RCRCE	Status output	RXCRCERR
T_{GCKST}_CHBD	Status output	CHBONDDONE
T_{GCKST}_RKCH	Status outputs	RXCHARISK[3:0]
T_{GCKST}_RRDIS	Status outputs	RXRUNDISP[3:0]
T_{GCKDO}_RDAT	Data outputs	RXDATA[31:0]
Clock:		
T_{RX2PWH}	Clock pulse width, High state	RXUSRCLK2
T_{RX2PWH}	Clock pulse width, Low state	RXUSRCLK2

Table 3-11: **Parameters Relative to the TX User Clock2 (TXUSRCLK2)**

Parameter	Function	Signals
Setup/Hold:		
T_{GCCK}_CFGEN/T_{GCKC}_CFGEN	Control inputs	CONFIGENABLE
T_{GCCK}_TBYP/T_{GCKC}_TBYP	Control inputs	TXBYPASS8B10B[3:0]
T_{GCCK}_TCRCE/T_{GCKC}_TCRCE	Control inputs	TXFORCECRCERR
T_{GCCK}_TPOL/T_{GCKC}_TPOL	Control inputs	TXPOLARITY
T_{GCCK}_TINH/T_{GCKC}_TINH	Control inputs	TXINHIBIT
T_{GCCK}_LBK/T_{GCKC}_LBK	Control inputs	LOOPBACK[1:0]
T_{GCCK}_TRST/T_{GCKC}_TRST	Control inputs	TXRESET
T_{GCCK}_TKCH/T_{GCKC}_TKCH	Control inputs	TXCHARISK[3:0]
T_{GCCK}_TCDM/T_{GCKC}_TCDM	Control inputs	TXCHARDISPMODE[3:0]
T_{GCCK}_TCDV/T_{GCKC}_TCDV	Control inputs	TXCHARDISPVAL[3:0]
T_{GDCK}_CFGIN/T_{GCKD}_CFGIN	Data inputs	CONFIGIN
T_{GDCK}_TDAT/T_{GCKD}_TDAT	Data inputs	TXDATA[31:0]
Clock to Out:		
T_{GCKST}_TBERR	Status outputs	TXBUFERR
T_{GCKST}_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
Clock:		
T_{TX2PWH}	Clock pulse width, High state	TXUSRCLK2
T_{TX2PWH}	Clock pulse width, Low state	TXUSRCLK2

Table 3-12: **Miscellaneous Clock Parameters**

Parameter	Function	Signals
Clock:		
T_{REFPWH}	Clock pulse width, High state	REFCLK(1)
T_{REFPWL}	Clock pulse width, Low state	REFCLK(1)
T_{TXPWH}	Clock pulse width, High state	TXUSRCLK(2)
T_{TXPWL}	Clock pulse width, Low state	TXUSRCLK(2)

Notes:
1. REFCLK is not synchronous to any Rocket I/O signals.
2. TXUSRCLK is not synchronous to any Rocket I/O signals.

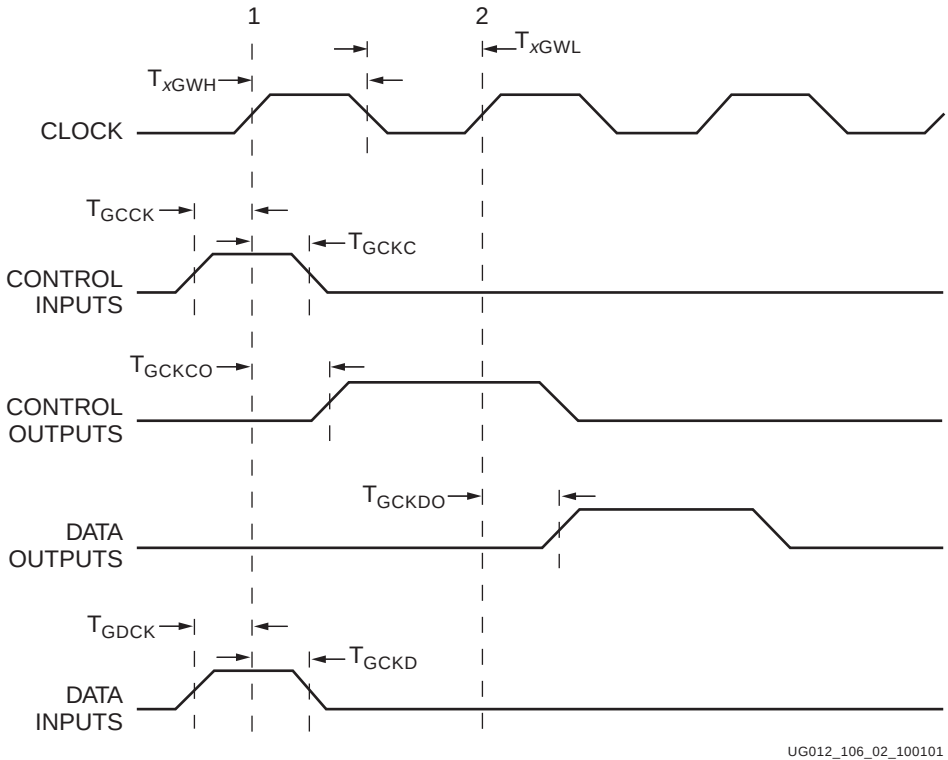


Figure 3-5: **Rocket I/O Timing Relative to Clock Edge**

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 RTL;
```

Verilog Template

```
// Module:       gt_reset
// Description:  Verilog Submodule
//               reset for4-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 (RXLOSSOFSYNC = 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.

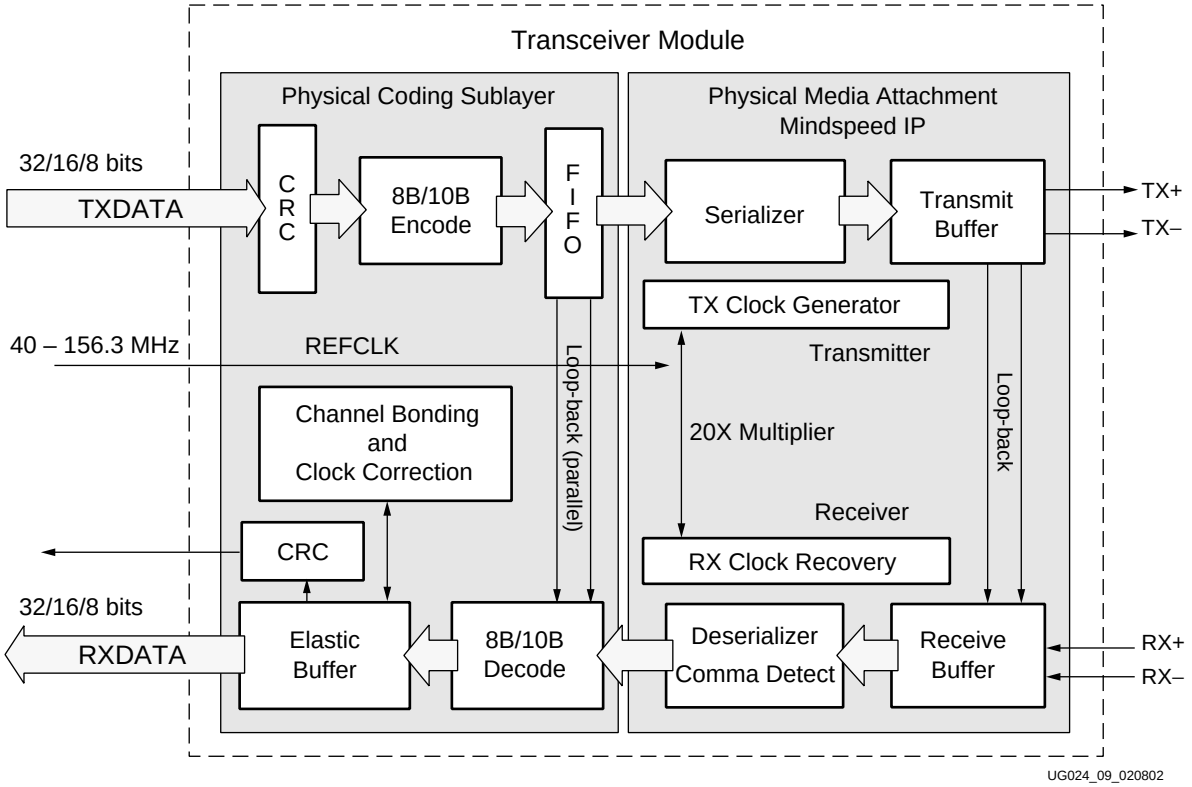


Figure 3-6: **8B/10B Data Flow**

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.

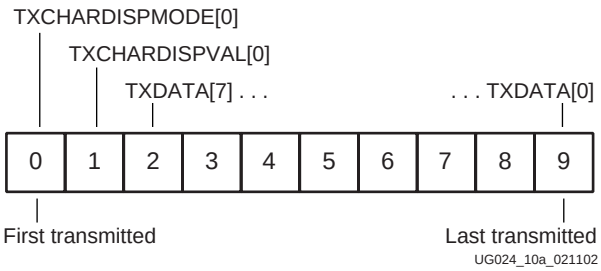


Figure 3-7: **10-Bit TX Data Map with 8B/10B Bypassed**

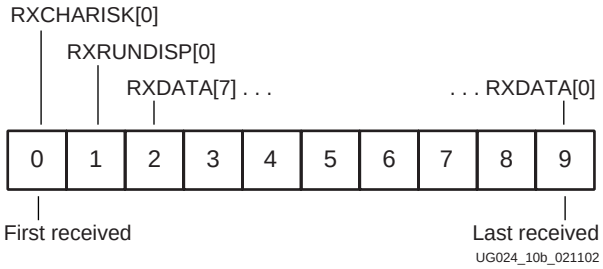


Figure 3-8: **10-Bit RX Data Map with 8B/10B Bypassed**

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 HGF EDCBA	Current RD – abcdei fghj	Current RD + 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 HGF EDCBA	Current RD – abcdei fghj	Current RD + 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 HGF EDCBA	Current RD – abcdei fghj	Current RD + 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 HGF EDCBA	Current RD – abcdei fghj	Current RD + 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 HGF EDCBA	Current RD – abcdei fghj	Current RD + 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 HGF EDCBA	Current RD – abcdei fghj	Current RD + 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 HGF EDCBA	Current RD – abcdei fghj	Current RD + 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 HGF EDCBA	Current RD – abcdei fghj	Current RD + 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.

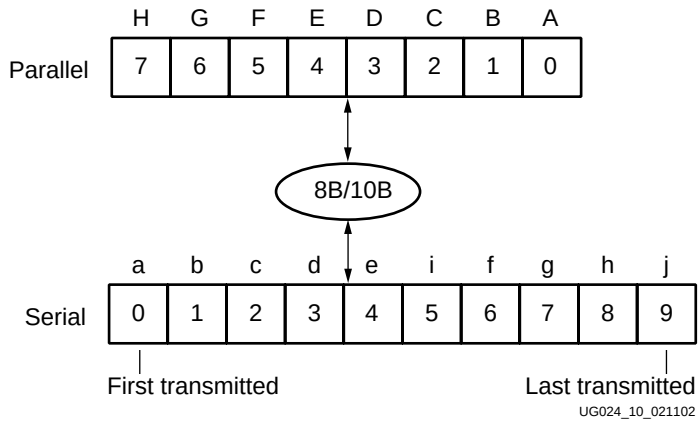


Figure 3-9: **8B/10B Parallel to Serial Conversion**

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].

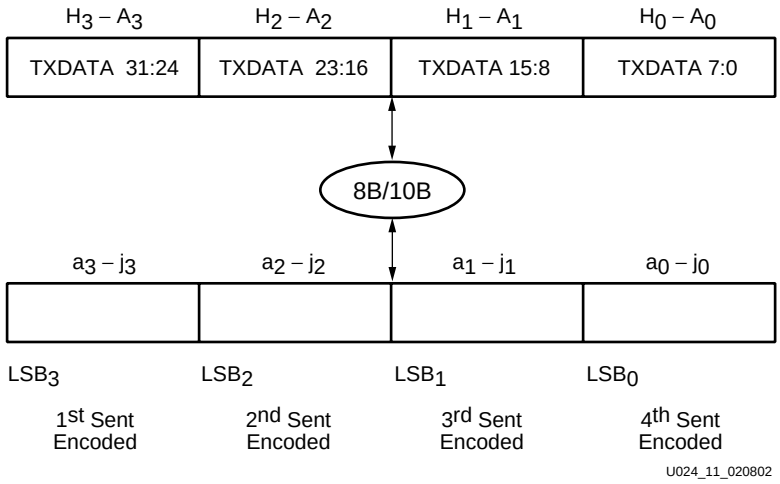


Figure 3-10: **4-Byte Serial Structure**

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.

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, in TXUSRCLK2 Cycles		RXP and RXN to RXDATA, 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.

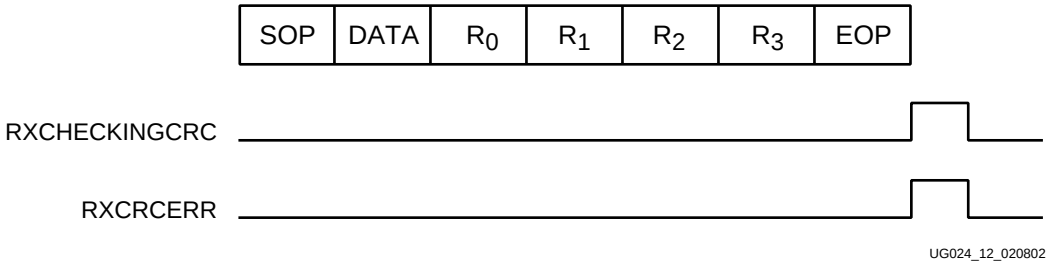


Figure 3-12: **USER_MODE / FIBRE_CHANNEL Mode**

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.

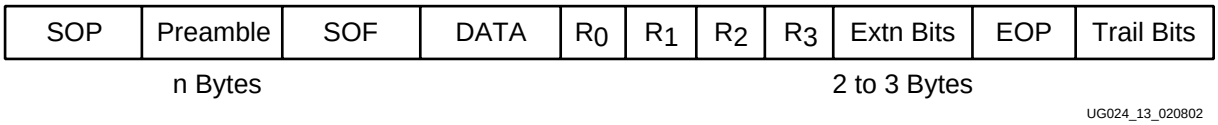


Figure 3-13: **Ethernet Mode**

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.

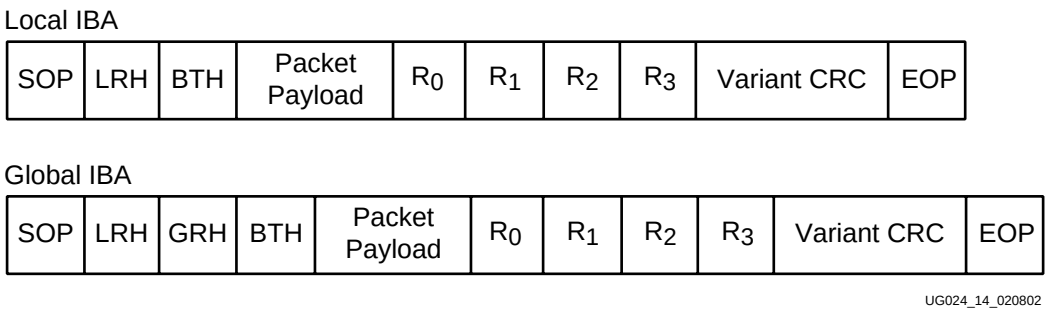


Figure 3-14: **Infiniband Mode**

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.

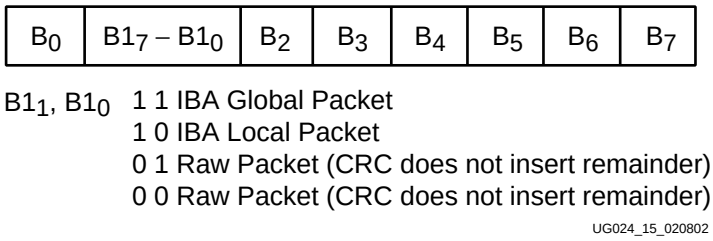


Figure 3-15: **Local Route Header**

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.

Chapter 4

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.

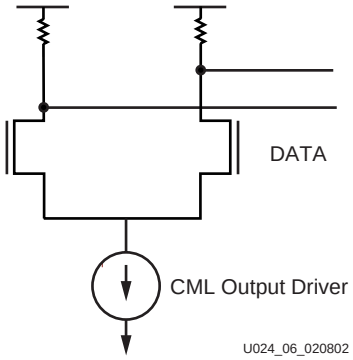


Figure 4-1: **Differential Amplifier**

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 50Ω (or, optionally, 75Ω) 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 program-mable
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).

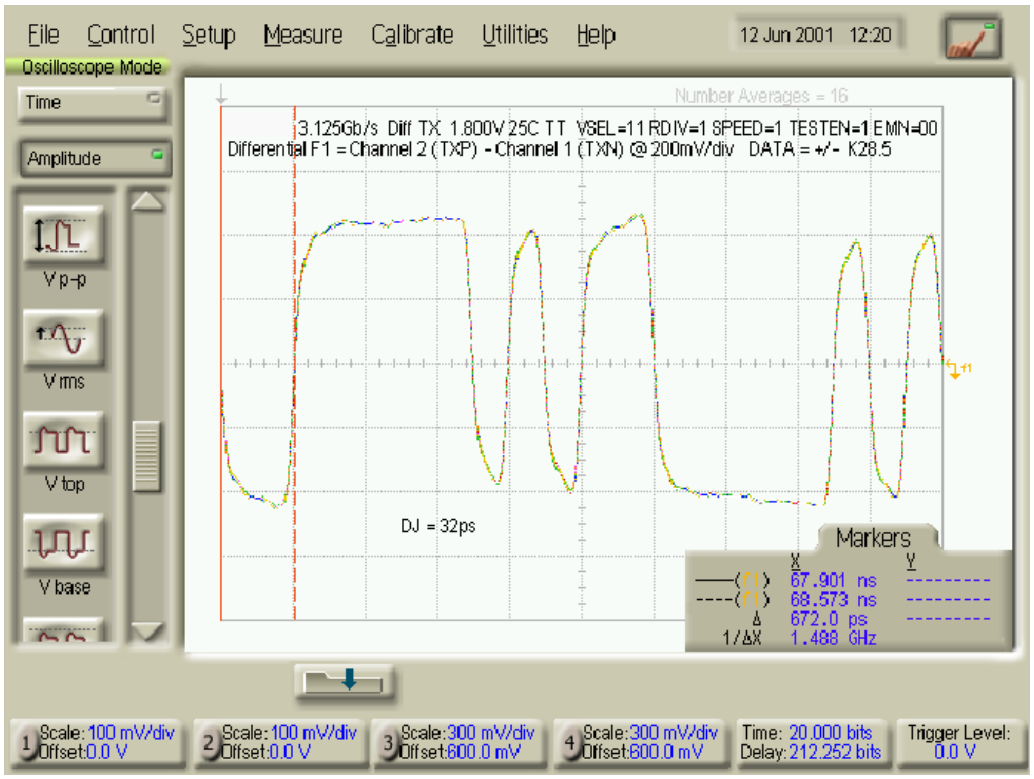
A second characteristic of Rocket I/O transceiver pre-emphasis is that the STRONG level is reduced after some time to the LOGIC level, thereby minimizing the voltage swing necessary to switch the differential pair into the opposite state.

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.

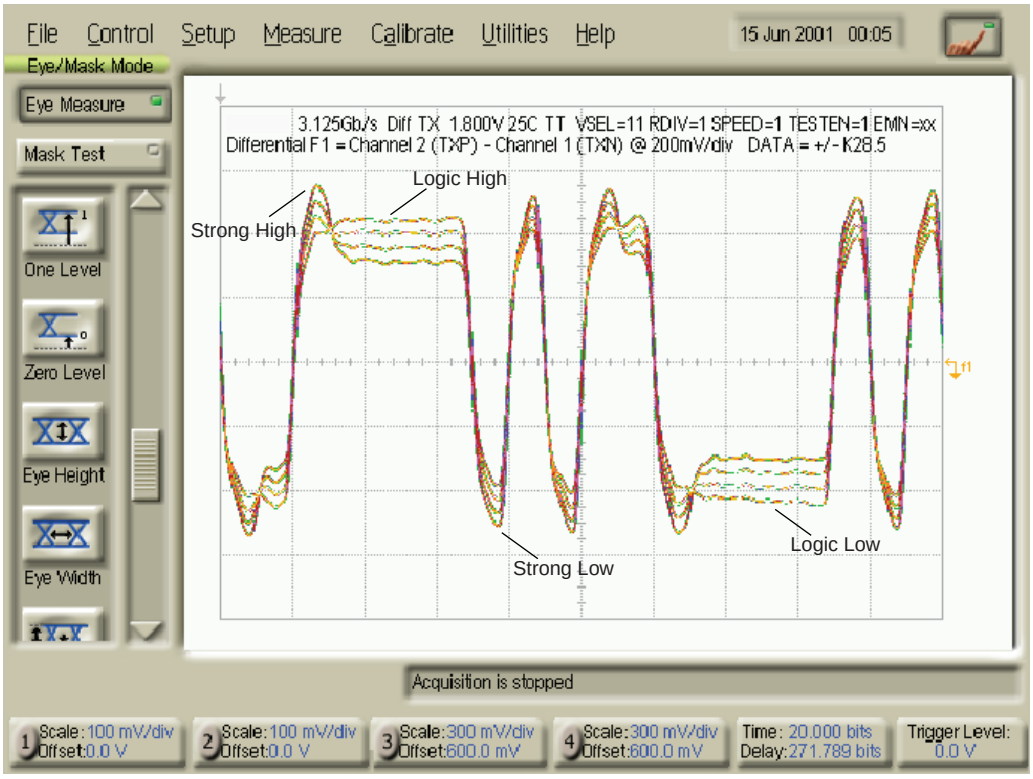
Table 4-2: **Pre-emphasis Values**

Attribute Values	Emphasis (%)
0	10
1	20
2	25
3	33



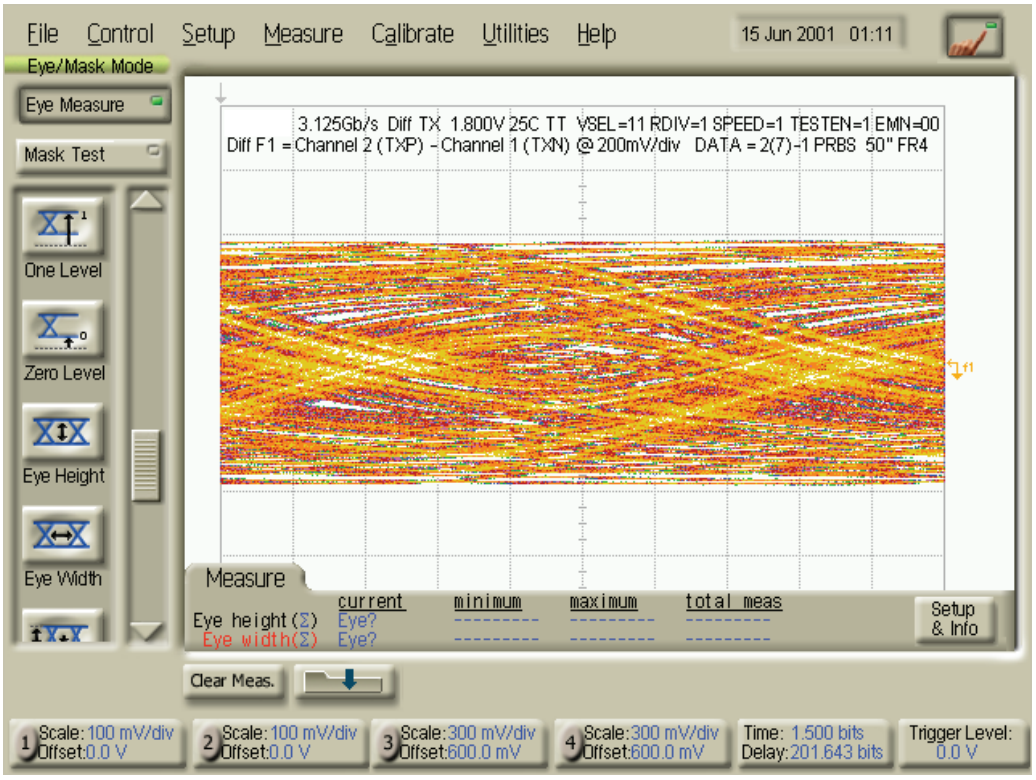
UG024_17_020802

Figure 4-2: **Alternating K28.5+ with No Pre-Emphasis**



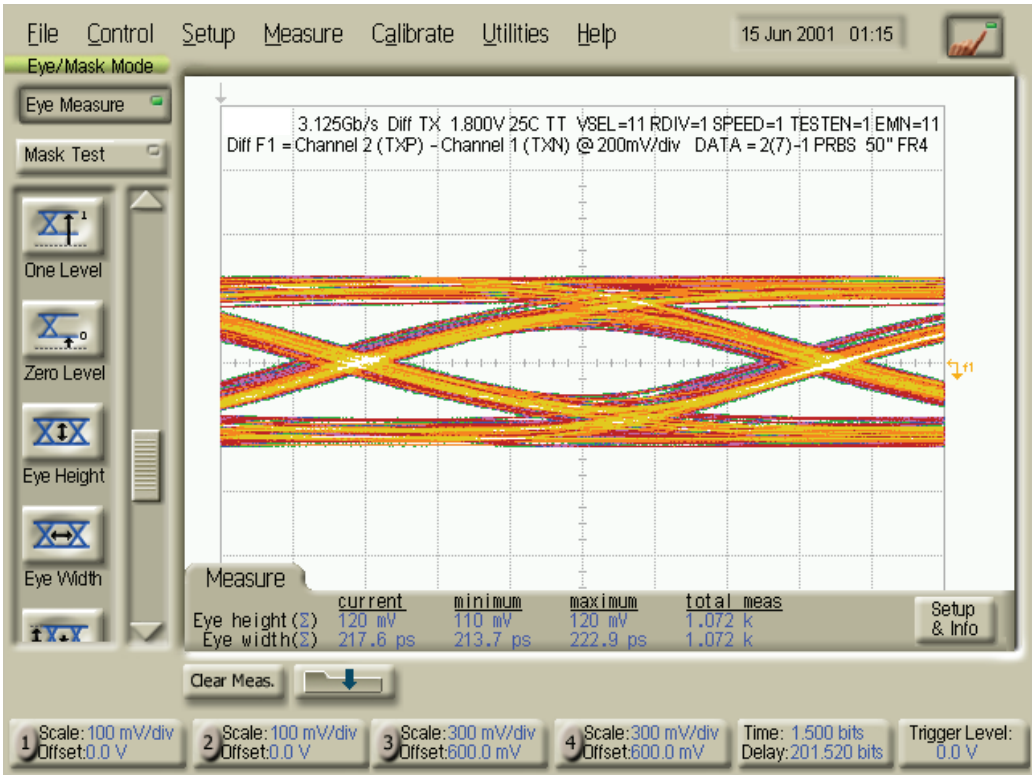
UG024_18_020802

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(1)	
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_{DCREF}	REFCLK duty cycle	45	50	55	%	
T_{RCLK}/T_{FCLK}	REFCLK rise and fall time (see Virtex-II Pro Data Sheet, Module 3)			75	ps	Between 20% and 80% voltage levels
T_{GJTT}	REFCLK total jitter			40	ps	Peak-to-peak
T_{LOCK}	Clock recovery frequency acquisition time		10		µs	
T_{UNLOCK}					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
AVCCAUXTX	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 AVCCAUXTX.

Refer to the manufacturer's Web page at http://www.linear-tech.com for further information about this device.

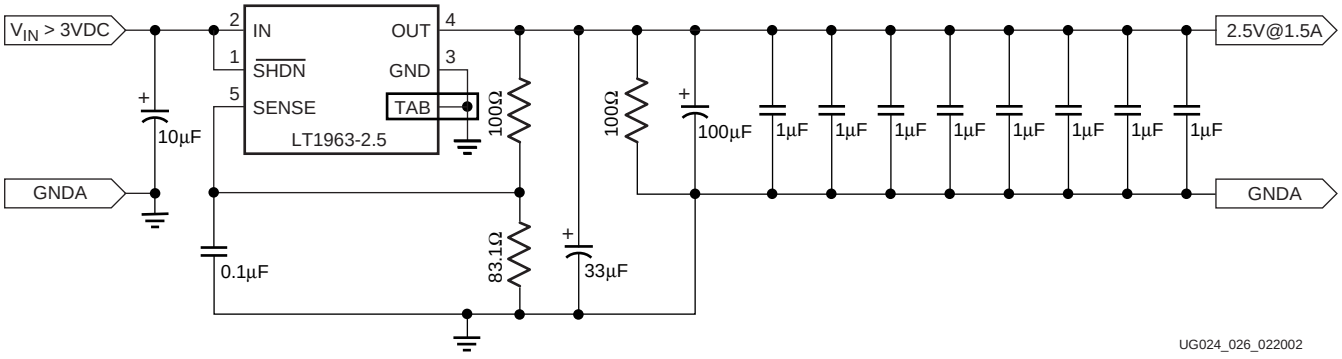


Figure 4-6: **Power Supply Circuit Using LT1963 Regulator**

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.

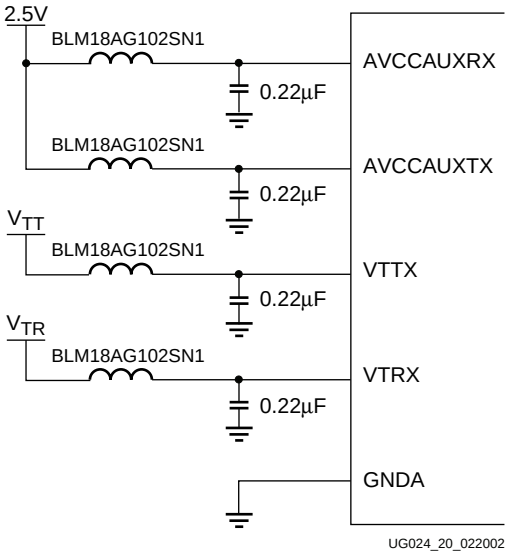


Figure 4-7: **Power Filtering Network for One Transceiver**

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 AVCCAUXTX 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.

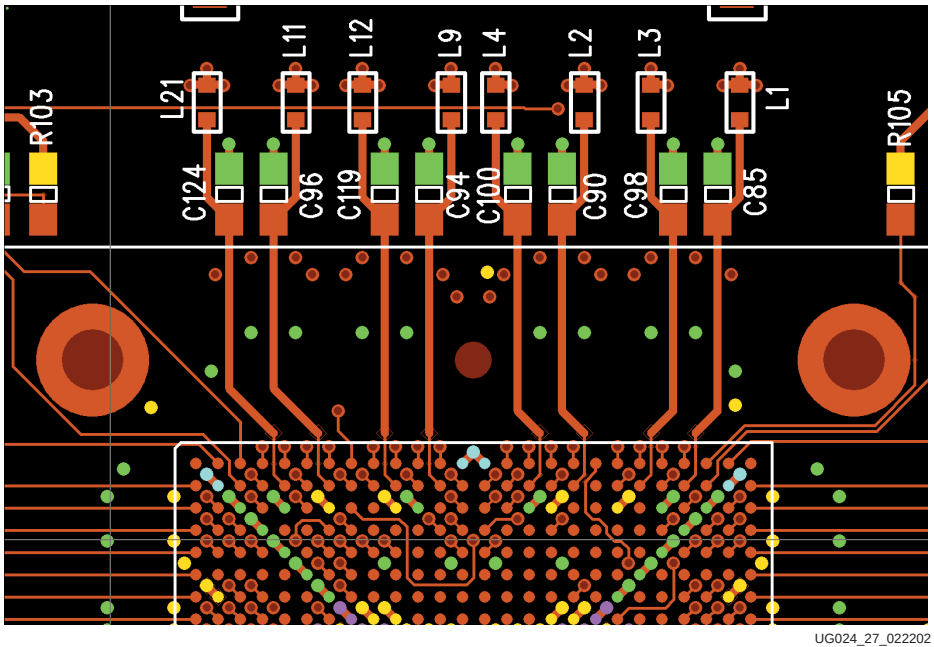


Figure 4-8: **Example Power Filtering PCB Layout for Four MGTs, Top Layer**

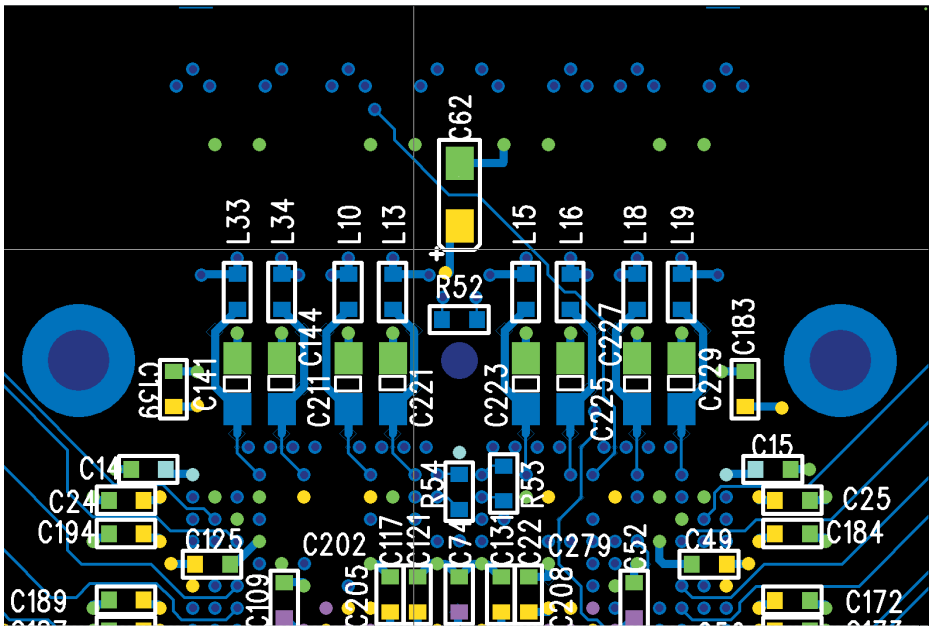


Figure 4-9: **Example Power Filtering PCB Layout for Four MGTs, Bottom Layer**

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 µF 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 100Ω or 150Ω differential trace impedance (depending on whether the 50Ω or 75Ω 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.

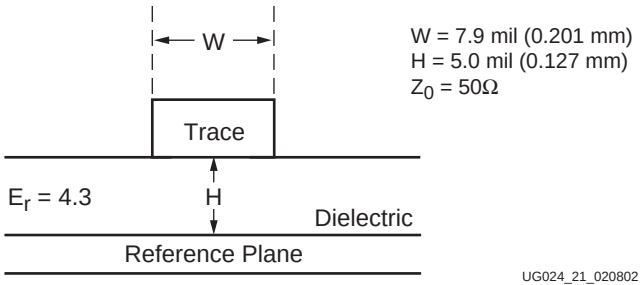


Figure 4-10: **Single-Ended Trace Geometry**

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 100Ω differential impedance.

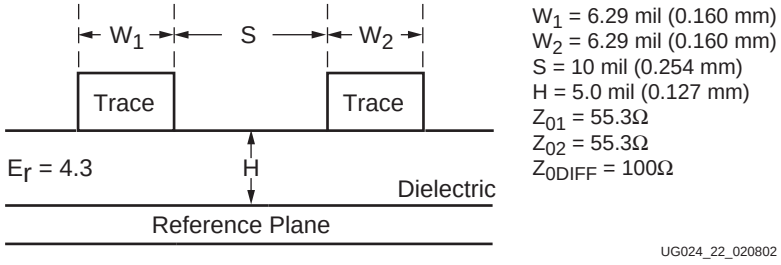


Figure 4-11: **Microstrip Edge-Coupled Differential Pair**

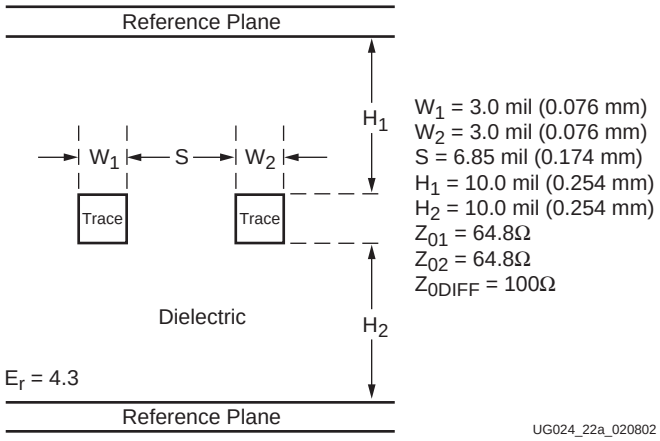


Figure 4-12: **Stripline Edge-Coupled Differential Pair**

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.

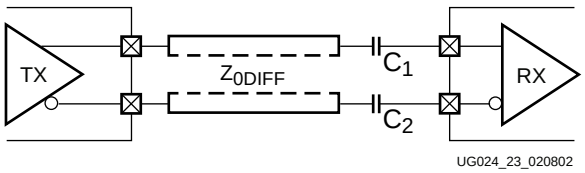


Figure 4-13: **AC-Coupled Serial Link**

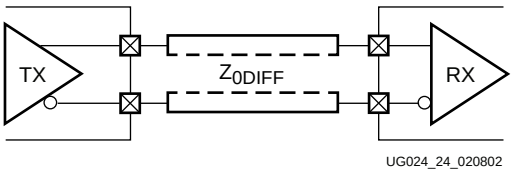


Figure 4-14: **DC-Coupled Serial Link**

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.

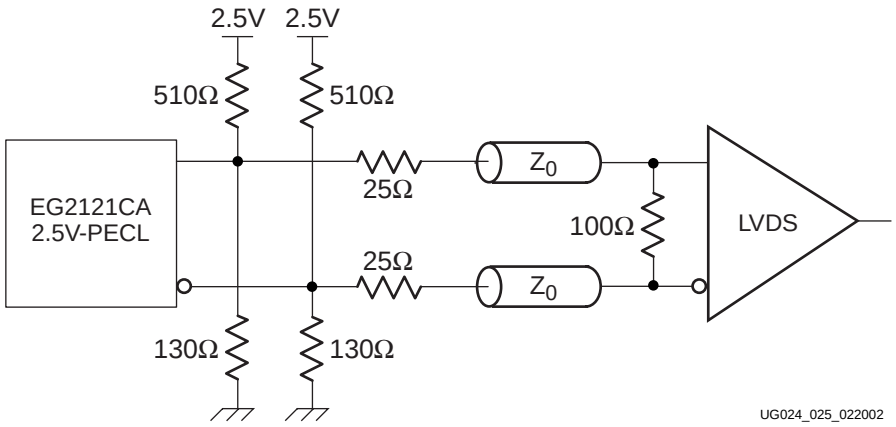


Figure 4-15: **Reference CLock Oscillator Interface**

XILINX®

Chapter 5

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.

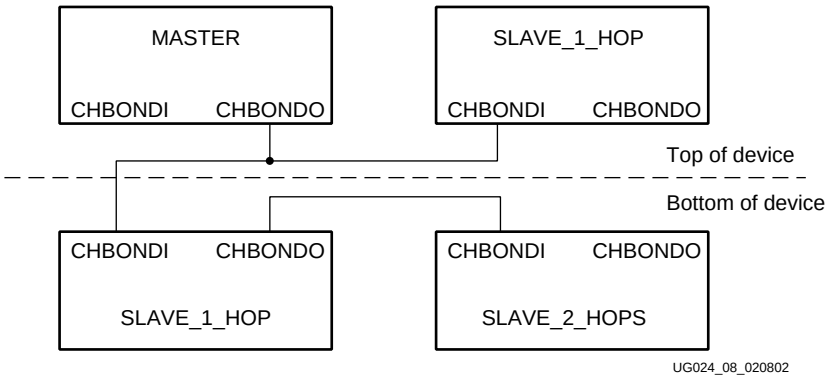


Figure 5-1: **2VP2 Implementation**

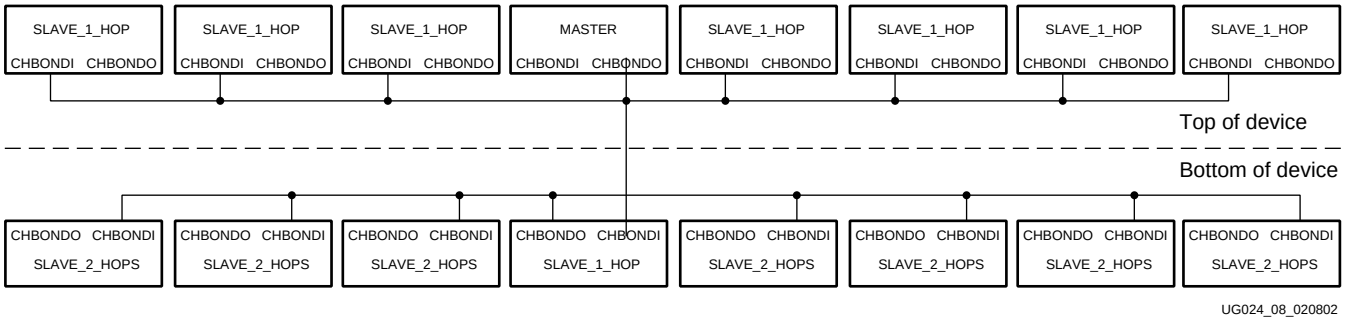


Figure 5-2: **2VP50 Implementation**

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 ns – 0.2 ns – 2.0 ns = 4.2 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.

XILINX®

Appendix A

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
                      );
```