PowerPC Processor Reference Guide

Embedded Development Kit

EDK (v3.1 EA) September 16, 2002

XILINX®

PowerPC Processor Reference Guide
EDK (v3.1 EA) September 16, 2002

The following table shows the revision history for this document..

	Version	Revision
09/16/02	1.0	Initial Embedded Develpment Kit (EDK) release.

Table of Contents

Chapter 3: User Programming Model

Chapter 4: PPC405 Privileged-Mode Programming Model

Chapter 5: Memory-System Management

Chapter 6: Virtual-Memory Management

Chapter 9: Debugging

Chapter 10: Reset and Initialization

Chapter 11: Instruction Set

Appendix A: Register Summary

About This Guide

This guide 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.

Table P-1: **General Notational Conventions *(Continued)***

Convention	Definition
cr_bit	Used in simplified mnemonics to specify a CR-bit position (0 to 31) used as an operand.
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 69
crbA	11:15	Specifies a bit in the CR used as a source of a CR-logical instruction.

Table P-2: **Instruction Field Definitions *(Continued)***

Field	Location	Description
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.
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.

Table P-2: **Instruction Field Definitions *(Continued)***

Field	Location	Description
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 63 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.
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
←	Assignment
∧	AND logical operator
¬	NOT logical operator
∨	OR logical operator
⊕	Exclusive-OR (XOR) logical operator
+	Two's-complement addition

Table P-3: **Pseudocode Conventions *(Continued)***

Convention	Definition
–	Two's-complement subtraction, unary minus
×	Multiplication
÷	Division yielding a quotient
%	Remainder of an integer division. For example, (33 % 32) = 1.
‖	Concatenation
=, ≠	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.
^{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.

Table P-3: **Pseudocode Conventions *(Continued)***

Convention	Definition
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.
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
¬, – (unary minus)	Right to left
×, ÷	Left to right
+, –	Left to right
‖	Left to right
=, ≠, <, >, $\overset{u}{<}$, $\overset{u}{>}$	Left to right
∧, ⊕	Left to right
∨	Left to right
←	None

Registers

Table 5 lists the PPC405 registers and their descriptive names.

Table P-5: **PPC405 Registers**

Register	Descriptive Name
CCR0	Core-configuration register 0
CR	Condition register
CTR	Count register
DAC*n*	Data-address compare *n*
DBCR*n*	Debug-control register *n*
DBSR	Debug-status register
DCCR	Data-cache cacheability register
DCWR	Data-cache write-through register
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

Table P-5: **PPC405 Registers *(Continued)***

Register	Descriptive Name
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 13, 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.

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.

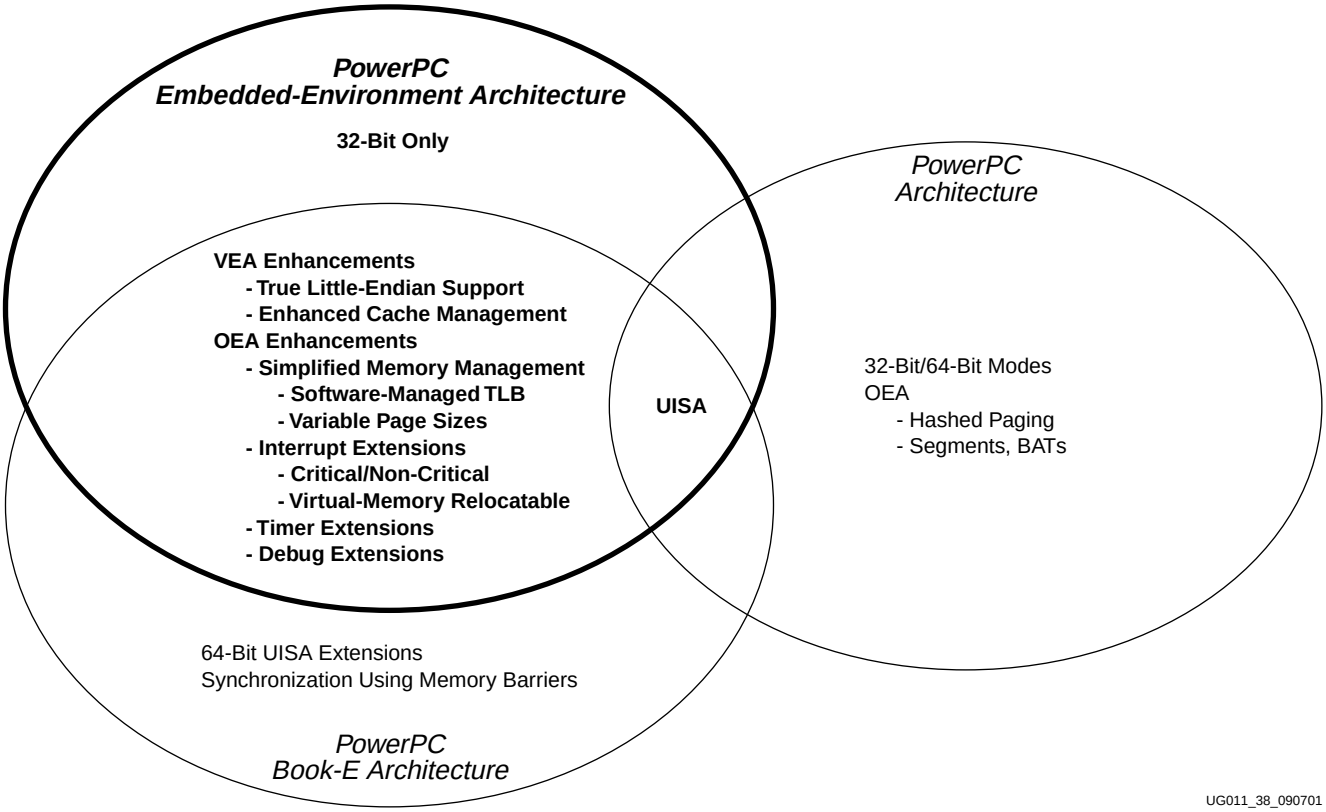


Figure 1-1: **Relationship of PowerPC Architectures**

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 45.

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 80.

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 74.

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

Register Set Summary

Figure 1-2, page 36 shows the registers contained in the PPC405. Descriptions of the registers are in the following sections.

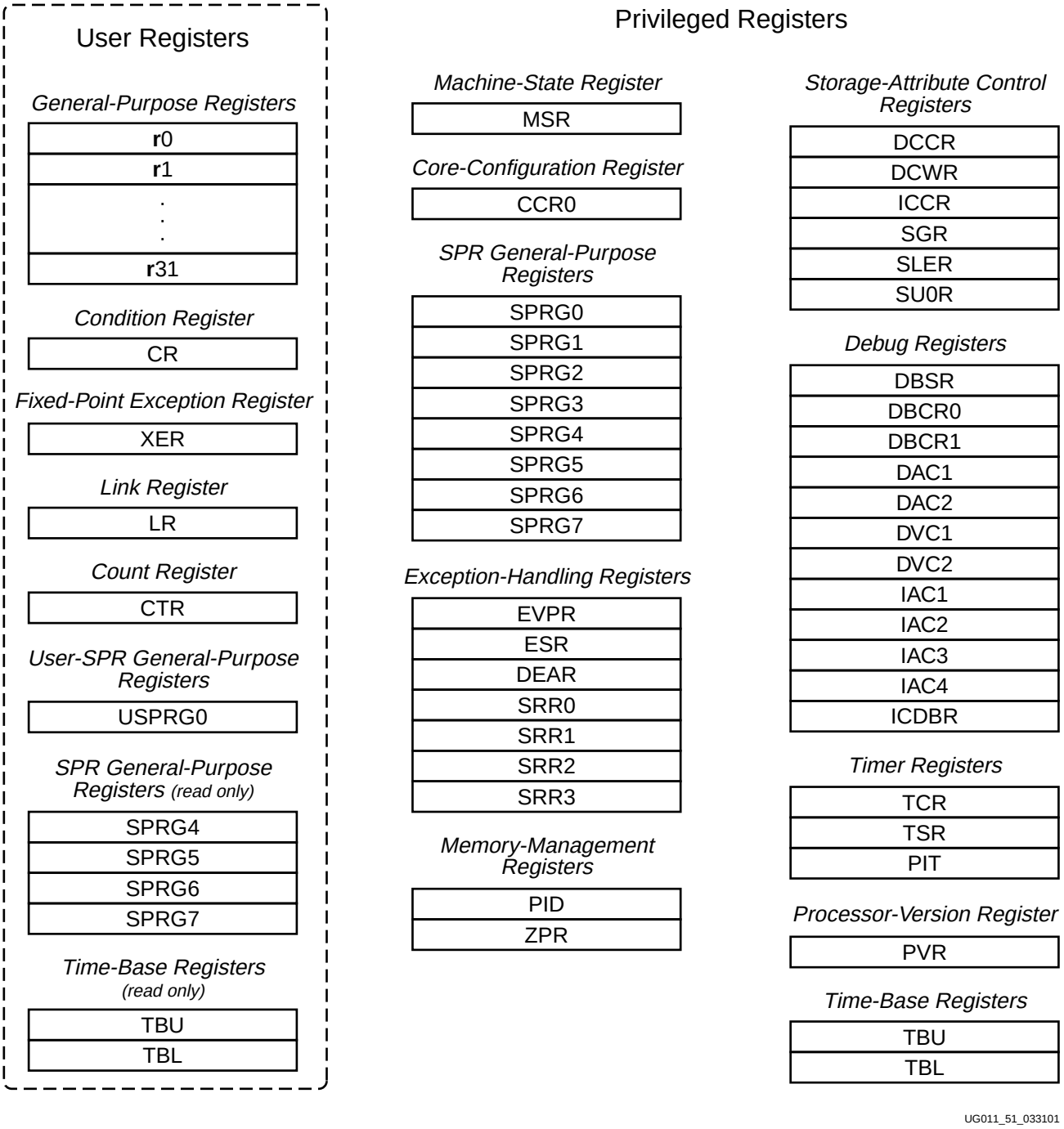


Figure 1-2: **PPC405 Registers**

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 62, 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 61, and **Privileged Registers**, page 131 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 133, 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 63, 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 ***PowerPC® 405 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.

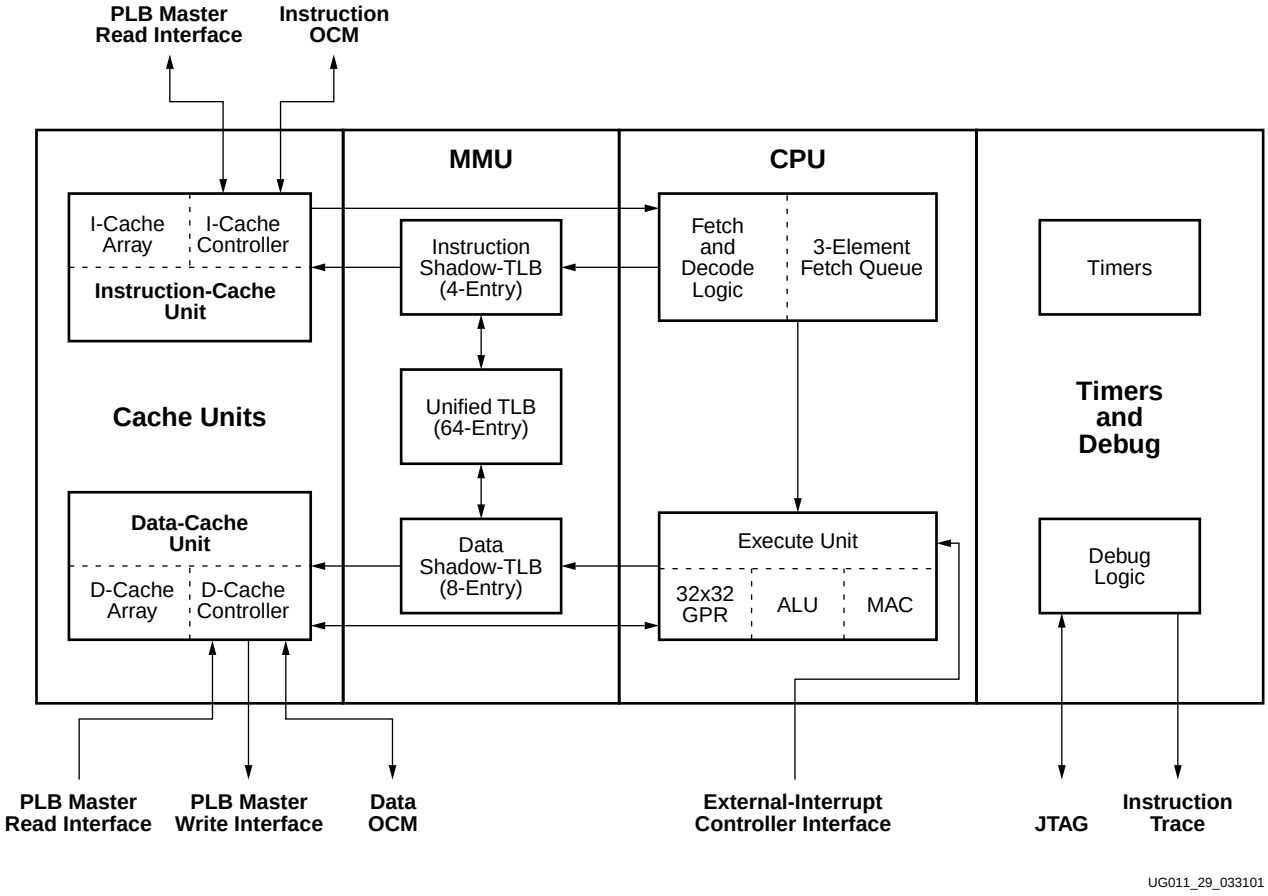


Figure 1-3: **PPC405 Organization**

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 72.

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 107 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 162 . For information on the hardware operation, use, and electrical characteristics of these interfaces, refer to the ***PowerPC® 405 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.

XILINX®

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 151, 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 126. 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 126.

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 133 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 215).

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 132. 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 136. 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 62. 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 136.

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**.

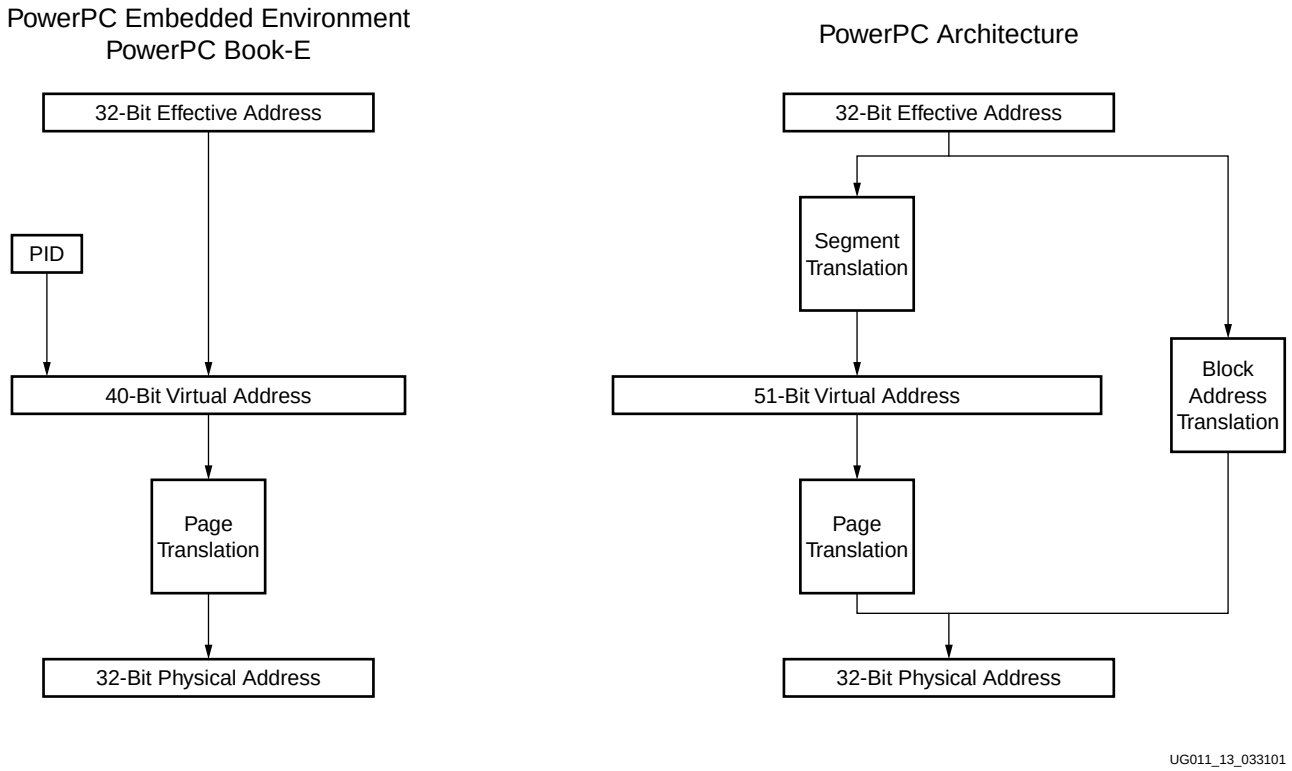


Figure 2-1: **PowerPC 32-Bit Memory Management**

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 174 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 48. 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.

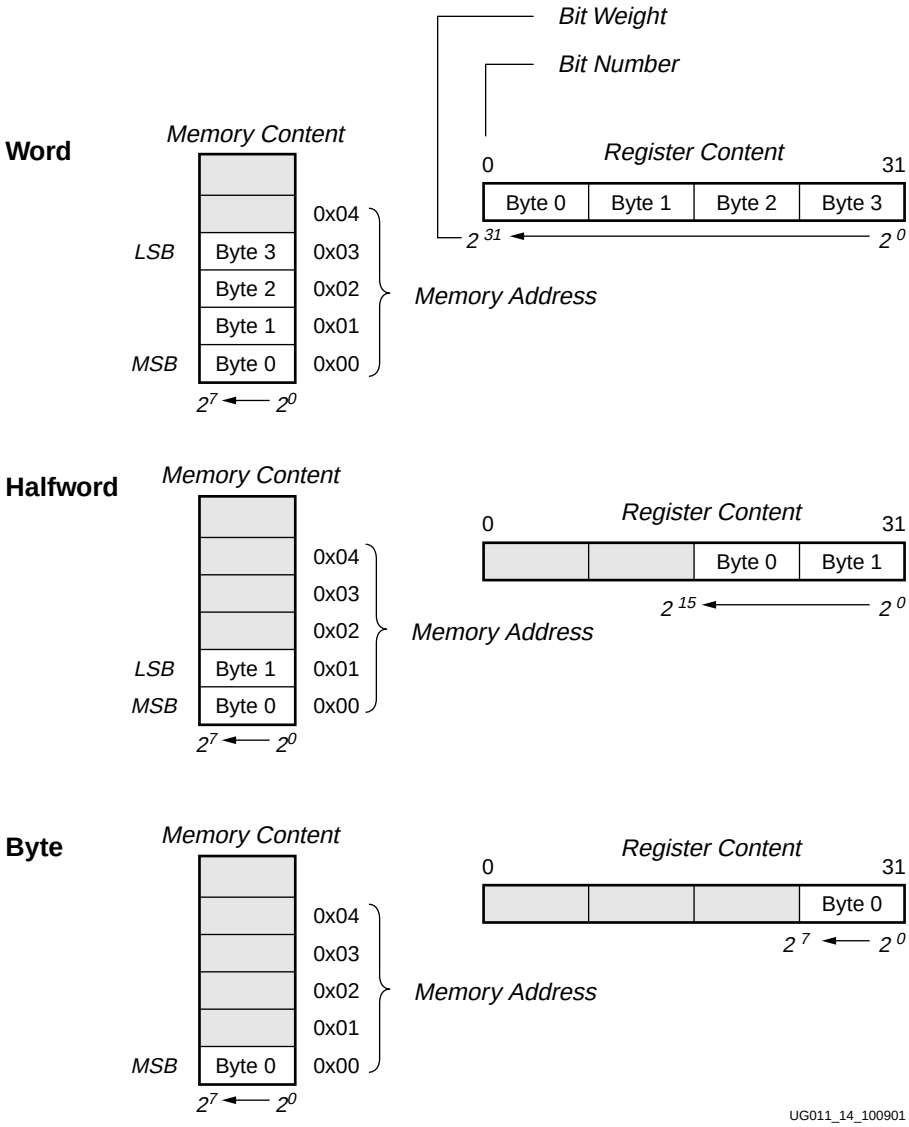


Figure 2-2: **Operand Data Types**

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 158 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 159 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 51, 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 87, 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 214 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 492. The **Instruction Set Information**, page 497 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 481. 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.

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

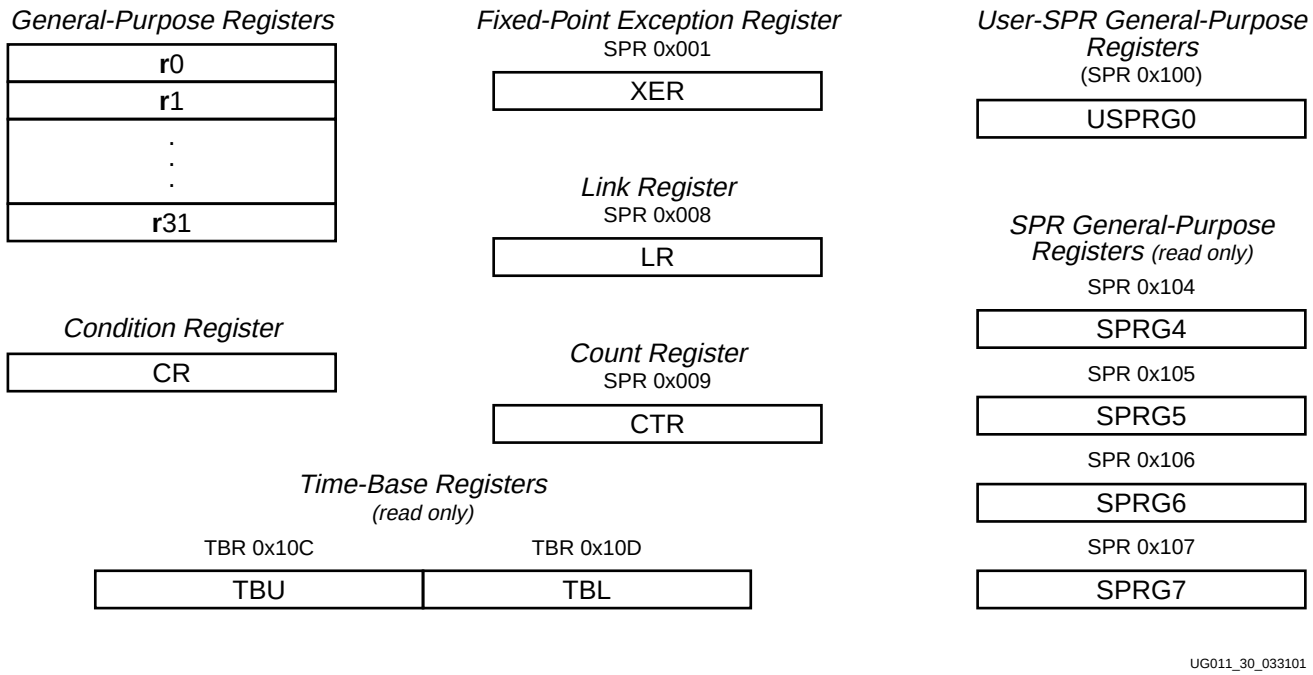


Figure 3-1: **PPC405 User Registers**

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 126 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 470 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 530 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.

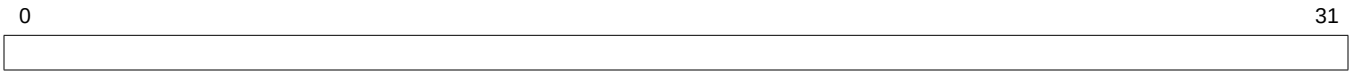


Figure 3-2: **General Purpose Registers (R0-R31)**

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.

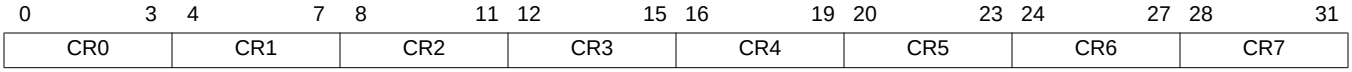


Figure 3-3: **Condition Register (CR)**

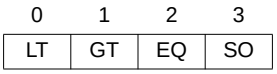


Figure 3-4: **CR*n* Field**

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.

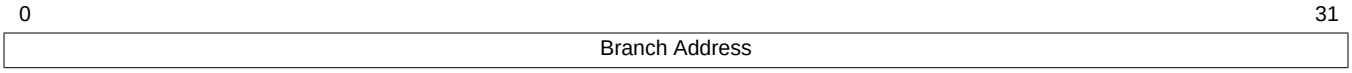


Figure 3-6: **Link Register (LR)**

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 73 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 69 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.

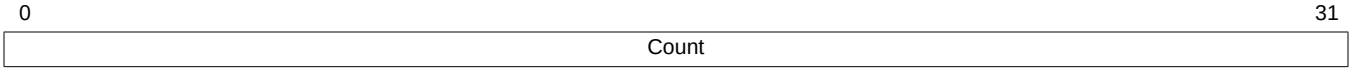


Figure 3-7: **Count Register (CTR)**

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.

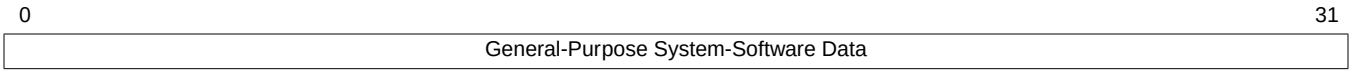


Figure 3-9: **SPR General-Purpose Registers (SPRG4–SPRG7)**

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 134).

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.

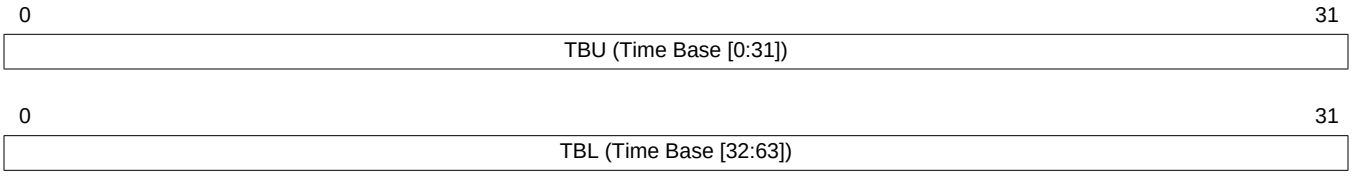


Figure 3-10: **Time-Base Register**

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 228, 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 72.

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 521 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 70. 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 70. 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 70. 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.

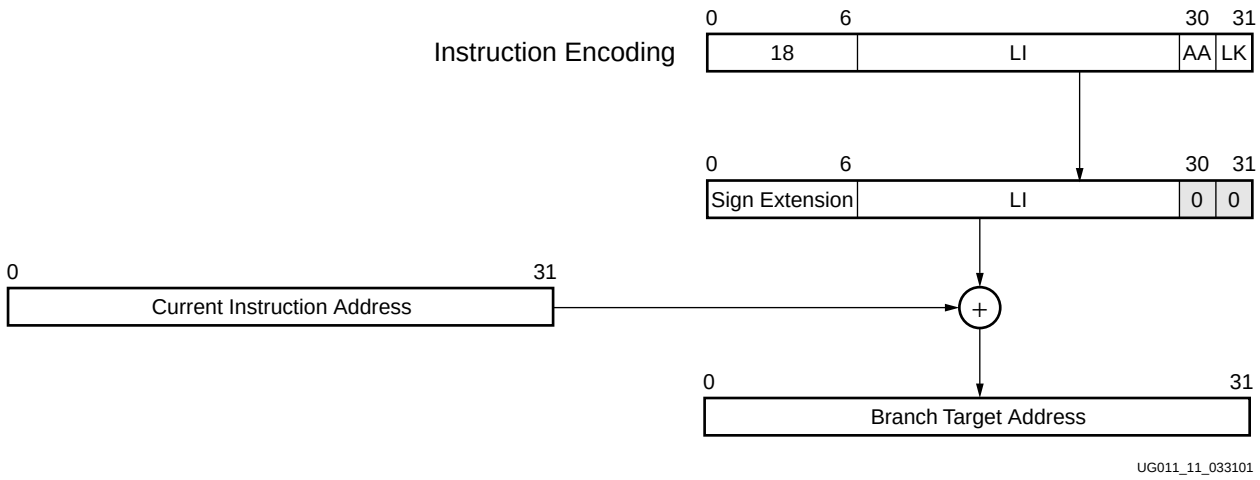


Figure 3-11: **Branch-to-Relative Addressing**

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.

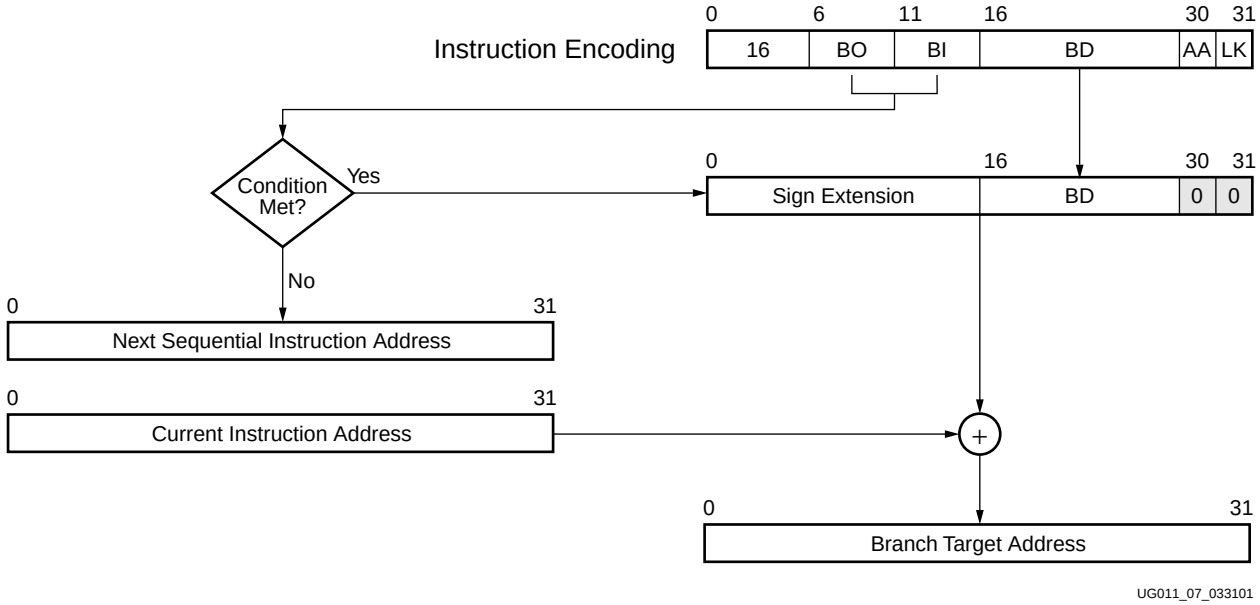


Figure 3-12: **Branch-Conditional to Relative Addressing**

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.

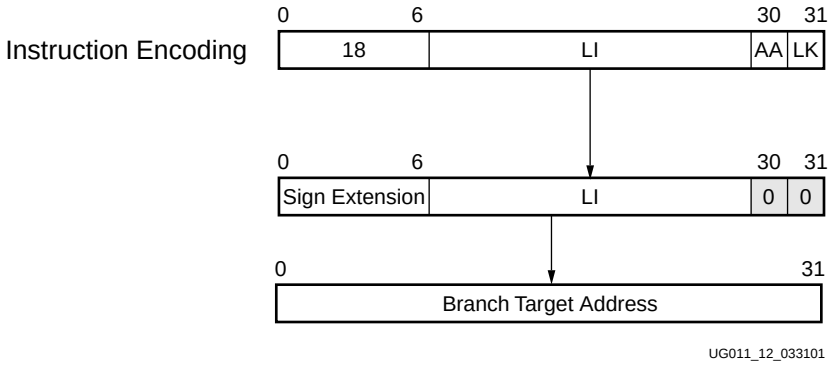


Figure 3-13: **Branch-to-Absolute Addressing**

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.

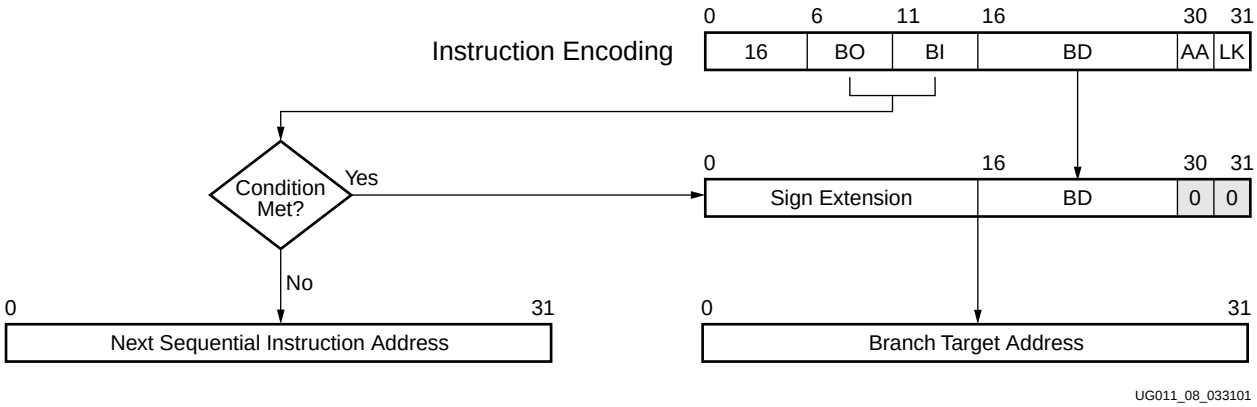


Figure 3-14: **Branch-Conditional to Absolute Addressing**

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.

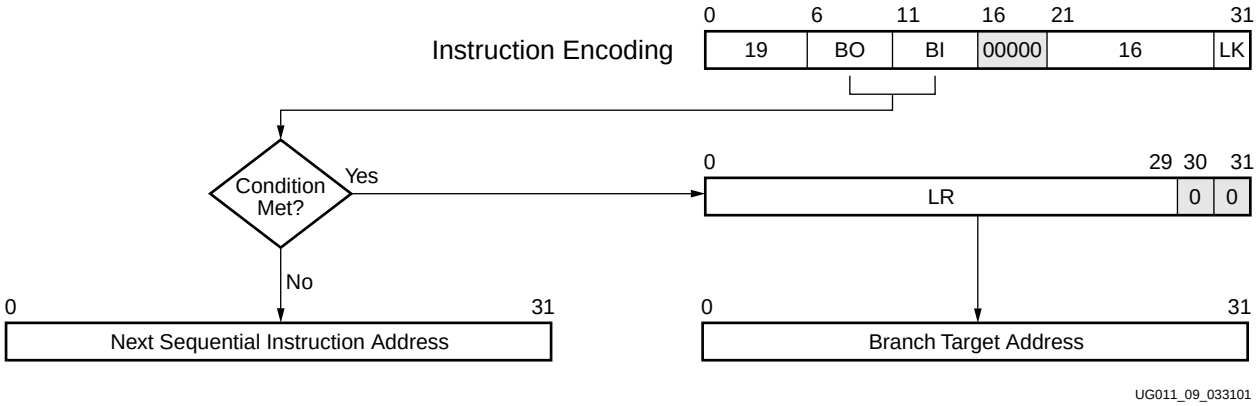


Figure 3-15: **Branch-Conditional to Link-Register Addressing**

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.

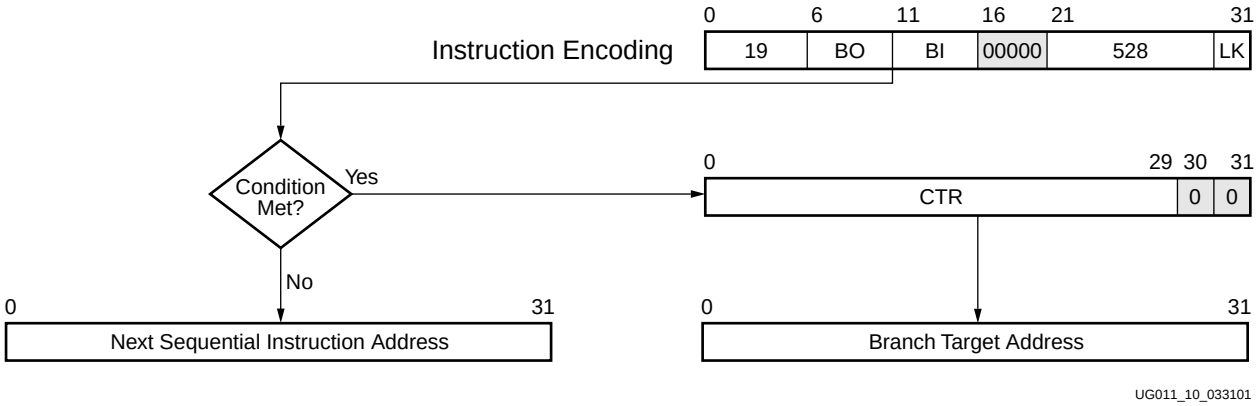


Figure 3-16: **Branch-Conditional to Count-Register Addressing**

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 528 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 218 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 215 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 250 for more information.

Simplified mnemonics are defined for the system-trap instructions. See **Trap Instructions**, page 532 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 126 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 170 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 55 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.

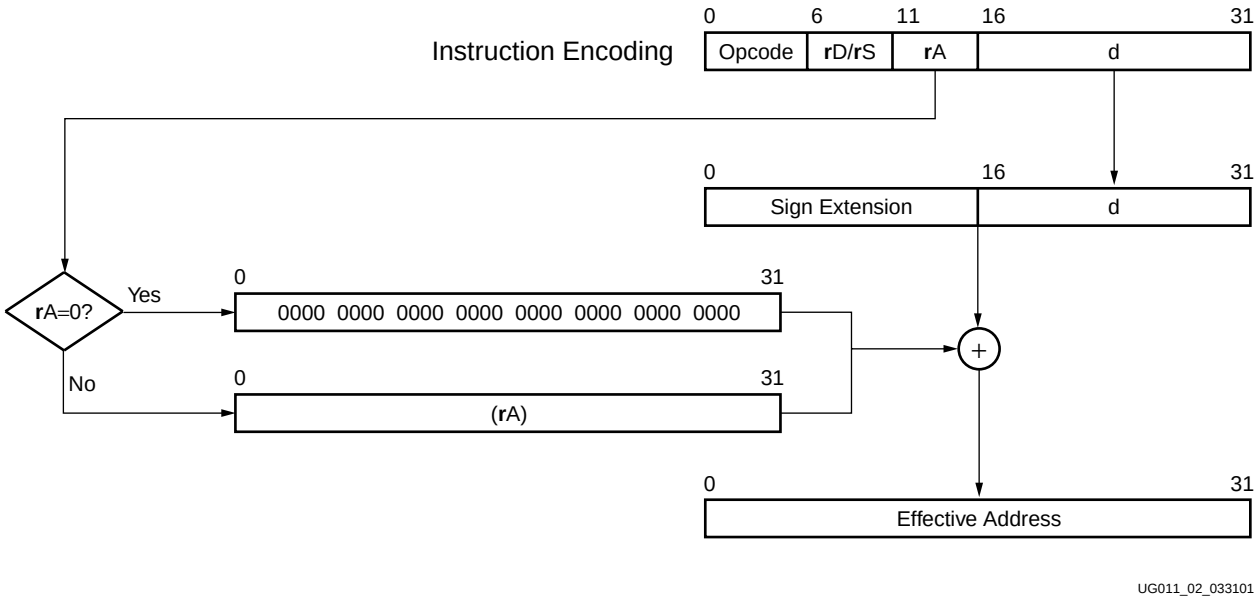


Figure 3-17: **Register-Indirect with Immediate-Index Addressing**

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.

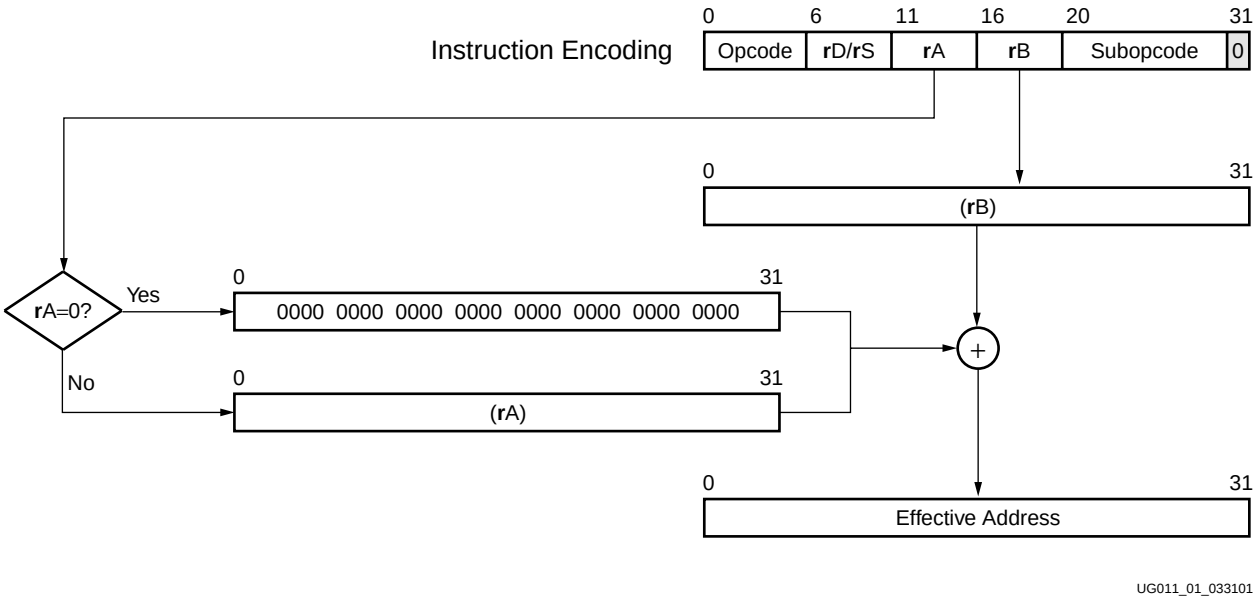


Figure 3-18: **Register-Indirect with Index Addressing**

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.

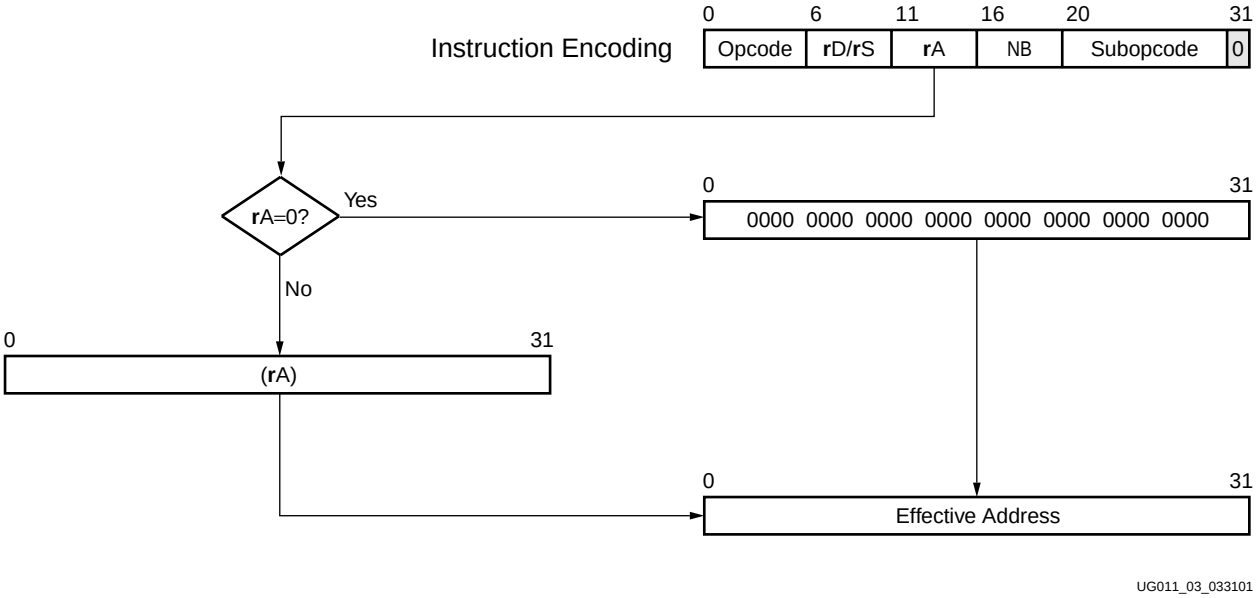


Figure 3-19: **Register-Indirect Addressing**

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 51.

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		

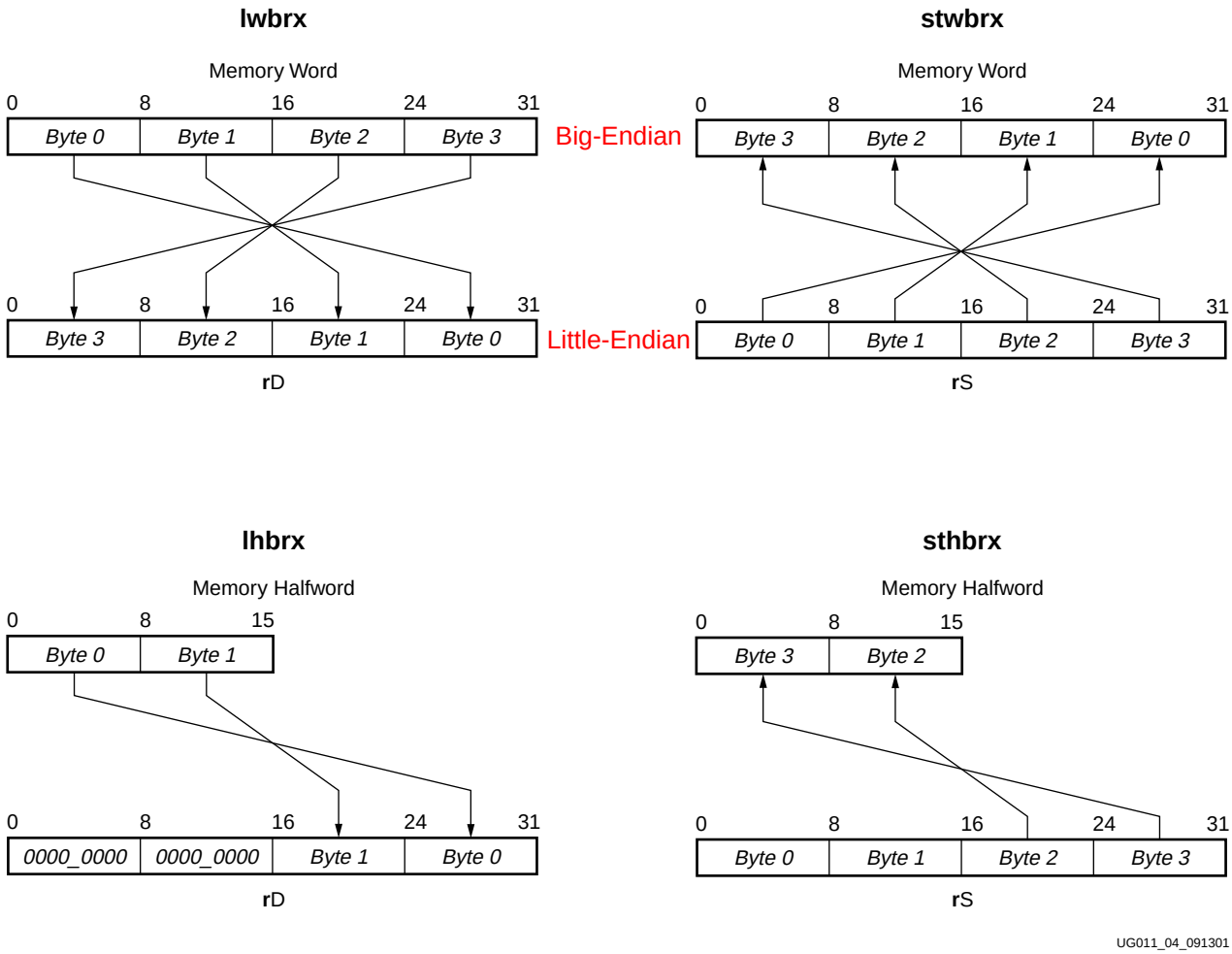


Figure 3-20: **Load and Store with Byte-Reverse Instructions**

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)

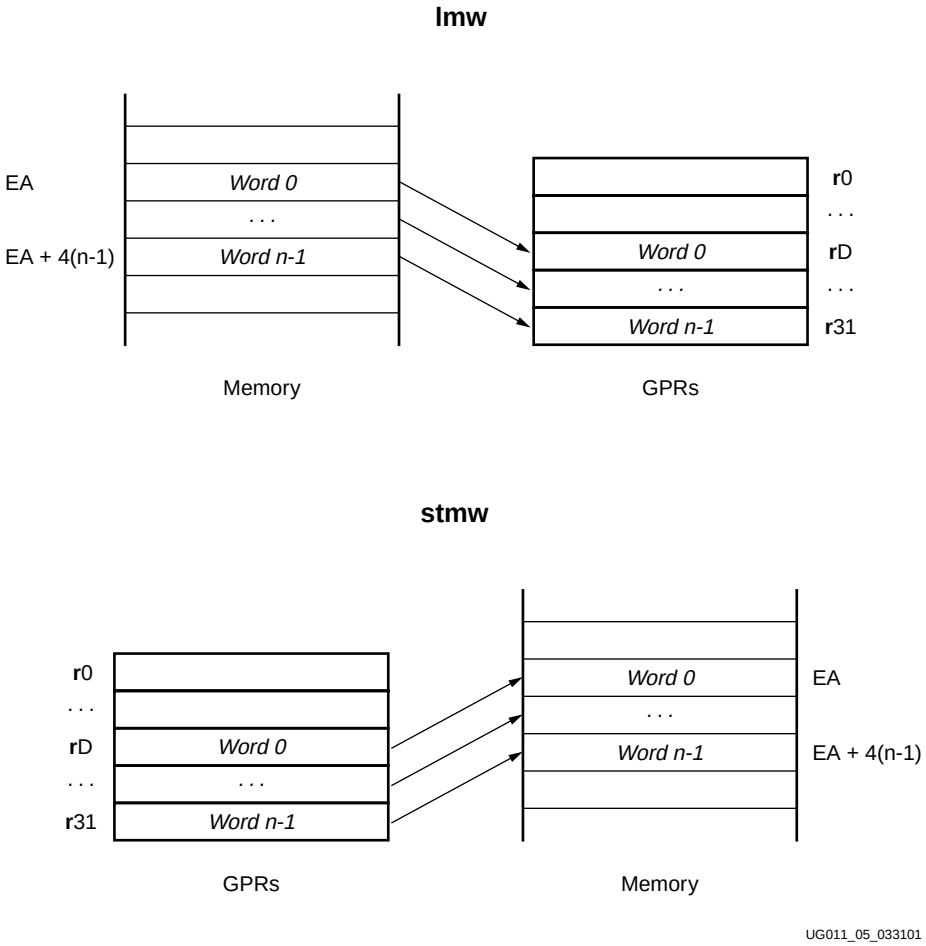


Figure 3-21: **Load and Store Multiple Instructions**

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 55 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.

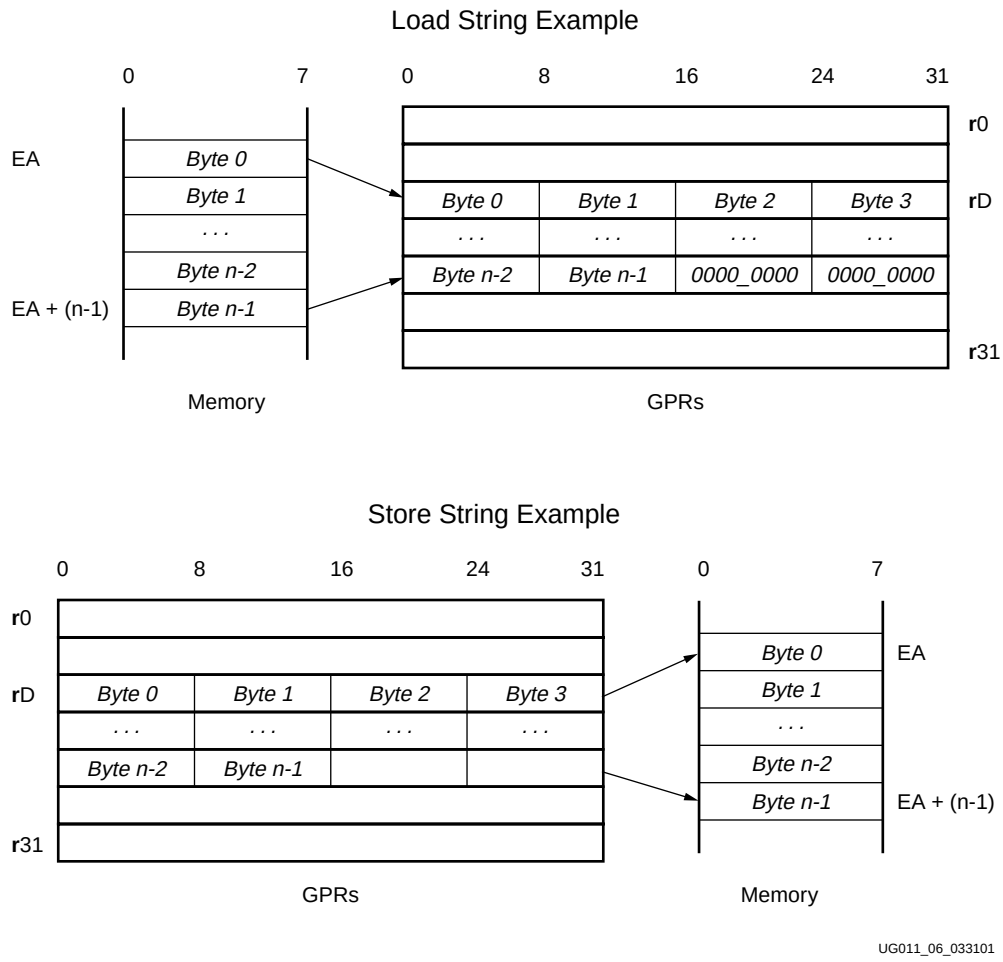


Figure 3-22: **Load and Store String Instructions**

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 63, 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 65 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 531 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 534 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 64 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 528 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 529 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.

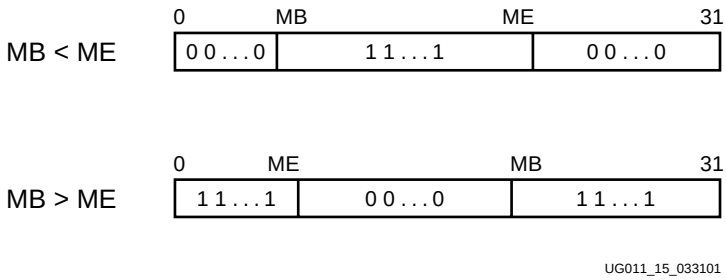


Figure 3-23: **Rotate Mask Generation**

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]).

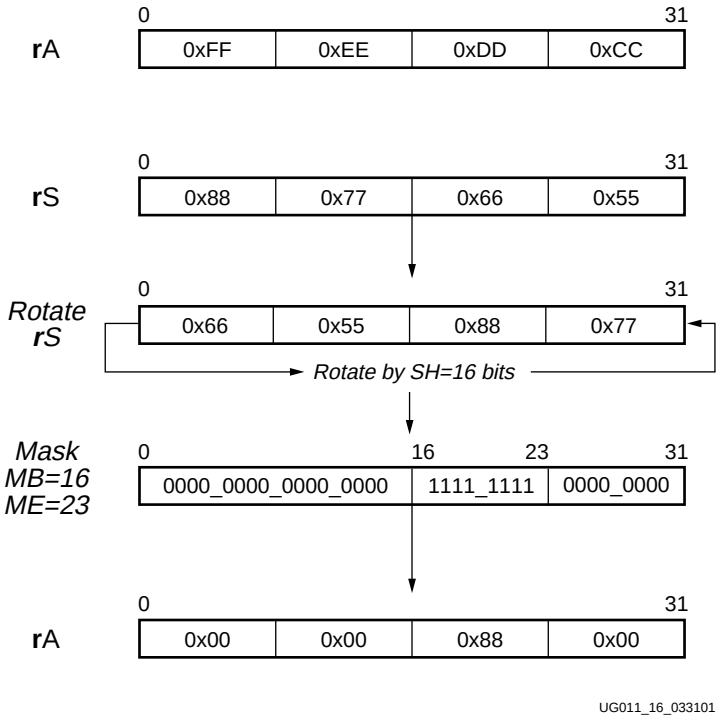


Figure 3-24: **Rotate Left then AND-with-Mask Immediate Example**

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		**T**he 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.

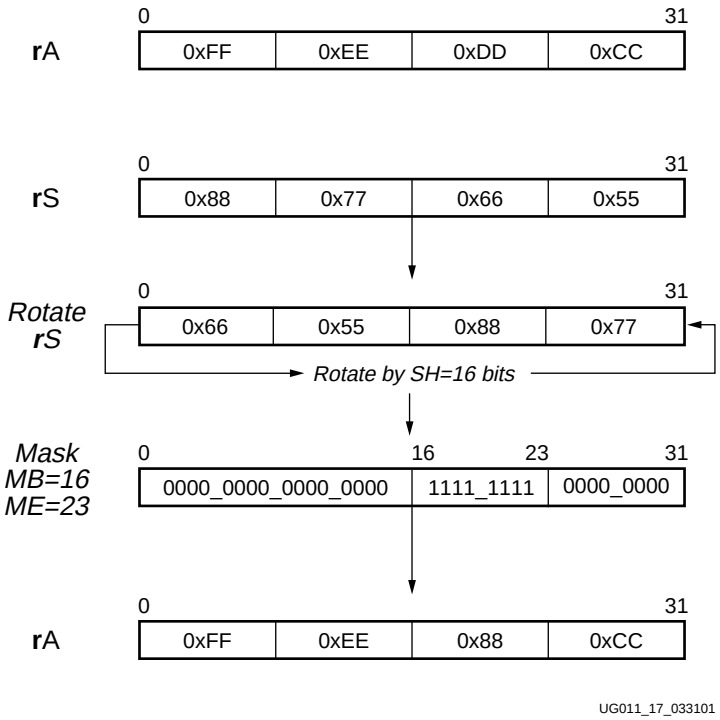


Figure 3-25: **Rotate Left then Mask-Insert Immediate Example**

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 529 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.

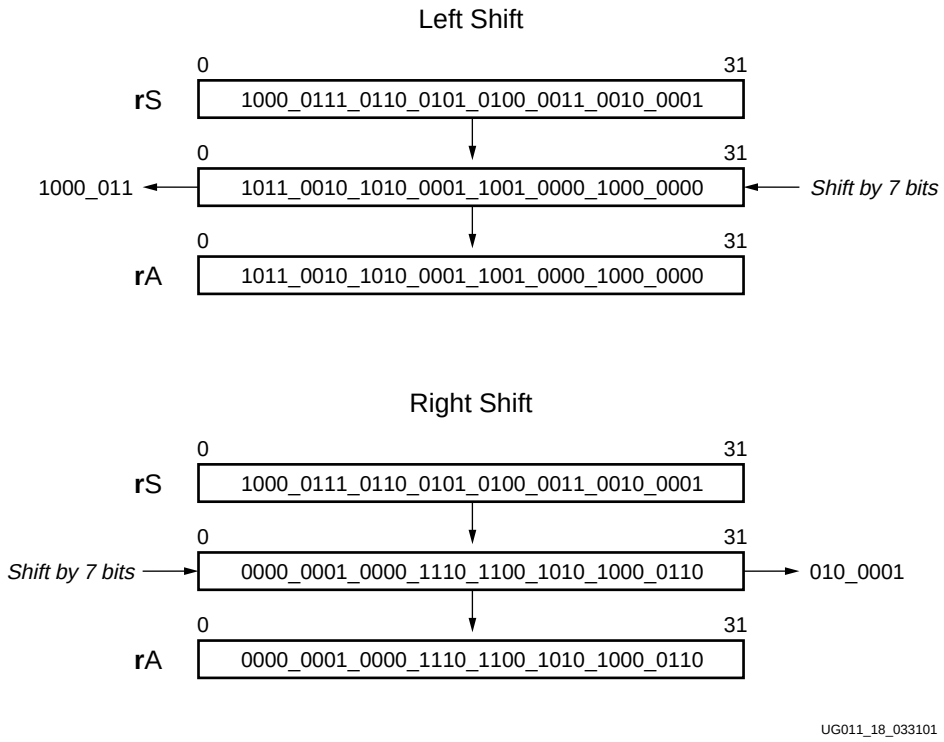


Figure 3-26: **Logical-Shift Examples**

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 540, 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.

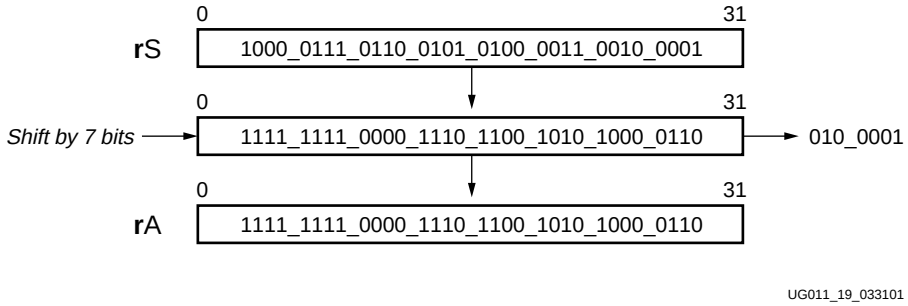


Figure 3-27: **Algebraic-Shift Example**

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.

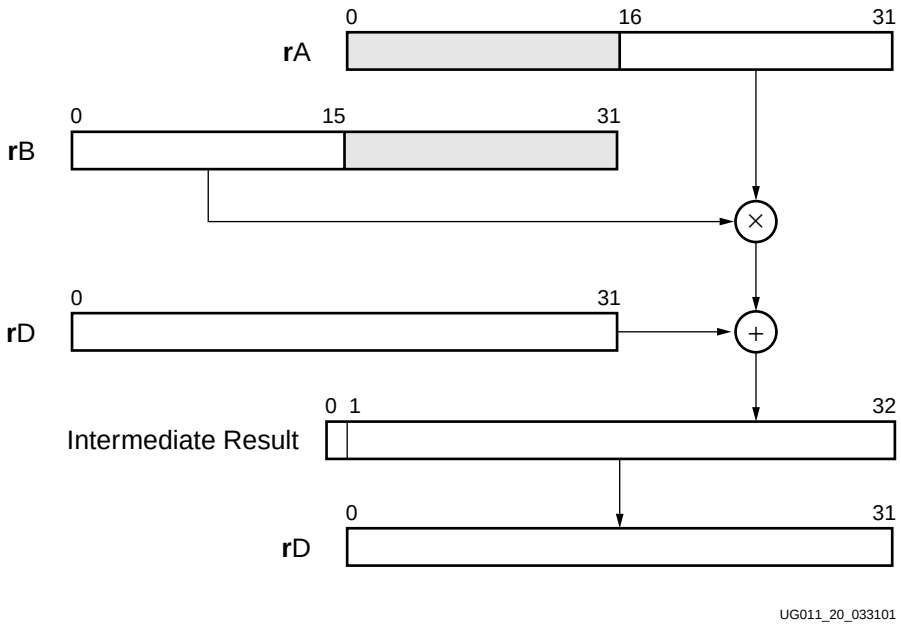


Figure 3-28: **Multiply-Accumulate Cross-Halfword to Word Operation**

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 107).

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.

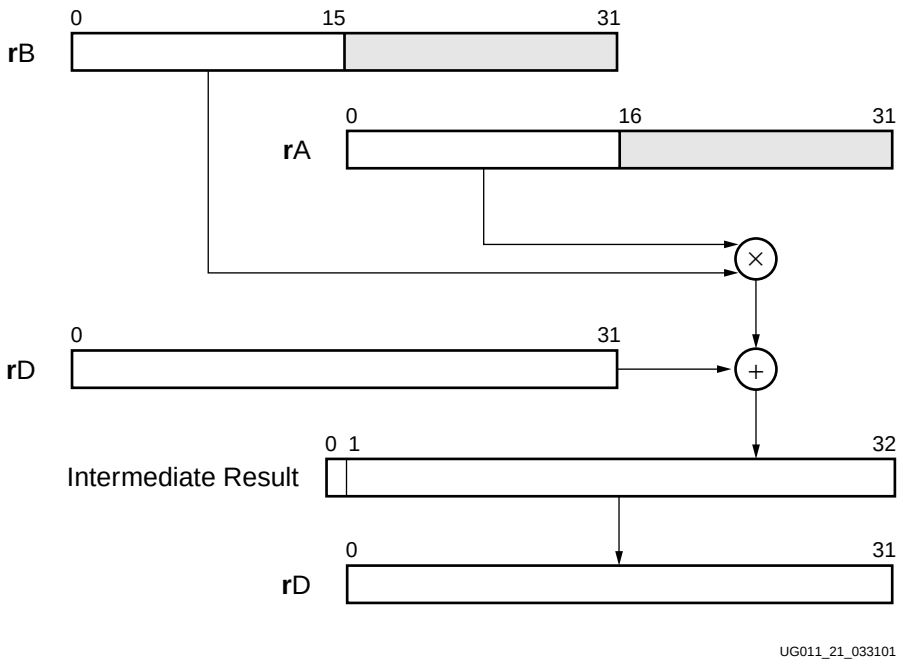


Figure 3-29: **Multiply-Accumulate High-Halfword to Word Operation**

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 107).

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.

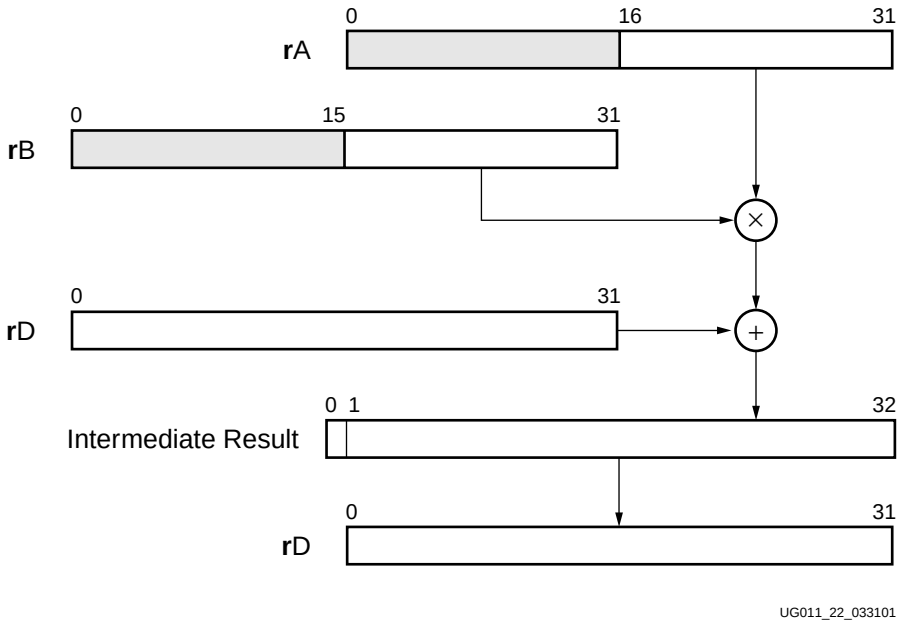


Figure 3-30: **Multiply-Accumulate Low-Halfword to Word Operation**

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.

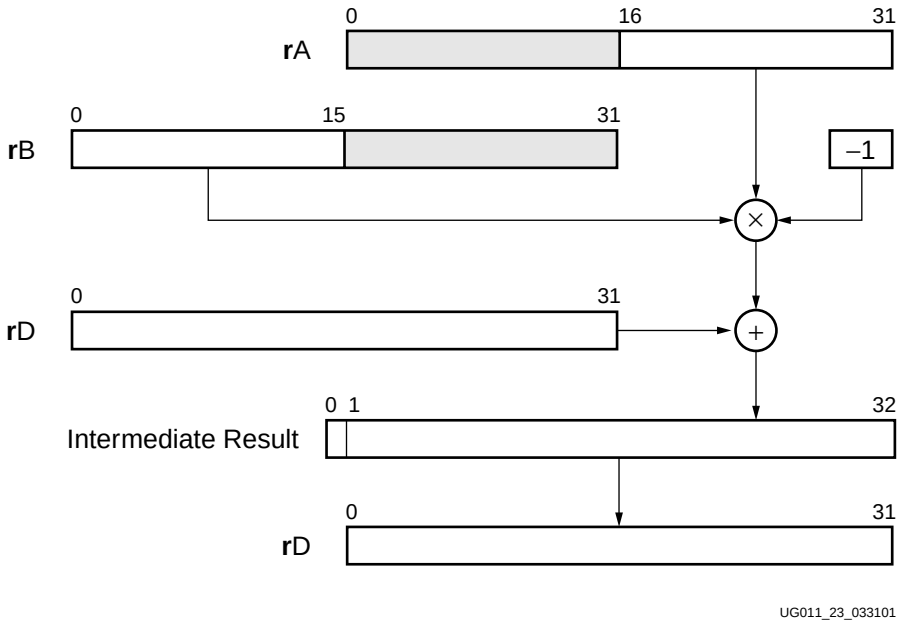


Figure 3-31: **Negative Multiply-Accumulate Cross-Halfword to Word Operation**

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 107).

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.

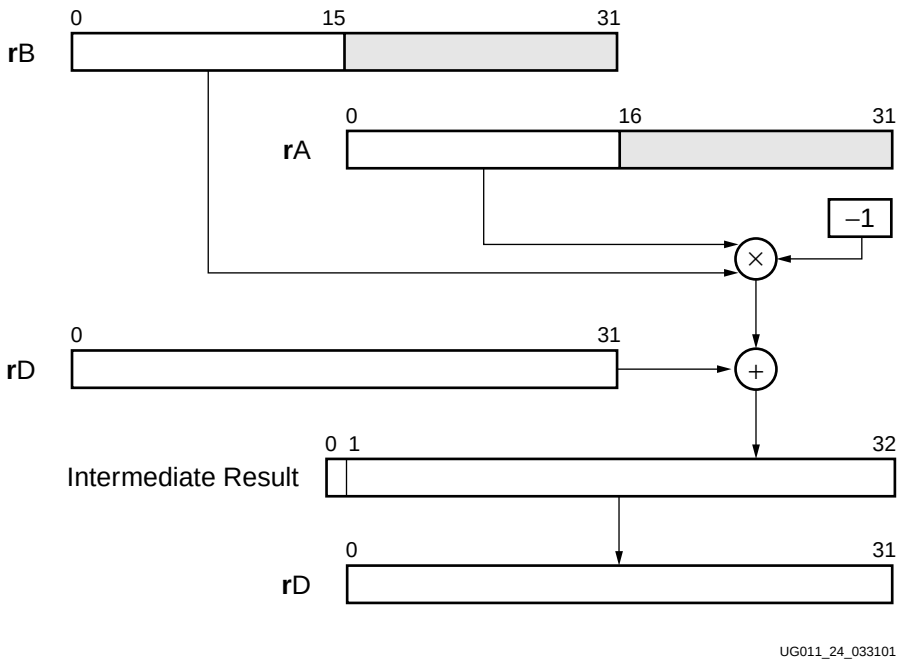


Figure 3-32: **Negative Multiply-Accumulate High-Halfword to Word Operation**

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 107).

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.

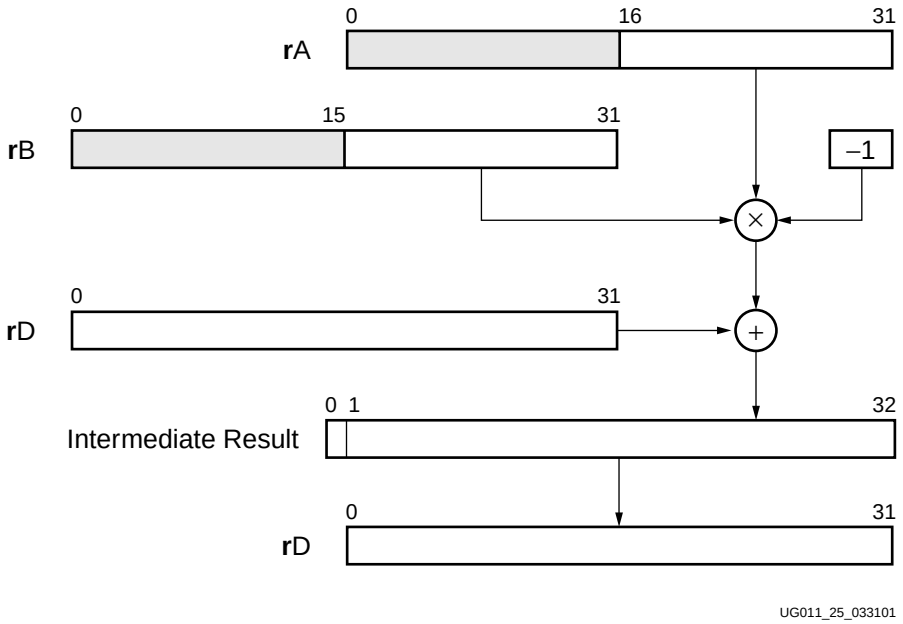


Figure 3-33: **Negative Multiply-Accumulate Low-Halfword to Word Operation**

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.

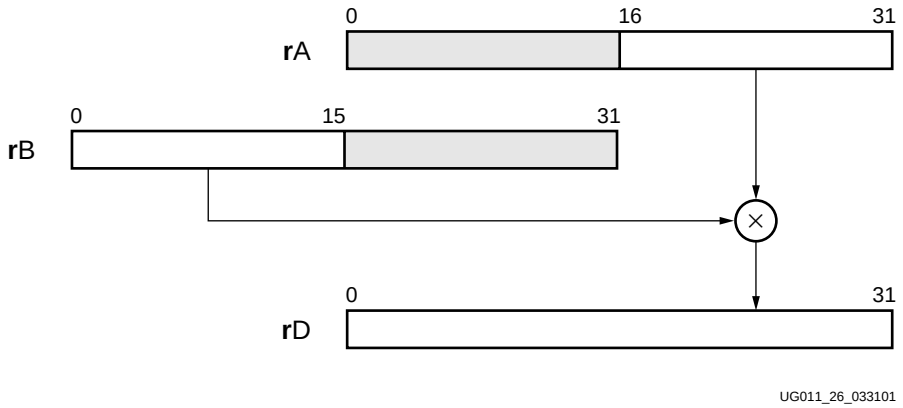


Figure 3-34: **Multiply Cross-Halfword to Word Operation**

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.

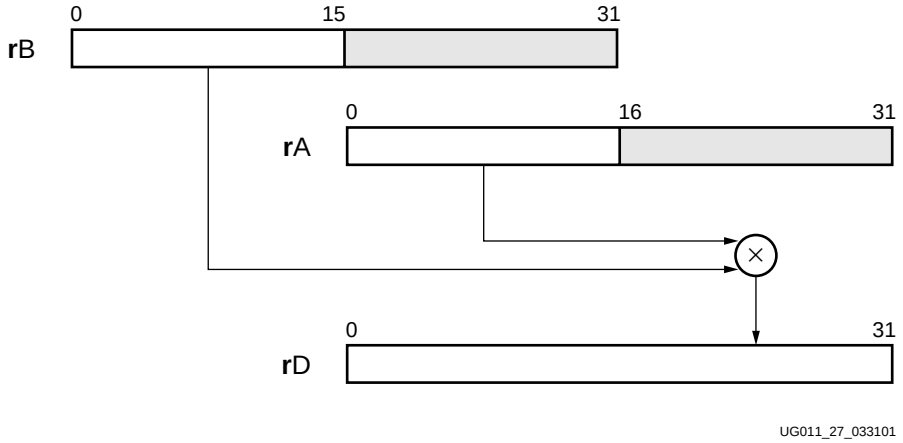


Figure 3-35: **Multiply High-Halfword to Word Operation**

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.

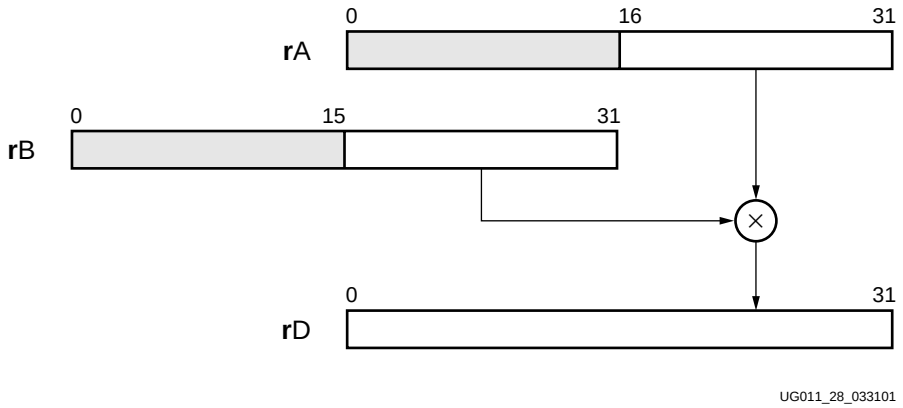


Figure 3-36: **Multiply Low-Halfword to Word Operation**

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 215, 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 78, for information on instructions used to manipulate bits and fields in the CR. See **Conditional Branch Control**, page 69, 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.

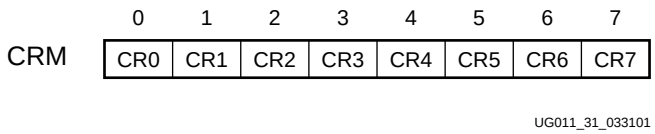


Figure 3-37: **mtcrf Field Mask (CRM) Format**

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.

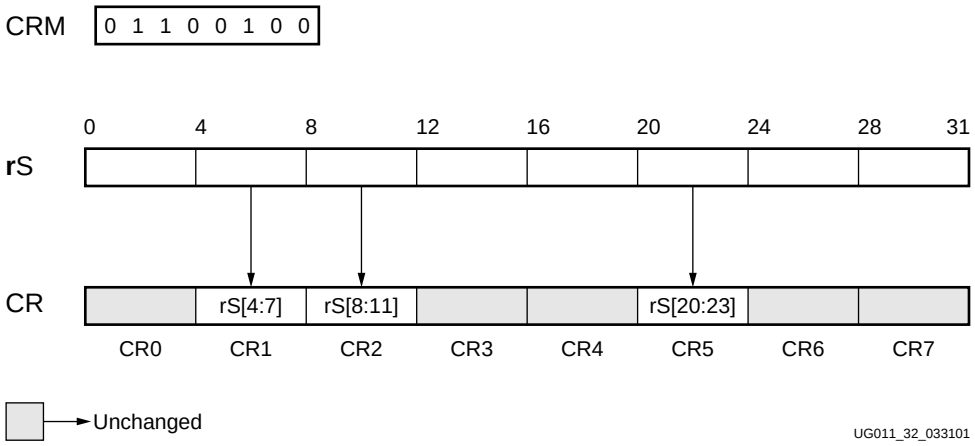


Figure 3-38: **mtcrf Example**

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 271.

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 44.

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 537.

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 159 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 228.

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 159.
- The processor ID register (PID) is described in **Virtual Mode**, page 174.
- The zone-protection register (ZPR) is described in **Virtual-Mode Access Protection**, page 185.
- The storage-attribute control registers are described in **Memory-System Control**, page 153.
- The exception-handling registers are described in **Interrupt-Handling Registers**,

page 201.

- The debug registers are described in **Debug Registers**, page 241.
- The timer registers, including the time base, are described in **Timer Resources**, page 227.

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 470 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 137, 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 530, 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 137, 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.

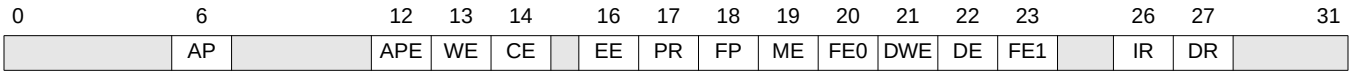


Figure 4-2: **Machine-State Register (MSR)**

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 138, 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 206, 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 206, 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 45, 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 208, 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 241, for more information.
22	DE	Debug Interrupt Enable 0—Disabled. 1—Enabled.	Controls the debug interrupt. See **Debug Interrupt (0x2000)**, page 225, 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 173, 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 173, for an introduction.
28:31		Reserved	

The initial state of the MSR following a processor reset is described in **Machine-State Register**, page 264.

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.

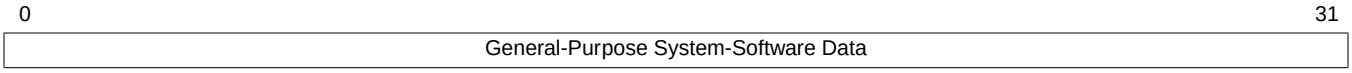


Figure 4-3: **SPR General-Purpose Registers (SPRG0–SPRG7)**

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 67).

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.

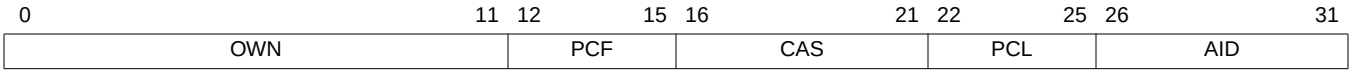


Figure 4-4: **Processor-Version Register (PVR)**

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 ***PowerPC® 405 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 198, for more information.	—
rfci	Return from Critical Interrupt	Return from critical-interrupt handler. See **Returning from Interrupt Handlers**, page 198, for more information.	—
sc	System Call	Causes a system-call exception to occur. See **System-Call Interrupt (0x0C00)**, page 218, 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 44, 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 271.

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 530, 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 271.

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 ***PowerPC® 405 Processor Block Manual*** for more information on the OCM and OCM controllers.

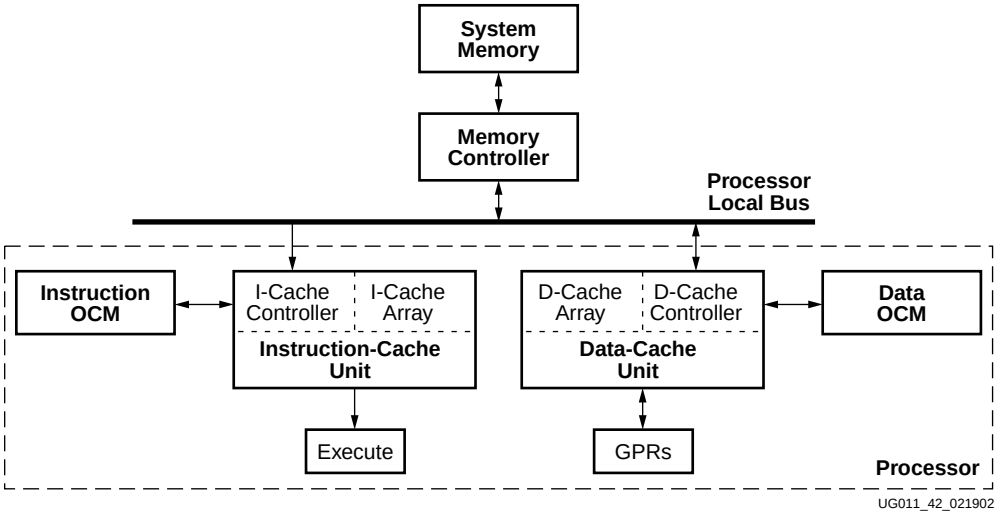


Figure 5-1: **PPC405 Memory-System Organization**

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.

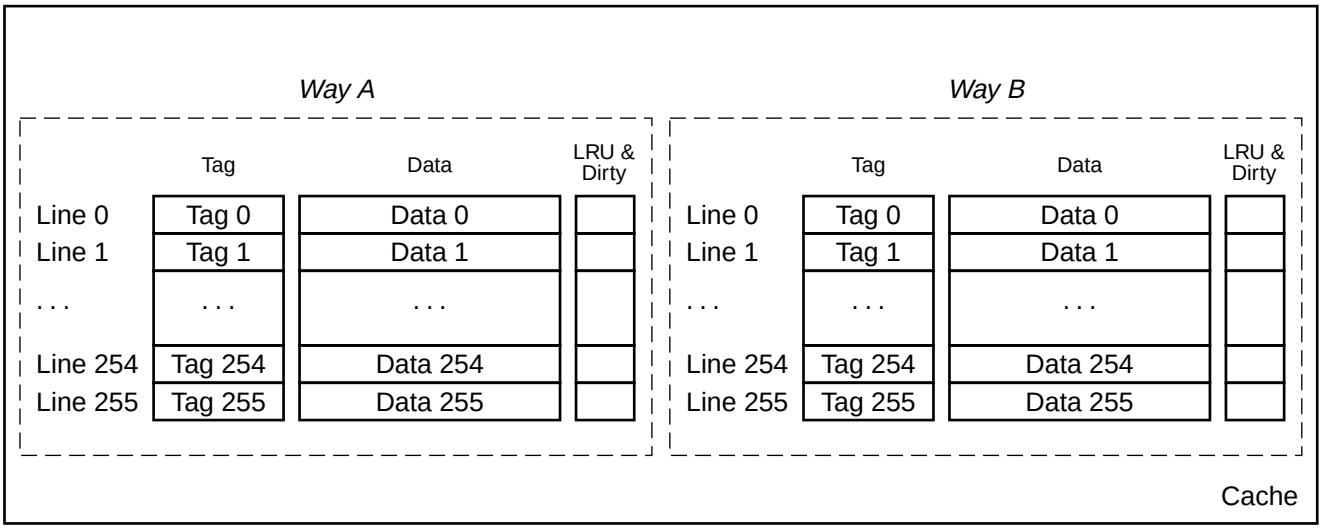


Figure 5-2: **Logical Structure of the PPC405 Cache Arrays**

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 145). 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.

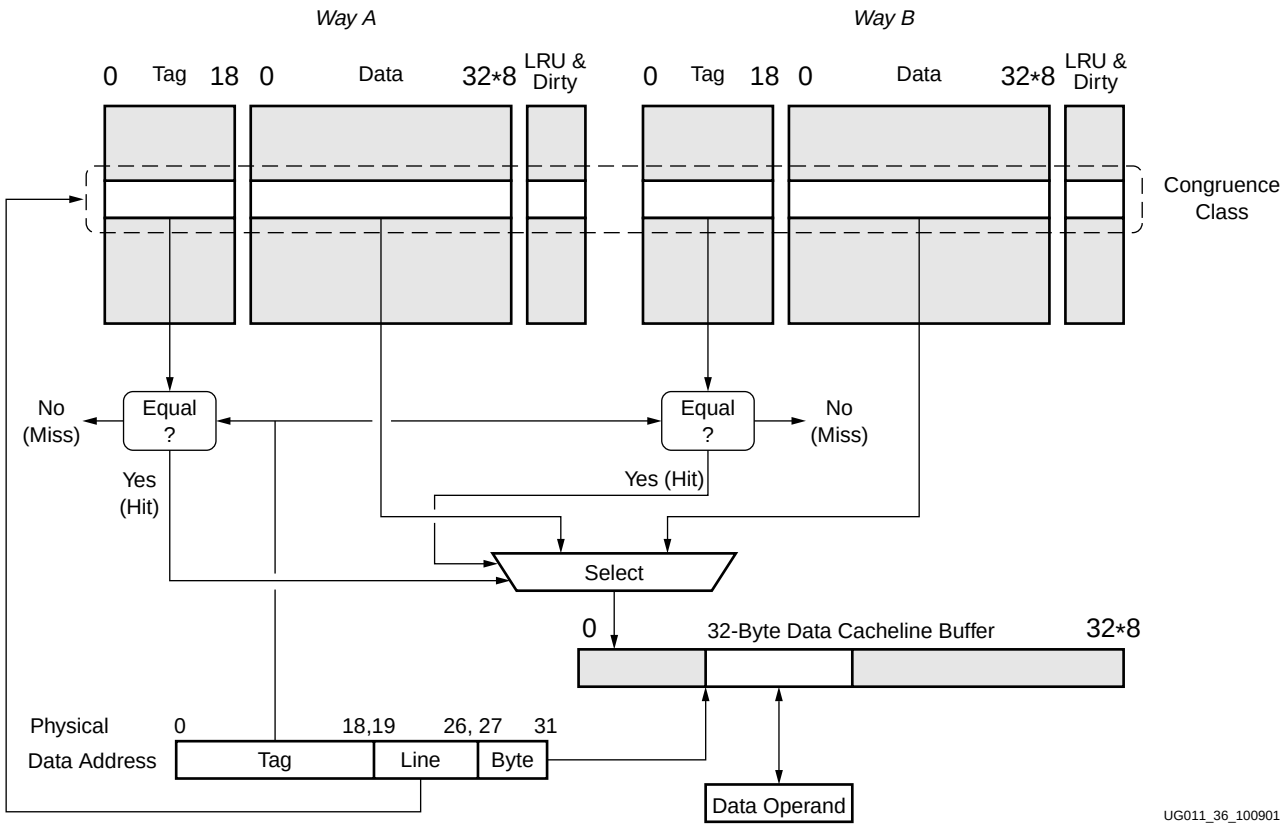


Figure 5-4: **Data-Cache Access Example**

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.

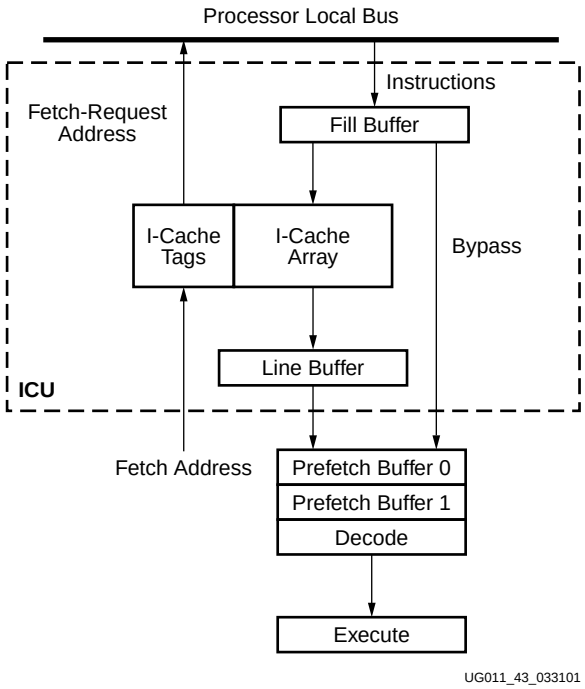


Figure 5-5: **Instruction Flow from the Instruction-Cache Unit**

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 162). 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 157, 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 181, 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 159, for information on this instruction). After the cache is invalidated, the ICCR can be configured.

Core-Configuration Register, page 162, 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.

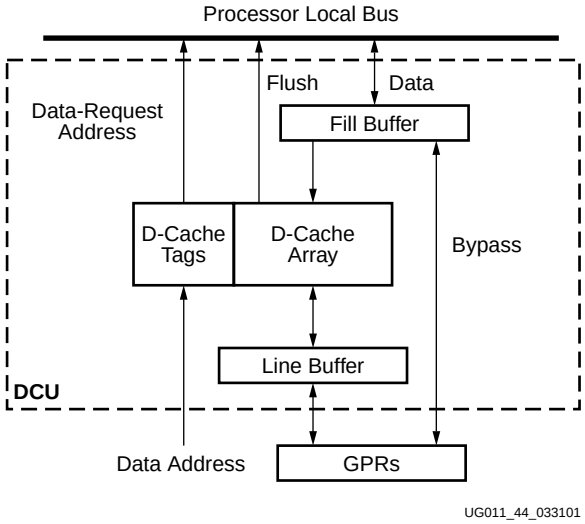


Figure 5-6: **Data Flow to/from the Data-Cache Unit**

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 162). 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 156, 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 159, 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 181, 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 166, 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 156, 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 181, 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 162, 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 163, 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 166, 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 126, 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 154). 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 181). In real mode, the storage-attribute control registers are used to define the storage attributes (see **Storage-Attribute Control Registers**, page 155).

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 155, 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 166, 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 151 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 162). 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 210 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 51 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 181 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

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
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.

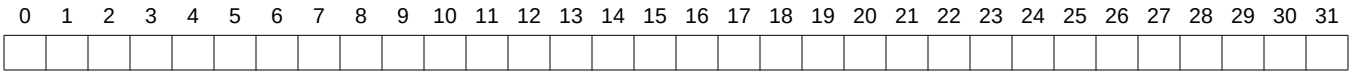


Figure 5-7: **Data-Cache Write-Through Register (DCWR)**

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.

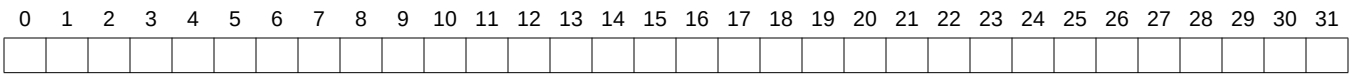


Figure 5-8: **Data-Cache Cacheability Register (DCCR)**

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 159 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 154 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.

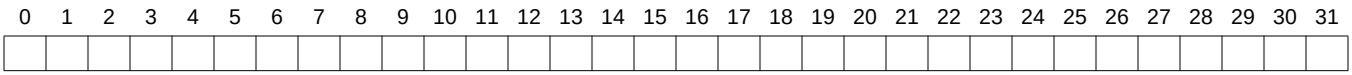


Figure 5-9: **Instruction-Cache Cacheability Register (ICCR)**

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 159 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 154 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 151 for more information.

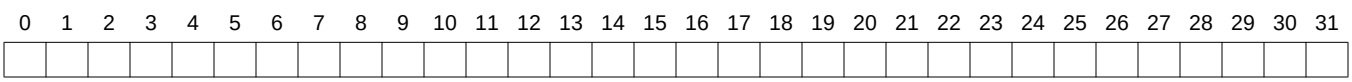


Figure 5-10: **Storage Guarded Register (SGR)**

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.

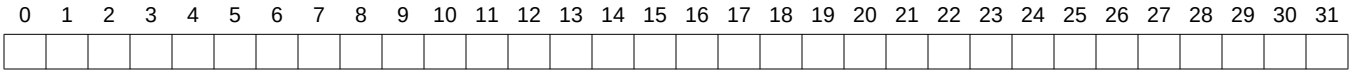


Figure 5-11: **Storage User-Defined 0 Register (SU0R)**

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 162). 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 210 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 51 for more information on big-endian and little-endian memory accesses.

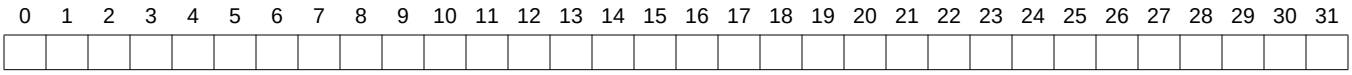


Figure 5-12: **Storage Little-Endian Register (SLER)**

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

Table 5-4: **Instruction-Cache Control Instructions**

Mnemonic	Name	Operation	Operand Syntax
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 171 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 172 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.

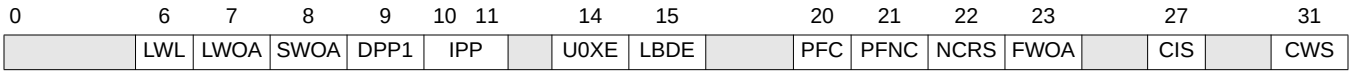


Figure 5-13: **Core-Configuration Register (CCR0)**

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 164 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 164 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 210 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 171 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 171 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*.

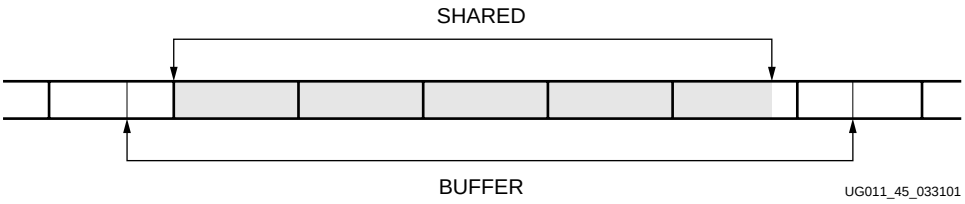


Figure 5-14: **Example of Shared-Memory Allocation**

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 162). 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.

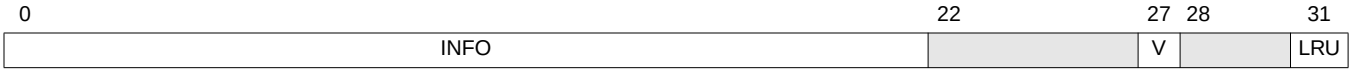


Figure 5-15: **Instruction-Cache Debug-Data Register (ICDBDR)**

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.

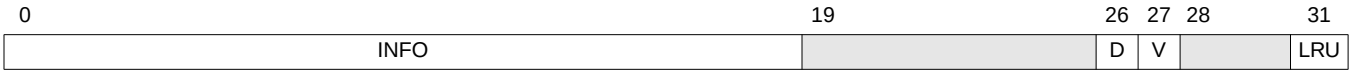


Figure 5-16: **Information Fields Loaded by dcread into rD**

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 46). 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 155, 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).

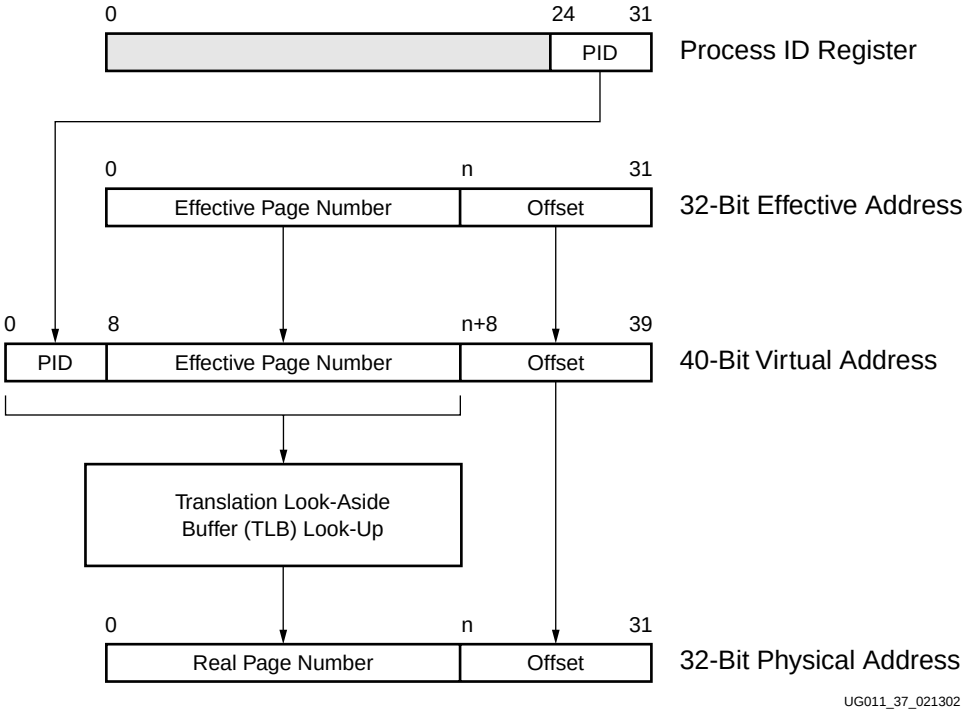


Figure 6-1: **Virtual-Mode Address Translation**

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 176). 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 181.

System software maintains a page-translation table that contains entries used to translate each virtual page into a physical page (see page 176). 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 181, 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 178, 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).

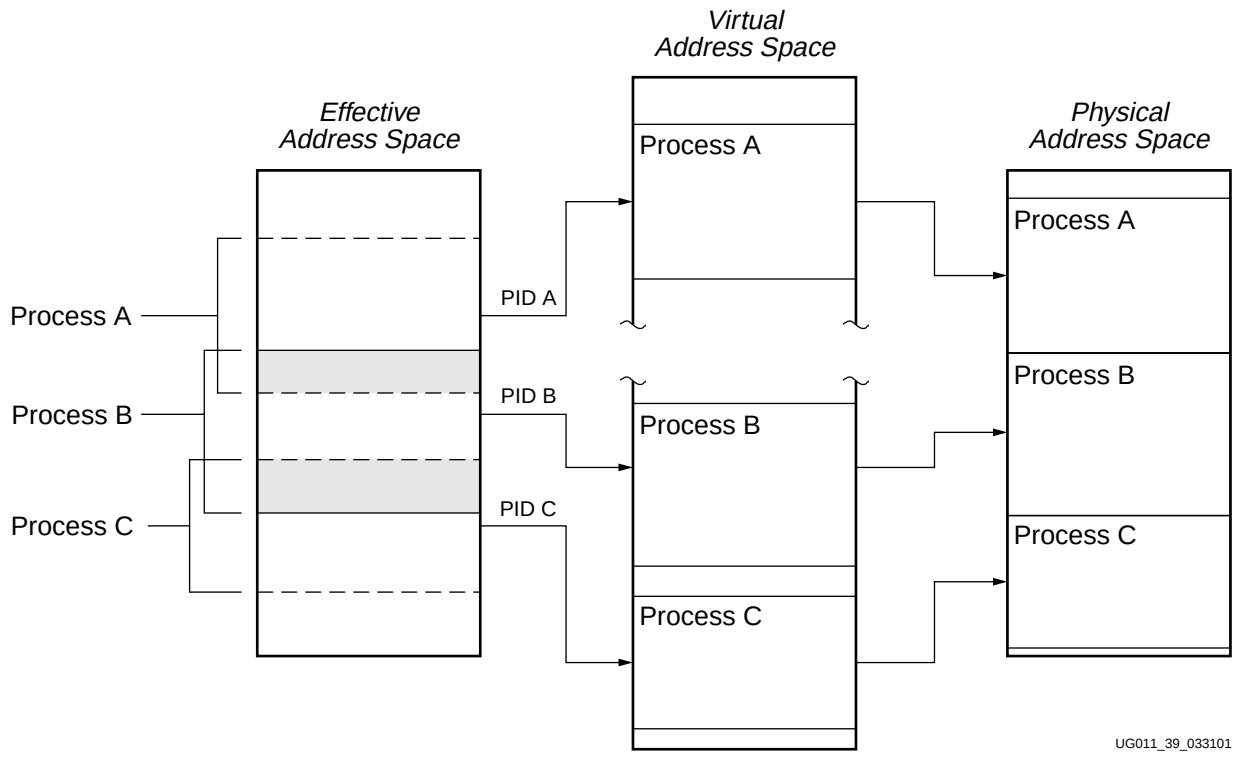


Figure 6-2: **Process-Mapping Example**

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.

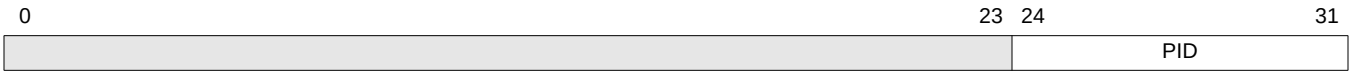


Figure 6-3: **Process-ID Register (PID)**

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.

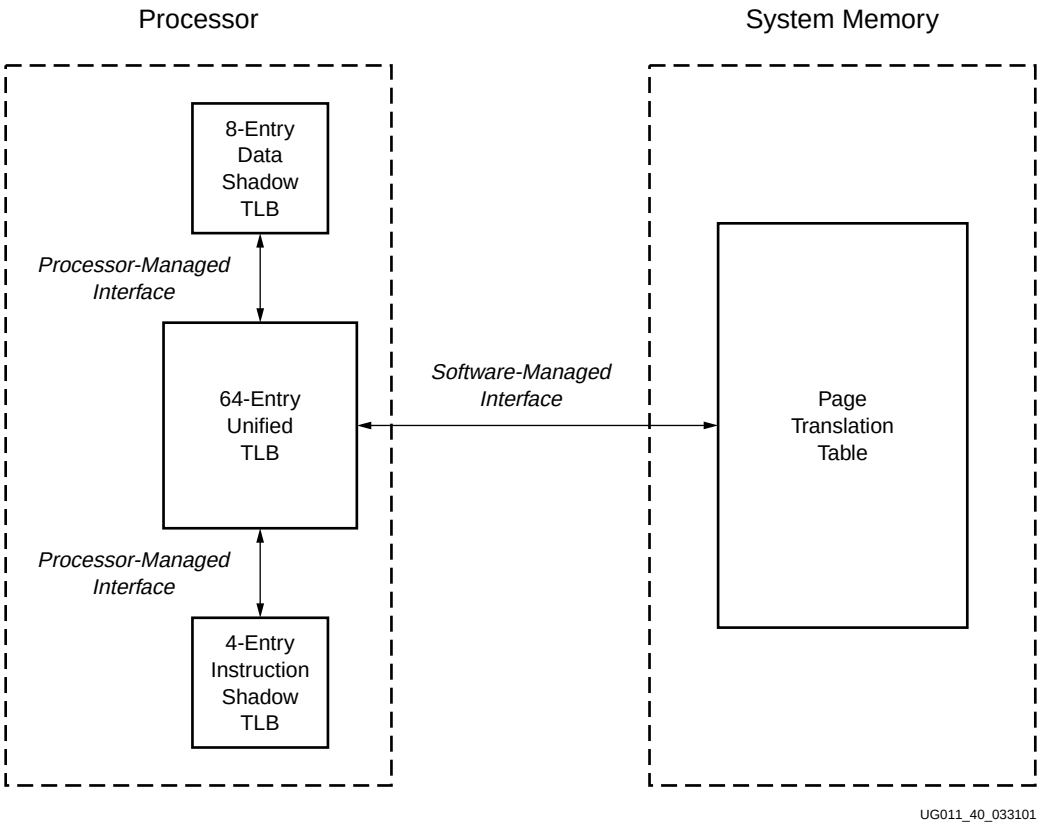


Figure 6-4: **Page-Translation Table and TLB Organization**

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.

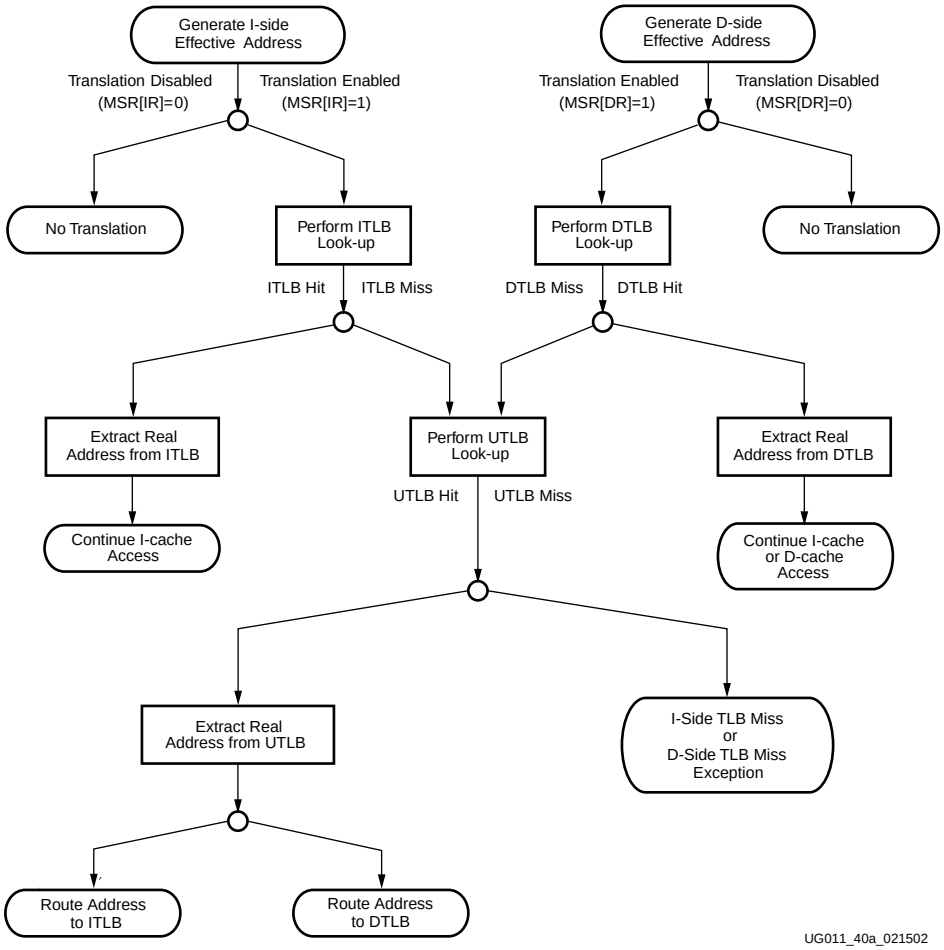


Figure 6-5: **ITLB/DTLB/UTLB Address Translation Flow**

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 190, 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.

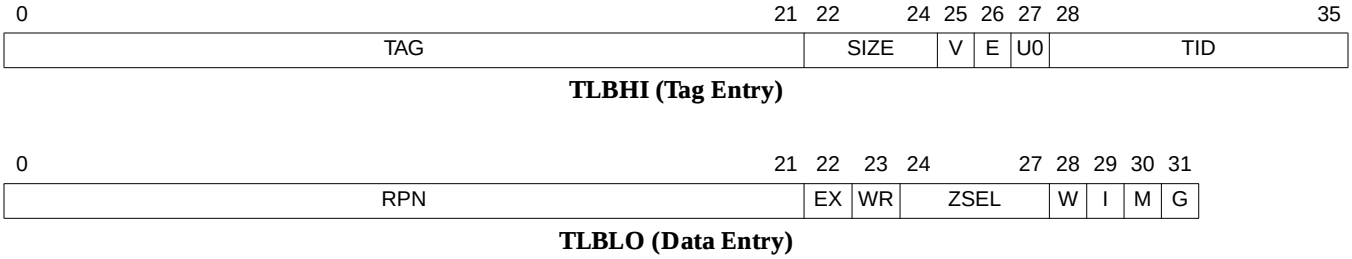


Figure 6-6: **TLB-Entry Format**

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 181, 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 181. 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 181, 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 185, 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 51, 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 155, 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 154, 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 154, 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 154, for more information.

In real mode, the storage-attribute control registers are used to define storage attributes. See **Storage-Attribute Control Registers**, page 155 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 175.

When assigning sizes to instruction pages, software must be careful to avoid creating the opportunity for instruction-cache synonyms. See **Instruction-Cache Synonyms**, page 145, 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
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 183.

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 181). 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 185, 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 181, for more information.

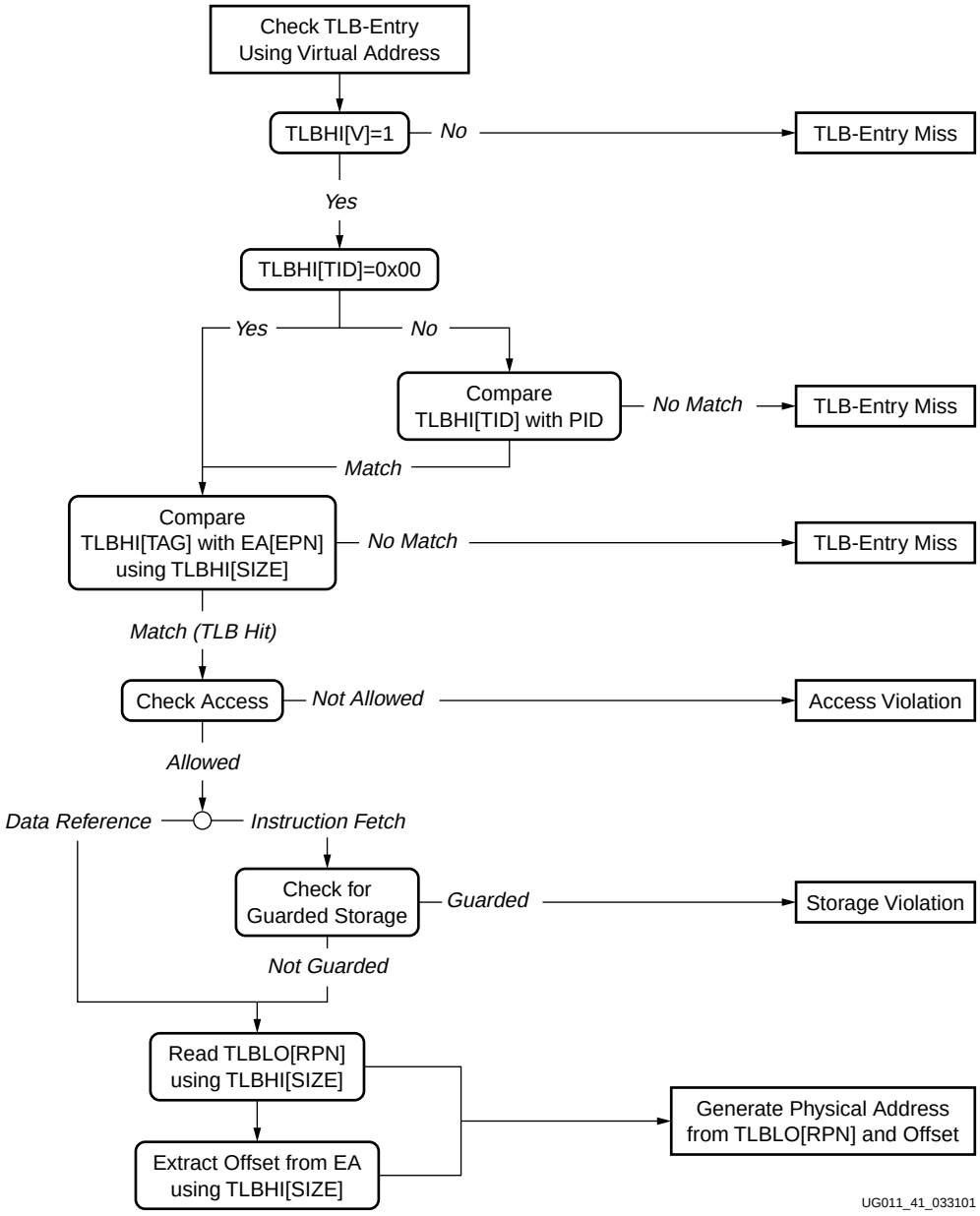


Figure 6-7: **General Process for Examining a TLB Entry**

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 210, for more information on this exception and **Zone Protection**, page 185, 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 212, for more information on this exception, **Guarded (G)**, page 154, for more information on guarded storage, and **Zone Protection**, page 185, 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 223, 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 224, 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.

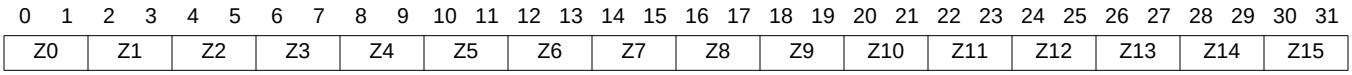


Figure 6-8: **Zone-Protection Register (ZPR)**

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 532, 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 55).
- The accessed memory location must be protected by the guarded storage attribute (see **Guarded (G)**, page 154).

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 206, 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.

Table 7-1: **Exceptions Supported by the PPC405D5 *(Continued)***

Exception	Vector Offset	Classification			Cause
Instruction Storage	0x0400	Noncritical	Synchronous	Precise	Instruction-access violation.
External	0x0500	Noncritical	Asynchronous	Precise	External noncritical-interrupt signal.
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.

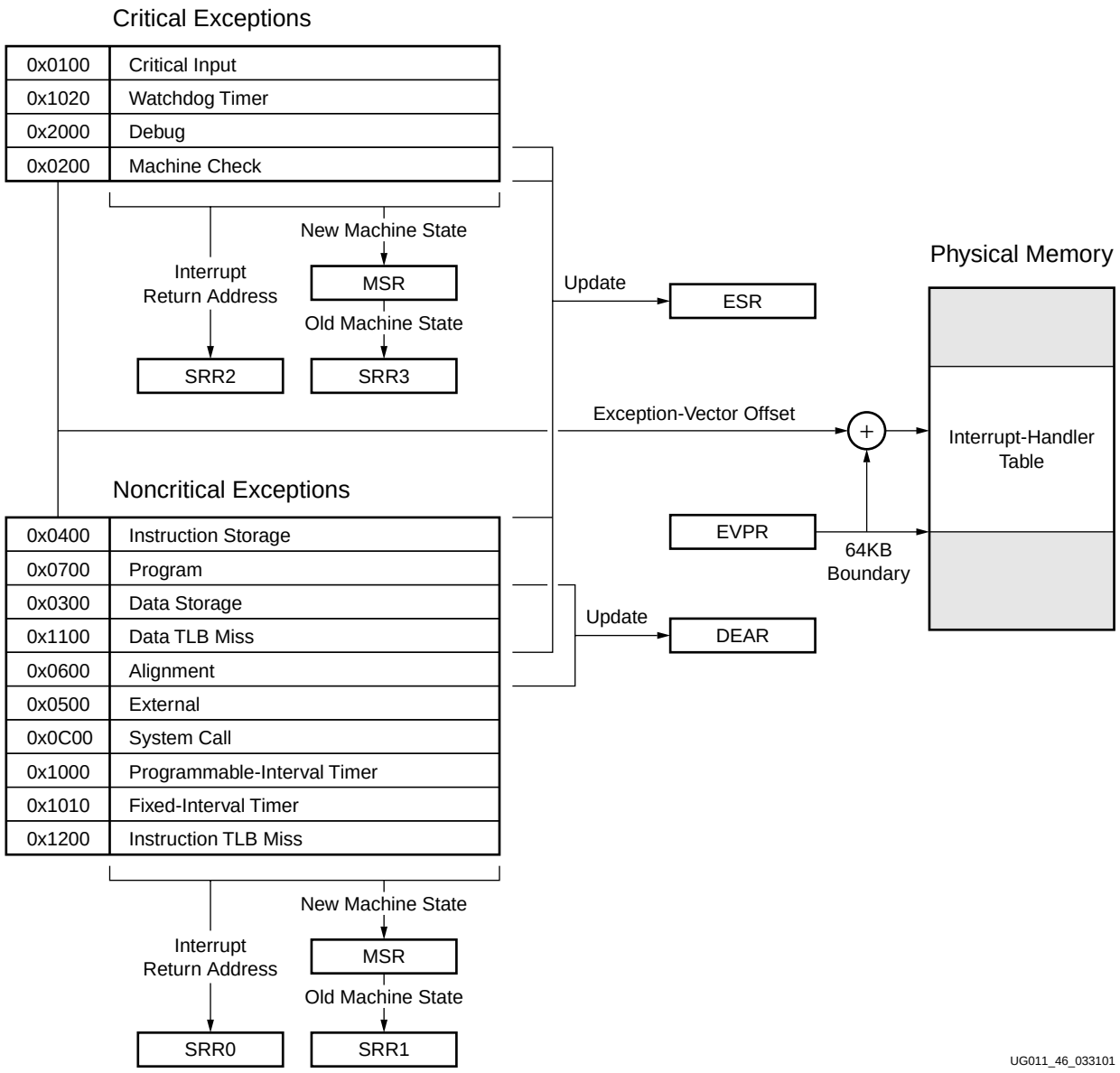


Figure 7-1: **PPC405 Exception Mechanism**

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 206 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

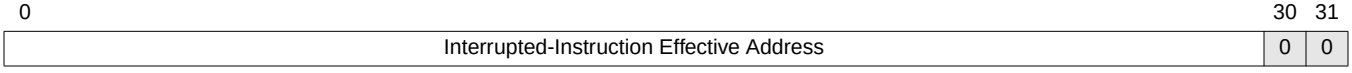
During an interrupt, the contents of the MSR (see page 133) are loaded into either SRR1 (noncritical interrupts) or SRR3 (critical interrupts). Depending on the interrupt, the MSR is updated with the values shown in Table 7-3.

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

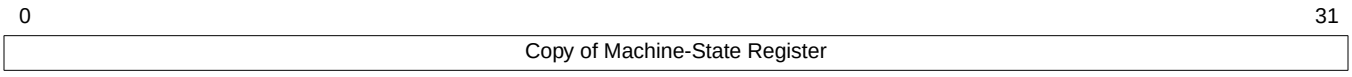
Bit	Name	Interrupt	Value	Description
0:5		All	0	Reserved
6	AP	All	0	This unsupported bit is cleared, but otherwise ignored.
7:11		All	0	Reserved
12	APE	All	0	This unsupported bit is cleared, but otherwise ignored.
13	WE	All	0	Processor wait state is disabled.
14	CE	Critical-Input Interrupt Machine-Check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

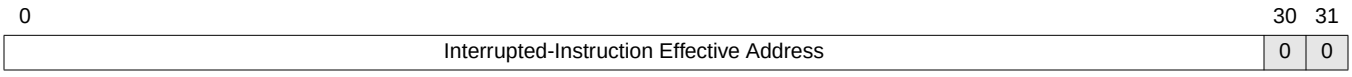
SRR0 is a privileged SPR with an address of 26 (0x01A) and SRR1 is a privileged SPR with an address of 27 (0x01B). Both registers are read and written using the **mfspr** and **mtspr** instructions.

Save/Restore Registers 2 and 3

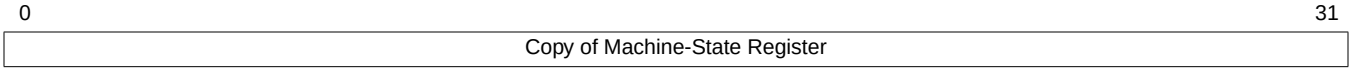
The save / restore registers 2 and 3 (SRR2 and SRR3) are 32-bit registers used to save machine state when a critical interrupt occurs. Interrupts defined as critical are:

- 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.

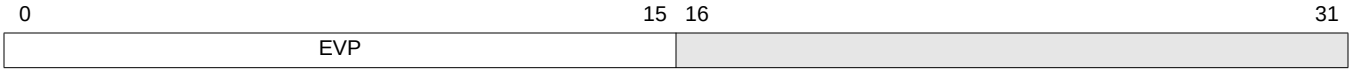


Figure 7-4: **Exception-Vector Prefix Register (EVPR)**

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.

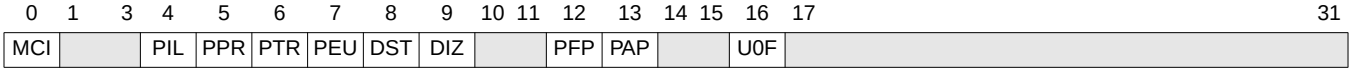


Figure 7-5: **Exception-Syndrome Register (ESR)**

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.

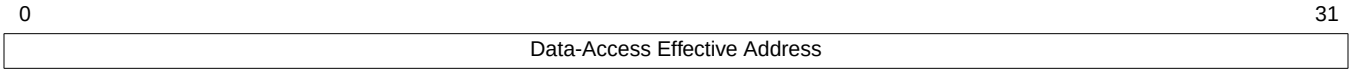


Figure 7-6: **Data Exception-Address Register (DEAR)**

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 182, 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 182, 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 245, 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 ***PowerPC® 405 Processor Block Manual*** for more information on setting the timer frequency.

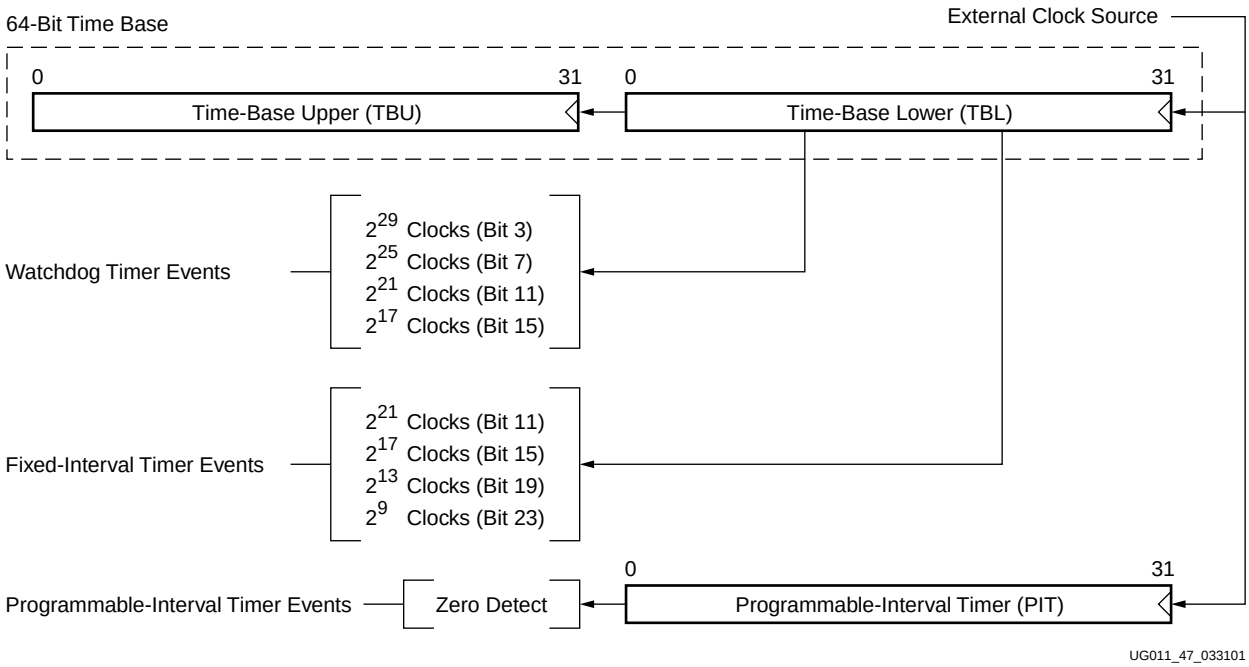


Figure 8-1: **PPC405 Timer Resources**

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.

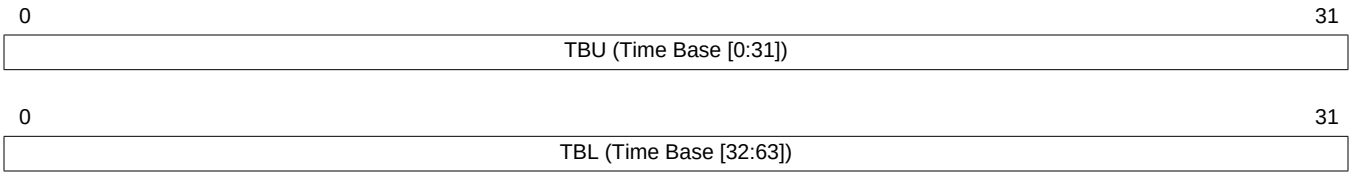


Figure 8-2: **Time-Base Register**

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 271).

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 530, 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.

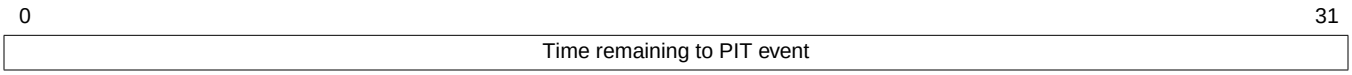


Figure 8-3: **Programmable-Interval Timer Register (PIT)**

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 236. 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.

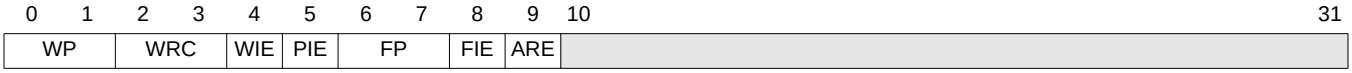


Figure 8-4: **Timer-Control Register (TCR)**

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.

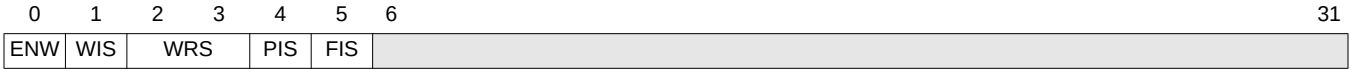


Figure 8-5: **Timer-Status Register (TSR)**

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 234, 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 206, 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.

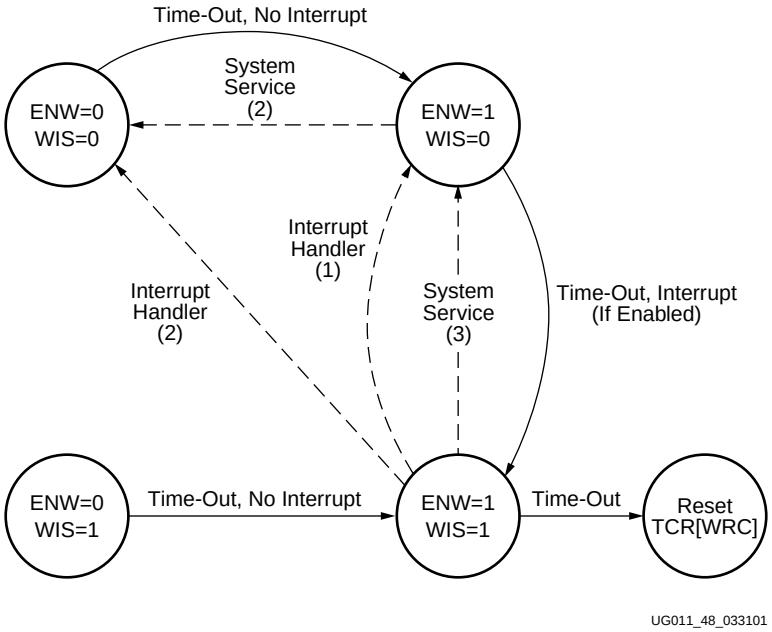


Figure 8-6: **Watchdog-Event State Machine**

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.

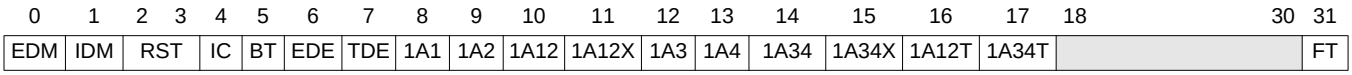


Figure 9-1: **Debug-Control Register 0 (DBCR0)**

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.
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.

Table 9-1: **Debug-Control Register 0 (DBCR0) Field Definitions *(Continued)***

Bit	Name	Function	Description
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.

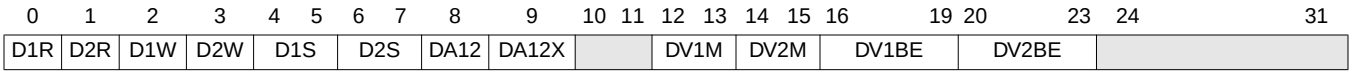


Figure 9-2: **Debug-Control Register 1 (DBCR1)**

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.

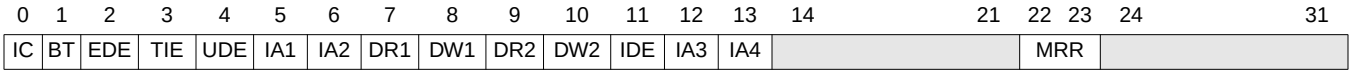


Figure 9-3: **Debug-Status Register (DBSR)**

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).

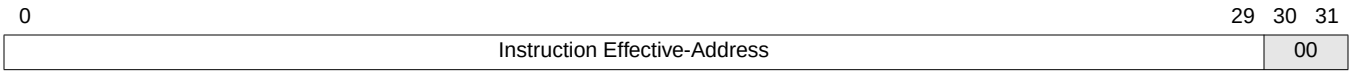


Figure 9-4: **Instruction Address-Compare Registers (IAC1–IAC4)**

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.

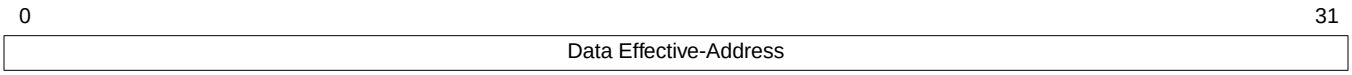


Figure 9-5: **Data Address-Compare Registers (DAC1, DAC2)**

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.

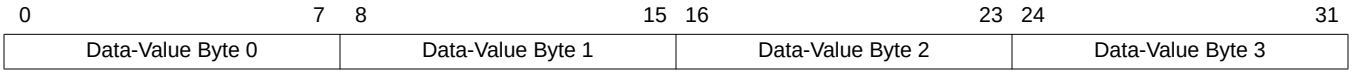


Figure 9-6: **Data Value-Compare Registers (DVC1, DVC2)**

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 259 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			
TDE Trap Instruction	TDE		TDE			
UDE Unconditional			UDE			

Table 9-4: **Debug Resources Used by Debug Events *(Continued)***

Debug Event	DBCR0	DBCR1	DBSR	IAC	DAC	DVC
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.

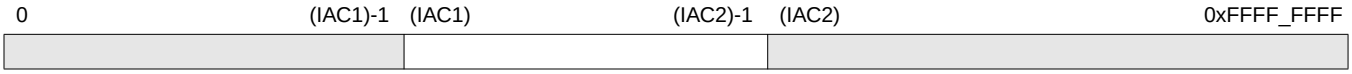
  If DBCR0[IA12X]=0, instruction addresses from (IAC1) to (IAC2)-1 fall within the range. Addresses from 0 to (IAC1)-1 and (IAC2) to 0xFFFF_FFFF fall outside the range.

  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.

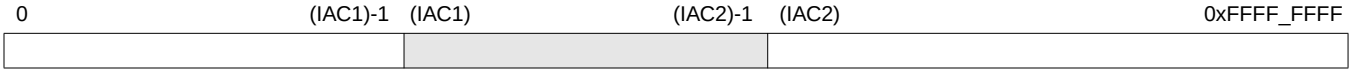
  If DBCR0[IA12X]=1, instruction addresses from 0 to (IAC1)-1 and (IAC2) to 0xFFFF_FFFF fall within the range. Addresses from (IAC1) to (IAC2)-1 fall outside the range.

  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

Table 9-8: **Examples of Using the D1S Size Field**

DAC Address	D1S Value (Granularity)	Operand Address	Access Size	DAC Match
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.

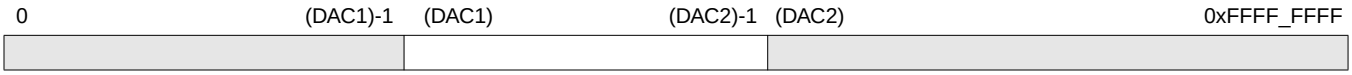
Read and write accesses can be checked independently. To check read accesses, software sets the D1R and / or D2R bits in the DBCR1 register. Only one of the two bits must be set to enable read checking for the entire range. If a read-access match is detected, the corresponding status bits in the DBSR 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

Figure 9-8: **DAC Address-Range Specification**

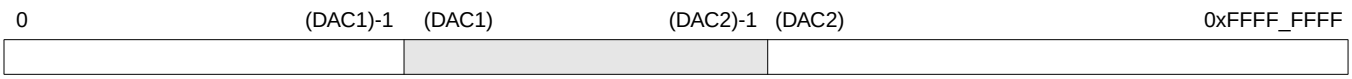


Figure 9-8: **DAC Address-Range Specification**

Table 9-9 summarizes the DBCR1 bits used to control DAC address-range comparisons and the DBSR bits used to report their status.

Table 9-9: **DAC Address-Range Match Resources**

Event Enable Bit (DBCR1)	DBCR1 [DA12X]	Type of Access Checked	Event Status Bit (DBSR)
D1R and / or D2R	0	Load (read) inclusive (DAC1) and (DAC2)-1	DR1 and / or DR2
D1W and / or D2W		Store (write) inclusive (DAC1) and (DAC2)-1	DW1 and / or DW2
D1R and / or D2R	1	Load (read) exclusive (DAC1) and (DAC2)-1	DR1 and / or DR2
D1W and / or D2W		Store (write) exclusive (DAC1) and (DAC2)-1	DW1 and / or DW2

The processor does not clear the DBSR status bits when DAC events fail to occur. After a DAC event is recorded by a debugger, the corresponding status bits should be cleared to prevent ambiguity when recording future debug events.

DAC Events Caused by Cache Instructions

DAC events can be caused by the execution of cache-control instructions. The following summarizes the type of DAC events that can occur when a cache-control instruction is executed:

- Cache-control instructions that can modify data are treated as stores (writes) by the debug mechanism. Instructions that can cause loss of data through invalidation are also treated as stores. Both types of instructions can cause DAC-write events. Instructions in this category are **dcbi** and **dcbz**.
- Cache-control instructions that invalidate unmodified are treated as loads. These instructions can cause DAC-read events but not DAC-write events. The **icbi** instruction falls in this category.
- Cache-control instructions that are not address specific do not cause DAC events. Instructions in this category are **dccci**, **iccci**, **dcread**, and **icread**.
- Cache-control instructions that update system memory with data already present in the cache are treated as loads (reads) by the access-protection mechanism. However, the debug mechanism can be used to cause a DAC-write event when these instructions are executed. Instructions in this category are **dcbf** and **dcbst**.
- Cache-control instructions that are speculative are treated as loads by the debug mechanism. These instructions can cause DAC-read events. Instructions in this category are **dcbt**, **dcbtst**, and **icbt**.
- Cache-control instructions that allocate cachelines are treated as stores. These instructions can cause DAC-write events. The **dcba** instruction falls in this category.

Table 9-10 summarizes the type of DAC event that can occur for each cache-control instruction.

Table 9-10: **DAC Events Caused by Cache-Control Instructions**

Instruction	DAC Read	DAC Write
dcba	No	Yes
dcbf	No	Yes
dcbi	No	Yes
dcbst	No	Yes
dcbt	Yes	No
dcbtst	Yes	No
dcbz	No	Yes
dccci	Does not cause DAC events.	
dcread	Does not cause DAC events.	
icbi	Yes	No
icbt	Yes	No
iccci	Does not cause DAC events.	
icread	Does not cause DAC events.	

Data Value-Compare Debug Event

A data value-compare (DVC) debug event occurs when:

1. A DAC match occurs. The operand effective-address of the data-access instruction must match the value contained in one of the DAC*n* registers, using the conditions specified by the DBCR1 register.
2. If the preceding DAC comparison detects a matching address, the data-value accessed at that address must match the value contained in one of the DVC*n* registers, using the conditions specified by the DBCR1 register.

The DAC comparison performed in the first step is set up to perform exact-address or address-range comparisons as described in the previous section (**Data Address-Compare Debug Event**). However, the DAC comparison does *not* cause a DAC debug event. Because DVC and DAC events share the same DAC registers, control bits, and status bits, a DAC event is disabled when the corresponding DVC event is enabled, as follows:

- If DVC1 events are enabled, DAC1 events are disabled.
- If DVC2 events are enabled, DAC2 events are disabled.
- If DVC1 and DVC2 events are enabled (as in range comparisons), DAC1 and DAC2 events are disabled.

Unlike DAC events, the DVC event occurs *after* the data-access instruction executes. If debug interrupts are enabled, the SRR2 register is loaded with the effective address of the instruction following the one that caused the DVC event.

DVC events are enabled by loading a non-zero value ($\neq$ 0b0000) into the byte-enable controls of the corresponding DVC*n* register. A non-zero value loaded into DBCR1[DV1BE] enables DVC1 events and a non-zero value loaded into DBCR1[DV2BE] enables DVC2 events. Referring to Figure 9-6, page 247, 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 ***PowerPC® 405 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.

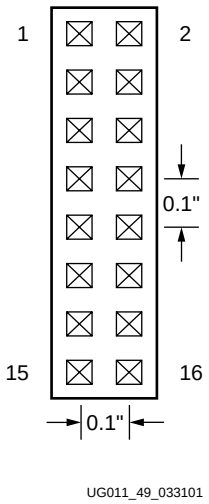


Figure 9-9: **JTAG-Connector Physical Layout**

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)	

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 9-13: **JTAG Connector Signals *(Continued)***

Pin	I/O	Signal Name	Description
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.

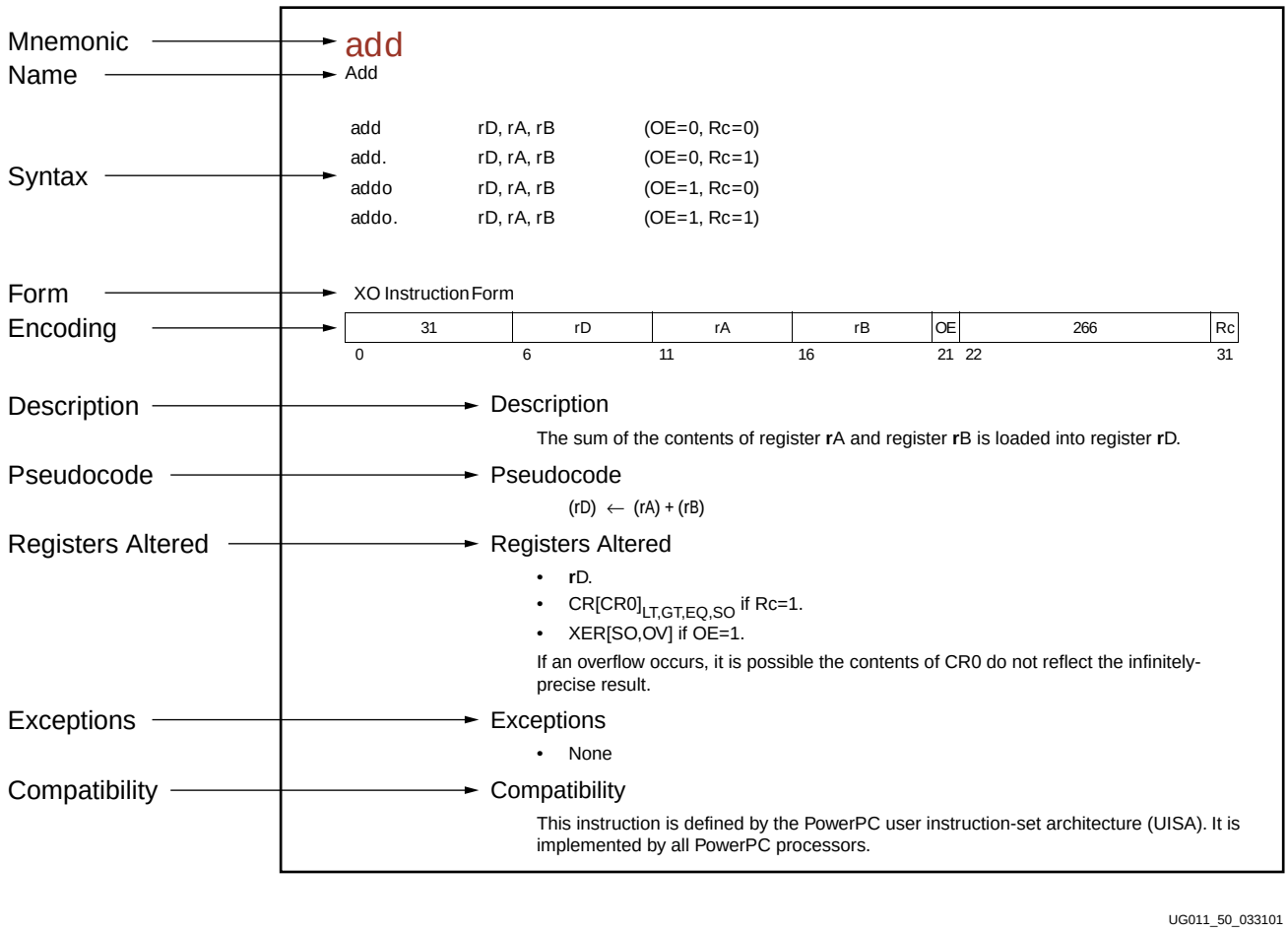


Figure 11-1: **Instruction Description Format**

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 492 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 17. The precedence of pseudocode operations is further described in the Preface in **Operator Precedence**, page 19.

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 92.

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 534.
- **Load Immediate**, page 534.
- **Subtract Instructions**, page 531.

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 531.

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 531.

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 534.
- **Subtract Instructions**, page 531.

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 92.

Pseudocode

```
(rD) ← (rA) + XER[CA] + (−1)
if    (rD) ≯ 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 92.

Pseudocode

```
(rD) ← (rA) + XER[CA]
if    (rD) ≯ 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) $\leftarrow$ (**r**S) $\wedge$ (**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

(**r**A) $\leftarrow$ (**r**S) $\wedge$ ($^{16}0$ || 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	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 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

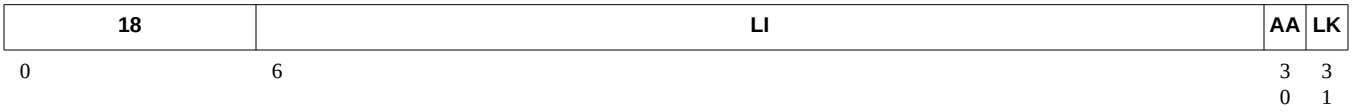
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

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

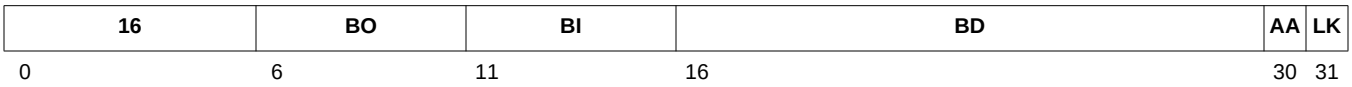
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

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 69. 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 69. 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 69. 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

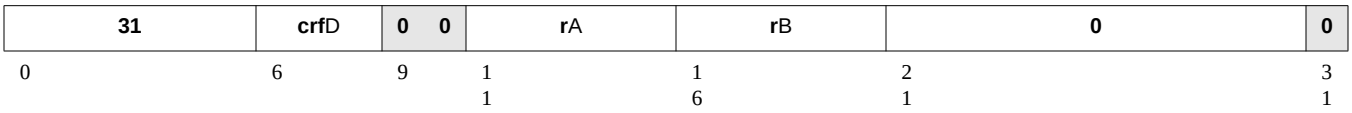
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

cmp

Compare

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

X Instruction Form



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 528.

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

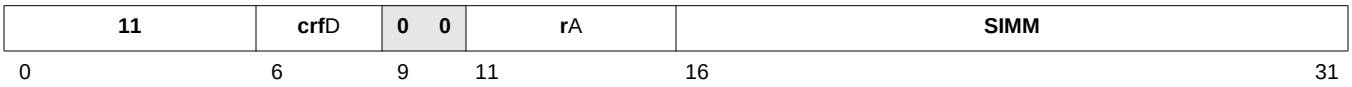
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

cmpi

Compare Immediate

cmpi **crf**D, 0, **r**A, SIMM

D Instruction Form



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 528.

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	20	21	30	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 528.

Pseudocode

$c_{0:3} \leftarrow$ 0b0000
if (**r**A) $\overset{u}{<}$ (**r**B) then $c_0 \leftarrow 1$
if (**r**A) $\overset{u}{>}$ (**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 528.

Pseudocode

$c_{0:3} \leftarrow$ 0b0000
if (**r**A) $\overset{u}{<}$ (^{16}0 || UIMM)then $c_0 \leftarrow 1$
if (**r**A) $\overset{u}{>}$ (^{16}0 || UIMM)then $c_1 \leftarrow 1$
if (**r**A) $=$ (^{16}0 || UIMM)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.

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 528.

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 528.

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	**crb**D	**crb**A	**crb**B	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 528.

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 528.

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	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 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 ← (**r**A|0) + (**r**B)
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

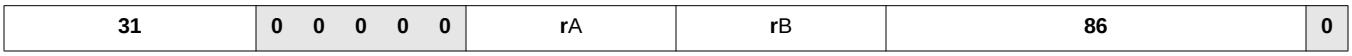
This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture. Implementation of this instruction is optional, and it is not guaranteed to be implemented on all PowerPC processors.

dcbf

Data Cache Block Flush

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

X Instruction Form



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 ← (**r**A|0) + (**r**B)
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

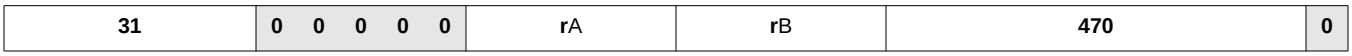
This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture.

dcbi

Data Cache Block Invalidate

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

X Instruction Form



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 ← (**r**A|0) + (**r**B)
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 ← (**r**A|0) + (**r**B)
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 ← (**r**A|0) + (**r**B)
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 ← (**r**A|0) + (**r**B)
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.

dcccci

Data Cache Congruence Class Invalidate

dcccci **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 **dcccci** instructions should be executed, one for each congruence class. Afterwards, cachability can be enabled.

Pseudocode

EA ← (**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:

$$\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}$$

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

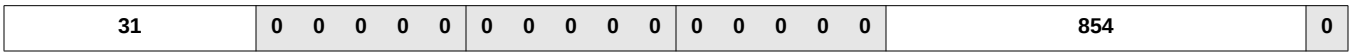
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

eieio

Enforce In Order Execution of I/O

eieio

X Instruction Form



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

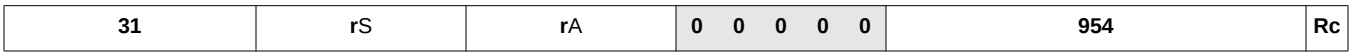
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

extsb

Extend Sign Byte

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

X Instruction Form



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

$(\mathbf{r}A) \leftarrow \text{EXTS}(\mathbf{r}S_{24:31})$

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

$(\mathbf{r}A) \leftarrow \text{EXTS}(\mathbf{r}S_{16:31})$

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

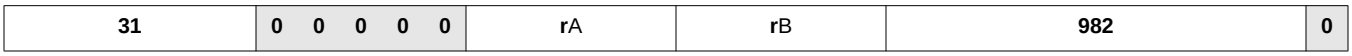
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

icbi

Instruction Cache Block Invalidate

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

X Instruction Form



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

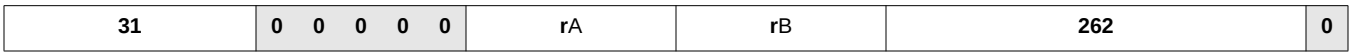
This instruction is defined by the virtual-environment architecture level (VEA) of the PowerPC architecture, the PowerPC embedded-environment architecture, and the PowerPC Book-E architecture.

icbt

Instruction Cache Block Touch

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

X Instruction Form



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

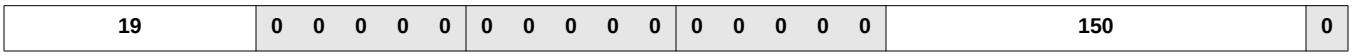
This instruction is implementation specific and is not guaranteed to be supported by other PowerPC processors.

isync

Instruction Synchronize

isync

XL Instruction Form



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 $\leftarrow$ (**r**A|0) + EXTS(d)
(**r**D) $\leftarrow$ $^{24}0$ || 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

0	6	11	16	31
43	rD	rA	d	

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 ← (**r**A) + EXTS(d)
(**r**D) ← EXTS(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:

- **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 $\leftarrow$ (**r**A|0) + (**r**B)
(**r**D) $\leftarrow$ $^{16}0$ || 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 ← (**r**A|0) + EXTS(d)
(**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.

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 ← (**r**A) + EXTS(d)
(**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:

- **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
RFINAL ← ((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 = RFINAL))
        then  (GPR(reg)) ← 0
  if ((reg ≠ rA) ∨ (reg = RFINAL))
    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 128 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 ← (**r**A) + EXTS(d)
(**r**D) ← MS(EA,4)
(**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.

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	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. 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 110.

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 110.

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 110.

Pseudocode

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

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.

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 110.

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 112.

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 112.

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 112.

Pseudocode

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

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.

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 112.

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 115.

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 115.

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 115.

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 115.

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	**crf**D	0 0	**crf**S	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

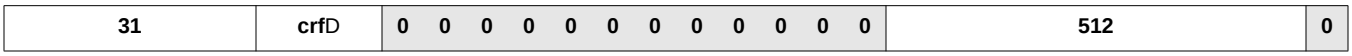
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

mcrxr

Move to Condition Register from XER

mcrxr **crf**D

X Instruction Form



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 271 for more information.

Pseudocode

DCRN $\leftarrow$ $DCRF_{5:9}$ || $DCRF_{0:4}$
(**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

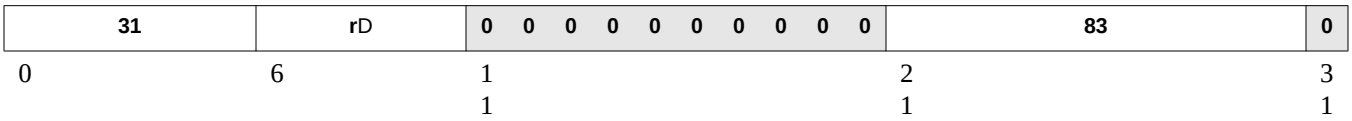
This instruction is defined by the PowerPC embedded-environment architecture and the PowerPC Book-E architecture. It is not defined by the PowerPC architecture, and is therefore not implemented by all PowerPC processors. The specific registers accessed by this instruction are implementation dependent.

mfmsr

Move from Machine State Register

mfmsr **r**D

X Instruction Form



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 271 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 530.

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 271 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 530.

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	11	12	20	21	31

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 125, 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 534.

Pseudocode

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

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 271 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 271 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 530.

Pseudocode

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

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 122.

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 122.

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 123.

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 123.

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	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 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 124.

Pseudocode

$(\mathbf{r}D)_{0:31} \leftarrow (\mathbf{r}A)_{16:31} \times (\mathbf{r}B)_{16:31}$ 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 124.

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	11	16 31

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	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 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	11	16	21	22	31

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 117.

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	11	16	21	22	31

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 117.

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	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. 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 119.

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 119.

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 121.

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 121.

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	11	16	21	31

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 534.

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 534.

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 534. The preferred no-operation (an instruction that does nothing) is:

```
ori 0,0,0
```

Pseudocode

(**r**A) $\leftarrow$ (**r**S) $\vee$ ($^{16}0$ || 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) $\leftarrow$ (**r**S) $\vee$ (UIMM || $^{16}0$)

Registers Altered

- **r**A.

Exceptions

- None.

Compatibility

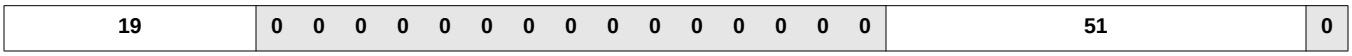
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

rfci

Return from Critical Interrupt

rfci

XL Instruction Form



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

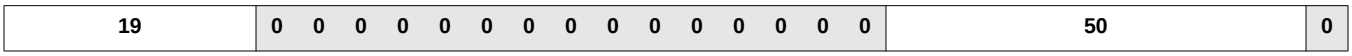
This instruction is defined by the operating-environment architecture level (OEA) of the the PowerPC embedded-environment architecture and the PowerPC Book-E architecture. It is not defined by the PowerPC architecture, and is therefore not implemented by all PowerPC processors.

rfi

Return from Interrupt

rfi

XL Instruction Form



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 102.

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 529.

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 102.

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 529.

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 102.

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 529.

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

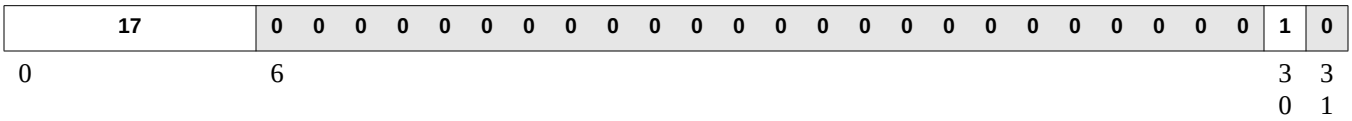
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

sc

System Call

sc

SC Instruction Form



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	1 1	1 6	2 1	3 1

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) ∨ (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.

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) ← (**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

$$EA \leftarrow (\mathbf{r}A|0) + (\mathbf{r}B)$$
$$MS(EA, 2) \leftarrow (\mathbf{r}S)_{24:31} \parallel (\mathbf{r}S)_{16:23}$$

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 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.
- 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          ← (rA) + (rB)
MS(EA, 2)   ← (rS)16:31
(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.

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 rS, rA, rB

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 128 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        ← (rA|0) + (rB)
MS(EA,4)  ← (rS)
```

Registers Altered

- None.

Exceptions

- Data storage—if the access is prevented by zone protection when data relocation is enabled.
  - No-access-allowed zone protection applies only to accesses in user mode.
  - Read-only zone protection applies to user and privileged modes.
- Data TLB miss—if data relocation is enabled and a valid translation-entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid-instruction forms results in a boundedly-undefined result rather than a program exception:

- Reserved bits containing a non-zero value.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (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	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.

Simplified mnemonics defined for this instruction are described in **Subtract Instructions**, page 531.

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 531.

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	11	16	21	22	31

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 94.

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	11	16 31

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 94.

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 94.

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

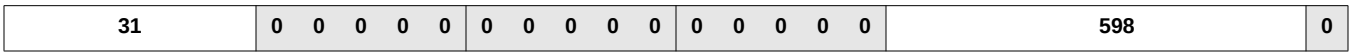
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

sync

Synchronize

sync

X Instruction Form



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

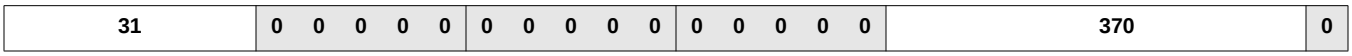
This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

tlbia

TLB Invalidate All

tlbia

X Instruction Form



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 179 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 532.

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

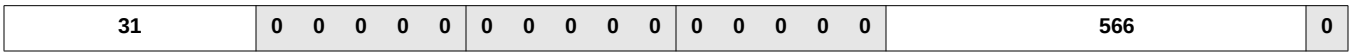
This instruction is defined as optional by the operating-environment architecture level (OEA) of the PowerPC embedded-environment architecture and the PowerPC Book-E architecture. Because it is optional and not defined by the PowerPC architecture it is not implemented by all PowerPC processors.

tlbsync

TLB Synchronize

tlbsync

X Instruction Form



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 179 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 532.

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	11	16	21	31

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 79 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 215.
- **Debug Interrupt (0x2000)**, page 225.
- **Trap-Instruction Debug Event**, page 250.
- **Internal-Debug Mode**, page 240.
- **External-Debug Mode**, page 240.

Simplified mnemonics defined for this instruction are described in **Trap Instructions**, page 532.

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 79 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 215.
- **Debug Interrupt (0x2000)**, page 225.
- **Trap-Instruction Debug Event**, page 250.
- **Internal-Debug Mode**, page 240.
- **External-Debug Mode**, page 240.

Simplified mnemonics defined for this instruction are described in **Trap Instructions**, page 532.

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] ← (**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) $\leftarrow$ (**r**S) $\oplus$ ($^{16}0$ || 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.

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 62
CR	Condition Register	**Condition Register (CR)**, page 63
MSR	Machine-State Register	**Machine-State Register**, page 133
CCR0	Core-Configuration Register 0	**Core-Configuration Register**, page 162
CTR	Count Register	**Count Register (CTR)**, page 66
DAC1	Data Address-Compare 1	**Data Address-Compare Registers**, page 247
DAC2	Data Address-Compare 2	
DBCR0	Debug-Control Register 0	**Debug-Control Registers**, page 241
DBCR1	Debug-Control Register 1	
DBSR	Debug-Status Register	**Debug-Status Register**, page 245
DCCR	Data-Cache Cacheability Register	**Data-Cache Cacheability Register (DCCR)**, page 156
DCWR	Data-Cache Write-Through Register	**Data-Cache Write-Through Register (DCWR)**, page 156
DEAR	Data-Error Address Register	**Data Exception-Address Register**, page 206
DVC1	Data Value-Compare 1	**Data Value-Compare Registers**, page 247
DVC2	Data Value-Compare 2	
ESR	Exception-Syndrome Register	**Exception-Syndrome Register**, page 204
EVPR	Exception-Vector Prefix Register	**Exception-Vector Prefix Register**, page 204

Table A-1: **PPC405 Register Cross-Reference *(Continued)***

Name	Descriptive Name	Cross Reference
IAC1	Instruction Address-Compare 1	**Instruction Address-Compare Registers**, page 246
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 157
ICDBDR	Instruction-Cache Debug-Data Register	**icread Instruction**, page 171
LR	Link Register	**Link Register (LR)**, page 65
PID	Process ID Register	**Process-ID Register**, page 176
PIT	Programmable-Interval Timer	**Programmable-Interval Timer Register**, page 231
PVR	Processor-Version Register	**Processor-Version Register**, page 135
SGR	Storage Guarded Register	**Storage Guarded Register (SGR)**, page 157
SLER	Storage Little-Endian Register	**Storage Little-Endian Register (SLER)**, page 158
SPRG0	SPR General-Purpose Register 0	**SPR General-Purpose Registers**, page 134
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 202
SRR1	Save / Restore Register 1	
SRR2	Save / Restore Register 2	**Save/Restore Registers 2 and 3**, page 203
SRR3	Save / Restore Register 3	
SU0R	Storage User-Defined 0 Register	**Storage User-Defined 0 Register (SU0R)**, page 158
TBL	Time-Base Lower	**Time Base**, page 228
TBU	Time-Base Upper	
TCR	Timer-Control Register	**Timer-Control Register**, page 232
TSR	Timer-Status Register	**Timer-Status Register**, page 233
USPRG0	User SPR General-Purpose Register 0	**User-SPR General-Purpose Register**, page 66
XER	Fixed-Point Exception Register	**Fixed-Point Exception Register (XER)**, page 65
ZPR	Zone-Protection Register	**Zone Protection**, page 185

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 472 lists the special-purpose registers sorted by SPRN and Table A-6, page 473 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
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

Table A-4: **Special-Purpose Registers Sorted by Name *(Continued)***

Name	Descriptive Name	SPRN		SPRF		Privileged	Access	Reset Value
		Dec	Hex	Hex	Binary			
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 470 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

Table B-1: **Instructions Sorted by Mnemonic *(Continued)***

	0	6	9	11	12	14	16	17	20	21	22	26	30	31
cmpli	**10**	**crf**D	**00**		**r**A		**SIMM**							
cntlzw	**31**	**r**S			**r**A		**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**			**r**A		**r**B			**758**				**0**
dcbf	**31**	**00000**			**r**A		**r**B			**86**				**0**
dcbi	**31**	**00000**			**r**A		**r**B			**470**				**0**
dcbst	**31**	**00000**			**r**A		**r**B			**54**				**0**
dcbt	**31**	**00000**			**r**A		**r**B			**278**				**0**
dcbtst	**31**	**00000**			**r**A		**r**B			**246**				**0**
dcbz	**31**	**00000**			**r**A		**r**B			**1014**				**0**
dccci	**31**	**00000**			**r**A		**r**B			**454**				**0**
dcread	**31**	**r**D			**r**A		**r**B			**486**				**0**
divw	**31**	**r**D			**r**A		**r**B		**OE**	**491**				**Rc**
divwu	**31**	**r**D			**r**A		**r**B		**OE**	**459**				**Rc**
eieio	**31**	**00000**			**00000**		**00000**			**854**				**0**
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**
icbi	**31**	**00000**			**r**A		**r**B			**982**				**0**
icbt	**31**	**00000**			**r**A		**r**B			**262**				**0**
iccci	**31**	**00000**			**r**A		**r**B			**966**				**0**
icread	**31**	**00000**			**r**A		**r**B			**998**				**0**
isync	**19**	**00000**			**00000**		**00000**			**150**				**0**
lbz	**34**	**r**D			**r**A		**d**							
lbzu	**35**	**r**D			**r**A		**d**							
lbzux	**31**	**r**D			**r**A		**r**B			**119**				**0**
lbzx	**31**	**r**D			**r**A		**r**B			**87**				**0**
lha	**42**	**r**D			**r**A		**d**							
lhau	**43**	**r**D			**r**A		**d**							
lhaux	**31**	**r**D			**r**A		**r**B			**375**				**0**
lhax	**31**	**r**D			**r**A		**r**B			**343**				**0**

Table B-1: **Instructions Sorted by Mnemonic *(Continued)***

	0	6	9	11	12	14	16	17	20	21	22	26	30	31
lhbrx	31	rD		rA			rB			790				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
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

Table B-1: **Instructions Sorted by Mnemonic *(Continued)***

	0	6	9	11	12	14	16	17	20	21	22	26	30	31
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**
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**

Table B-1: **Instructions Sorted by Mnemonic *(Continued)***

	0	6	9	11	12	14	16	17	20	21	22	26	30	31
sthu	45	rS		rA			d							
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**	**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**
mcrf	**19**	**crf**D	**00**	**crf**S	**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**	**crf**D	**00**	rA	**SIMM**	
cmpi	**11**	**crf**D	**00**	rA	**SIMM**	

Table B-24: **D Form *(Continued)***

	0	6	9	11	16	31
addic	**12**	**rD**		**rA**	**SIMM**	
addic.	**13**	**rD**		**rA**	**SIMM**	
addi	**14**	**rD**		**rA**	**SIMM**	
addis	**15**	**rD**		**rA**	**SIMM**	
ori	**24**	**rS**		**rA**	**UIMM**	
oris	**25**	**rS**		**rA**	**UIMM**	
xori	**26**	**rS**		**rA**	**UIMM**	
xoris	**27**	**rS**		**rA**	**UIMM**	
andi.	**28**	**rS**		**rA**	**UIMM**	
andis.	**29**	**rS**		**rA**	**UIMM**	
lwz	**32**	**rD**		**rA**	**d**	
lwzu	**33**	**rD**		**rA**	**d**	
lbz	**34**	**rD**		**rA**	**d**	
lbzu	**35**	**rD**		**rA**	**d**	
stw	**36**	**rS**		**rA**	**d**	
stwu	**37**	**rS**		**rA**	**d**	
stb	**38**	**rS**		**rA**	**d**	
stbu	**39**	**rS**		**rA**	**d**	
lhz	**40**	**rD**		**rA**	**d**	
lhzu	**41**	**rD**		**rA**	**d**	
lha	**42**	**rD**		**rA**	**d**	
lhau	**43**	**rD**		**rA**	**d**	
sth	**44**	**rS**		**rA**	**d**	
sthu	**45**	**rS**		**rA**	**d**	
lmw	**46**	**rD**		**rA**	**d**	
stmw	**47**	**rS**		**rA**	**d**	

Table B-25: **I Form**

	0	6	30	31
b	**18**	**LI**	**AA**	**LK**

Table B-26: **M Form**

	0	6	11	16	21	26	31
rlwimi	**20**	**rS**	**rA**	**SH**	**MB**	**ME**	**Rc**
rlwinm	**21**	**rS**	**rA**	**SH**	**MB**	**ME**	**Rc**
rlwnm	**23**	**rS**	**rA**	**rB**	**MB**	**ME**	**Rc**

Table B-27: **SC Form**

	0	6	11	16	21	26	30	31
sc	**17**	**00000**	**00000**	**00000**	**00000**	**0000**	**1**	**0**

Table B-28: **X Form**

	0	6	9	11	16	17	21	31
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 272
add.	Add and Record		
addc	Add Carrying		page 273
addc.	Add Carrying and Record		
addco	Add Carrying with Overflow Enabled		
addco.	Add Carrying with Overflow Enabled and Record		
adde	Add Extended		page 274
adde.	Add Extended and Record		
addeo	Add Extended with Overflow Enabled		
addeo.	Add Extended with Overflow Enabled and Record		
addi	Add Immediate		page 275
addic	Add Immediate Carrying		page 276
addic.	Add Immediate Carrying and Record		page 277
addis	Add Immediate Shifted		page 278
addme	Add to Minus One Extended		page 279
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 272
addo.	Add with Overflow Enabled and Record		
addze	Add to Zero Extended		page 280
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 281
and.	AND and Record		
andc	AND with Complement		page 282
andc.	AND with Complement and Record		
andi.	AND Immediate and Record		page 283
andis.	AND Immediate Shifted and Record		page 284
b	Branch		page 285
ba	Branch Absolute		
bc	Branch Conditional		page 286
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 288
bcctrl	Branch Conditional to Count Register and Link		
bcl	Branch Conditional and Link		page 286
bcla	Branch Conditional Absolute and Link		
bclr	Branch Conditional to Link Register		page 290
bclrl	Branch Conditional to Link Register and Link		
bctr	Branch to Count Register	**bcctr**	page 523
bctrl	Branch to Count Register and Link	**bcctrl**	page 524
bdnz	Branch if Decremented CTR Not Zero	**bc**	page 522
bdnza	Branch if Decremented CTR Not Zero Absolute	**bca**	page 522
bdnzf	Branch if Decremented CTR Not Zero and Condition False	**bc**	page 522
bdnzfa	Branch if Decremented CTR Not Zero and Condition False Absolute	**bca**	page 522
bdnzfl	Branch if Decremented CTR Not Zero and Condition False and Link	**bcl**	page 523
bdnzfla	Branch if Decremented CTR Not Zero and Condition False Absolute and Link	**bcla**	page 523
bdnzflr	Branch if Decremented CTR Not Zero and Condition False to Link Register	**bclr**	page 523
bdnzflrl	Branch if Decremented CTR Not Zero and Condition False to Link Register and Link	**bclrl**	page 524
bdnzl	Branch if Decremented CTR Not Zero and Link	**bcl**	page 523
bdnzla	Branch if Decremented CTR Not Zero Absolute and Link	**bcla**	page 523
bdnzlr	Branch if Decremented CTR Not Zero to Link Register	**bclr**	page 523
bdnzlrl	Branch if Decremented CTR Not Zero to Link Register and Link	**bclrl**	page 524
bdnzt	Branch if Decremented CTR Not Zero and Condition True	**bc**	page 522
bdnzta	Branch if Decremented CTR Not Zero and Condition True Absolute	**bca**	page 522
bdnztl	Branch if Decremented CTR Not Zero and Condition True and Link	**bcl**	page 523
bdnztla	Branch if Decremented CTR Not Zero and Condition True Absolute and Link	**bcla**	page 523
bdnztlr	Branch if Decremented CTR Not Zero and Condition True to Link Register	**bclr**	page 523
bdnztlrl	Branch if Decremented CTR Not Zero and Condition True to Link Register and Link	**bclrl**	page 524
bdz	Branch if Decremented CTR Zero	**bc**	page 522
bdza	Branch if Decremented CTR Zero Absolute	**bca**	page 522
bdzf	Branch if Decremented CTR Zero and Condition False	**bc**	page 522
bdzfa	Branch if Decremented CTR Zero and Condition False Absolute	**bca**	page 522
bdzfl	Branch if Decremented CTR Zero and Condition False and Link	**bcl**	page 523
bdzfla	Branch if Decremented CTR Zero and Condition False Absolute and Link	**bcla**	page 523
bdzflr	Branch if Decremented CTR Zero and Condition False to Link Register	**bclr**	page 523
bdzflrl	Branch if Decremented CTR Zero and Condition False to Link Register and Link	**bclrl**	page 524
bdzl	Branch if Decremented CTR Zero and Link	**bcl**	page 523
bdzla	Branch if Decremented CTR Zero Absolute and Link	**bcla**	page 523

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 523
bdzlrl	Branch if Decremented CTR Zero to Link Register and Link	**bclrl**	page 524
bdzt	Branch if Decremented CTR Zero and Condition True	**bc**	page 522
bdzta	Branch if Decremented CTR Zero and Condition True Absolute	**bca**	page 522
bdztl	Branch if Decremented CTR Zero and Condition True and Link	**bcl**	page 523
bdztla	Branch if Decremented CTR Zero and Condition True Absolute and Link	**bcla**	page 523
bdztlr	Branch if Decremented CTR Zero and Condition True to Link Register	**bclr**	page 523
bdztlrl	Branch if Decremented CTR Zero and Condition True to Link Register and Link	**bclrl**	page 524
beq	Branch if Equal	**bc**	page 525
beqa	Branch if Equal Absolute	**bca**	page 525
beqctr	Branch if Equal to Count Register	**bcctr**	page 526
beqctrl	Branch if Equal to Count Register and Link	**bcctrl**	page 527
beql	Branch if Equal and Link	**bcl**	page 526
beqla	Branch if Equal Absolute and Link	**bcla**	page 526
beqlr	Branch if Equal to Link Register	**bclr**	page 526
beqlrl	Branch if Equal to Link Register and Link	**bclrl**	page 527
bf	Branch if Condition False	**bc**	page 522
bfa	Branch if Condition False Absolute	**bca**	page 522
bfctr	Branch if Condition False to Count Register	**bcctr**	page 523
bfctrl	Branch if Condition False to Count Register and Link	**bcctrl**	page 524
bfl	Branch if Condition False and Link	**bcl**	page 523
bfla	Branch if Condition False Absolute and Link	**bcla**	page 523
bflr	Branch if Condition False to Link Register	**bclr**	page 523
bflrl	Branch if Condition False to Link Register and Link	**bclrl**	page 524
bge	Branch if Greater Than or Equal	**bc**	page 525
bgea	Branch if Greater Than or Equal Absolute	**bca**	page 525
bgectr	Branch if Greater Than or Equal to Count Register	**bcctr**	page 526
bgectrl	Branch if Greater Than or Equal to Count Register and Link	**bcctrl**	page 527
bgel	Branch if Greater Than or Equal and Link	**bcl**	page 526
bgela	Branch if Greater Than or Equal Absolute and Link	**bcla**	page 526
bgelr	Branch if Greater Than or Equal to Link Register	**bclr**	page 526
bgelrl	Branch if Greater Than or Equal to Link Register and Link	**bclrl**	page 527
bgt	Branch if Greater Than	**bc**	page 525
bgta	Branch if Greater Than Absolute	**bca**	page 525
bgtctr	Branch if Greater Than to Count Register	**bcctr**	page 526
bgtctrl	Branch if Greater Than to Count Register and Link	**bcctrl**	page 527

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 526
bgtla	Branch if Greater Than Absolute and Link	**bcla**	page 526
bgtlr	Branch if Greater Than to Link Register	**bclr**	page 526
bgtlrl	Branch if Greater Than to Link Register and Link	**bclrl**	page 527
bl	Branch and Link		page 285
bla	Branch Absolute and Link		
ble	Branch if Less Than or Equal	**bc**	page 525
blea	Branch if Less Than or Equal Absolute	**bca**	page 525
blectr	Branch if Less Than or Equal to Count Register	**bcctr**	page 526
blectrl	Branch if Less Than or Equal to Count Register and Link	**bcctrl**	page 527
blel	Branch if Less Than or Equal and Link	**bcl**	page 526
blela	Branch if Less Than or Equal Absolute and Link	**bcla**	page 526
blelr	Branch if Less Than or Equal to Link Register	**bclr**	page 526
blelrl	Branch if Less Than or Equal to Link Register and Link	**bclrl**	page 527
blr	Branch to Link Register	**bclr**	page 523
blrl	Branch to Link Register and Link	**bclrl**	page 524
blt	Branch if Less Than	**bc**	page 525
blta	Branch if Less Than Absolute	**bca**	page 525
bltctr	Branch if Less Than to Count Register	**bcctr**	page 526
bltctrl	Branch if Less Than to Count Register and Link	**bcctrl**	page 527
bltl	Branch if Less Than and Link	**bcl**	page 526
bltla	Branch if Less Than Absolute and Link	**bcla**	page 526
bltlr	Branch if Less Than to Link Register	**bclr**	page 526
bltlrl	Branch if Less Than to Link Register and Link	**bclrl**	page 527
bne	Branch if Not Equal	**bc**	page 525
bnea	Branch if Not Equal Absolute	**bca**	page 525
bnectr	Branch if Not Equal to Count Register	**bcctr**	page 526
bnectrl	Branch if Not Equal to Count Register and Link	**bcctrl**	page 527
bnel	Branch if Not Equal and Link	**bcl**	page 526
bnela	Branch if Not Equal Absolute and Link	**bcla**	page 526
bnelr	Branch if Not Equal to Link Register	**bclr**	page 526
bnelrl	Branch if Not Equal to Link Register and Link	**bclrl**	page 527
bng	Branch if Not Greater Than	**bc**	page 525
bnga	Branch if Not Greater Than Absolute	**bca**	page 525
bngctr	Branch if Not Greater Than to Count Register	**bcctr**	page 526
bngctrl	Branch if Not Greater Than to Count Register and Link	**bcctrl**	page 527

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 526
bngla	Branch if Not Greater Than Absolute and Link	**bcla**	page 526
bnglr	Branch if Not Greater Than to Link Register	**bclr**	page 526
bnglrl	Branch if Not Greater Than to Link Register and Link	**bclrl**	page 527
bnl	Branch if Not Less Than	**bc**	page 525
bnla	Branch if Not Less Than Absolute	**bca**	page 525
bnlctr	Branch if Not Less Than to Count Register	**bcctr**	page 526
bnlctrl	Branch if Not Less Than to Count Register and Link	**bcctrl**	page 527
bnll	Branch if Not Less Than and Link	**bcl**	page 526
bnlla	Branch if Not Less Than Absolute and Link	**bcla**	page 526
bnllr	Branch if Not Less Than to Link Register	**bclr**	page 526
bnllrl	Branch if Not Less Than to Link Register and Link	**bclrl**	page 527
bns	Branch if Not Summary Overflow	**bc**	page 525
bnsa	Branch if Not Summary Overflow Absolute	**bca**	page 525
bnsctr	Branch if Not Summary Overflow to Count Register	**bcctr**	page 526
bnsctrl	Branch if Not Summary Overflow to Count Register and Link	**bcctrl**	page 527
bnsl	Branch if Not Summary Overflow and Link	**bcl**	page 526
bnsla	Branch if Not Summary Overflow Absolute and Link	**bcla**	page 526
bnslr	Branch if Not Summary Overflow to Link Register	**bclr**	page 526
bnslrl	Branch if Not Summary Overflow to Link Register and Link	**bclrl**	page 527
bso	Branch if Summary Overflow	**bc**	page 525
bsoa	Branch if Summary Overflow Absolute	**bca**	page 525
bsoctr	Branch if Summary Overflow to Count Register	**bcctr**	page 526
bsoctrl	Branch if Summary Overflow to Count Register and Link	**bcctrl**	page 527
bsol	Branch if Summary Overflow and Link	**bcl**	page 526
bsola	Branch if Summary Overflow Absolute and Link	**bcla**	page 526
bsolr	Branch if Summary Overflow to Link Register	**bclr**	page 526
bsolrl	Branch if Summary Overflow to Link Register and Link	**bclrl**	page 527
bt	Branch if Condition True	**bc**	page 522
bta	Branch if Condition True Absolute	**bca**	page 522
btctr	Branch if Condition True to Count Register	**bcctr**	page 523
btctrl	Branch if Condition True to Count Register and Link	**bcctrl**	page 524
btl	Branch if Condition True and Link	**bcl**	page 523
btla	Branch if Condition True Absolute and Link	**bcla**	page 523
btlr	Branch if Condition True to Link Register	**bclr**	page 523
btlrl	Branch if Condition True to Link Register and Link	**bclrl**	page 524

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 529
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 292
cmpi	Compare Immediate		page 293
cmpl	Compare Logical		page 294
cmpli	Compare Logical Immediate		page 295
cmplw	Compare Logical Word	**cmpl**	page 528
cmplwi	Compare Logical Word Immediate	**cmpli**	
cmpw	Compare Word	**cmp**	
cmpwi	Compare Word Immediate	**cmpi**	
cntlzw	Count Leading Zeros Word		page 296
cntlzw.	Count Leading Zeros Word and Record		
crand	Condition Register AND		page 297
crandc	Condition Register AND with Complement		page 298
crclr	Condition Register Clear	**crxor**	page 528
creqv	Condition Register Equivalent		page 299
crmove	Condition Register Move	**cror**	page 528
crnand	Condition Register NAND		page 300
crnor	Condition Register NOR		page 301
crnot	Condition Register Not	**crnor**	page 528
cror	Condition Register OR		page 302
crorc	Condition Register OR with Complement		page 303
crset	Condition Register Set	**creqv**	page 528
crxor	Condition Register XOR		page 304
dcba	Data Cache Block Allocate		page 305
dcbf	Data Cache Block Flush		page 307
dcbi	Data Cache Block Invalidate		page 309
dcbst	Data Cache Block Store		page 311
dcbt	Data Cache Block Touch		page 313
dcbtst	Data Cache Block Touch for Store		page 314
dcbz	Data Cache Block Clear to Zero		page 316
dccci	Data Cache Congruence Class Invalidate		page 318

Table B-33: **Complete List of Instruction Mnemonics *(Continued)***

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
dcread	Data Cache Read		page 320
divw	Divide Word		page 322
divw.	Divide Word and Record		
divwo	Divide Word with Overflow Enabled		
divwo.	Divide Word with Overflow Enabled and Record		
divwu	Divide Word Unsigned		page 324
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 325
eqv	Equivalent		page 326
eqv.	Equivalent and Record		
extlwi	Extract and Left Justify Immediate	**rlwinm**	page 529
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 327
extsb.	Extend Sign Byte and Record		
extsh	Extend Sign Halfword		page 328
extsh.	Extend Sign Halfword and Record		
icbi	Instruction Cache Block Invalidate		page 329
icbt	Instruction Cache Block Touch		page 331
iccci	Instruction Cache Congruence Class Invalidate		page 333
icread	Instruction Cache Read		page 334
inslwi	Insert from Left Immediate	**rlwimi**	page 529
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 336
la	Load Address	**addi**	page 534
lbz	Load Byte and Zero		page 337
lbzu	Load Byte and Zero with Update		page 338
lbzux	Load Byte and Zero with Update Indexed		page 339
lbzx	Load Byte and Zero Indexed		page 340
lha	Load Halfword Algebraic		page 341
lhau	Load Halfword Algebraic with Update		page 342

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 343
lhax	Load Halfword Algebraic Indexed		page 344
lhbrx	Load Halfword Byte-Reverse Indexed		page 345
lhz	Load Halfword and Zero		page 346
lhzu	Load Halfword and Zero with Update		page 347
lhzux	Load Halfword and Zero with Update Indexed		page 348
lhzx	Load Halfword and Zero Indexed		page 349
li	Load Immediate	**addi**	page 534
lis	Load Immediate Shifted	**addis**	page 534
lmw	Load Multiple Word		page 350
lswi	Load String Word Immediate		page 352
lswx	Load String Word Indexed		page 354
lwarx	Load Word and Reserve Indexed		page 356
lwbrx	Load Word Byte-Reverse Indexed		page 357
lwz	Load Word and Zero		page 358
lwzu	Load Word and Zero with Update		page 359
lwzux	Load Word and Zero with Update Indexed		page 360
lwzx	Load Word and Zero Indexed		page 361
macchw	Multiply Accumulate Cross Halfword to Word Modulo Signed		page 362
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 363
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 364
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 365
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 366
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 367
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 368
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 369
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 370
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 371
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 372
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 373
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 374
mcrxr	Move to Condition Register from XER		page 375
mfccr0	Move From Core-Configuration Register 0	**mfspr**	page 530
mfcr	Move from Condition Register		page 376
mfctr	Move From Count Register	**mfspr**	page 530
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 377
mfdcwr	Move From Data-Cache Write-Through Register	**mfspr**	page 530
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 378
mfpid	Move From Process ID Register	**mfspr**	page 530
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 379

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 530
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 380
mftbl	Move From Time-Base Lower	**mfspr**	page 530
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 534
mr.	Move Register and Record	**or.**	
mtccr0	Move to Core-Configuration Register 0	**mtspr**	page 530
mtcr	Move to Condition Register	**mtcrf**	page 535
mtcrf	Move to Condition Register Fields		page 381
mtctr	Move to Count Register	**mtspr**	page 530
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 383

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 530
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 384
mtpid	Move to Process ID Register	**mtspr**	page 530
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 385
mtsprg0	Move to SPR General-Purpose Register 0	**mtspr**	page 530
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 530
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 530
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 386
mulchw.	Multiply Cross Halfword to Word Signed and Record		
mulchwu	Multiply Cross Halfword to Word Unsigned		page 387
mulchwu.	Multiply Cross Halfword to Word Unsigned and Record		
mulhhw	Multiply High Halfword to Word Signed		page 388
mulhhw.	Multiply High Halfword to Word Signed and Record		
mulhhwu	Multiply High Halfword to Word Unsigned		page 389
mulhhwu.	Multiply High Halfword to Word Unsigned and Record		
mulhw	Multiply High Word		page 390
mulhw.	Multiply High Word and Record		
mulhwu	Multiply High Word Unsigned		page 391
mulhwu.	Multiply High Word Unsigned and Record		
mullhw	Multiply Low Halfword to Word Signed		page 392
mullhw.	Multiply Low Halfword to Word Signed and Record		
mullhwu	Multiply Low Halfword to Word Unsigned		page 393
mullhwu.	Multiply Low Halfword to Word Unsigned and Record		
mulli	Multiply Low Immediate		page 394
mullw	Multiply Low Word		page 395
mullw.	Multiply Low Word and Record		
mullwo	Multiply Low Word with Overflow Enabled		page 395
mullwo.	Multiply Low Word with Overflow Enabled and Record		
nand	NAND		page 396
nand.	NAND and Record		
neg	Negate		page 397
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 398
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 399
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 400
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 401
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 402
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 403
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 534
nor	NOR		page 404
nor.	NOR and Record		
not	Complement (Not) Register	**nor**	page 535
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 405
or.	OR and Record		
orc	OR with Complement		page 406
orc.	OR with Complement and Record		
ori	OR Immediate		page 407
oris	OR Immediate Shifted		page 408
rfci	Return from Critical Interrupt		page 409
rfi	Return from Interrupt		page 410
rlwimi	Rotate Left Word Immediate then Mask Insert		page 411
rlwimi.	Rotate Left Word Immediate then Mask Insert and Record		
rlwinm	Rotate Left Word Immediate then AND with Mask		page 412
rlwinm.	Rotate Left Word Immediate then AND with Mask and Record		
rlwnm	Rotate Left Word then AND with Mask		page 413
rlwnm.	Rotate Left Word then AND with Mask and Record		
rotlw	Rotate Left	**rlwinm**	page 529
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 414
slw	Shift Left Word		page 415
slw.	Shift Left Word and Record		
slwi	Shift Left Immediate	**rlwinm**	page 529
slwi.	Shift Left Immediate and Record	**rlwinm.**	
sraw	Shift Right Algebraic Word		page 416
sraw.	Shift Right Algebraic Word and Record		
srawi	Shift Right Algebraic Word Immediate		page 417
srawi.	Shift Right Algebraic Word Immediate and Record		
srw	Shift Right Word		page 418
srw.	Shift Right Word and Record		
srwi	Shift Right Immediate	**rlwinm**	page 529
srwi.	Shift Right Immediate and Record	**rlwinm.**	
stb	Store Byte		page 419
stbu	Store Byte with Update		page 420
stbux	Store Byte with Update Indexed		page 421

Table B-33: **Complete List of Instruction Mnemonics *(Continued)***

Mnemonic or Simplified Mnemonic	Instruction Name	Equivalent Mnemonic	Reference
stbx	Store Byte Indexed		page 422
sth	Store Halfword		page 423
sthbrx	Store Halfword Byte-Reverse Indexed		page 424
sthu	Store Halfword with Update		page 425
sthux	Store Halfword with Update Indexed		page 426
sthx	Store Halfword Indexed		page 427
stmw	Store Multiple Word		page 428
stswi	Store String Word Immediate		page 429
stswx	Store String Word Indexed		page 431
stw	Store Word		page 433
stwbrx	Store Word Byte-Reverse Indexed		page 434
stwcx.	Store Word Conditional Indexed		page 436
stwu	Store Word with Update		page 438
stwux	Store Word with Update Indexed		page 439
stwx	Store Word Indexed		page 440
sub	Subtract	**subf**	page 532
sub.	Subtract and Record	**subf.**	
subc	Subtract Carrying	**subfc**	page 532
subc.	Subtract Carrying and Record	**subfc.**	
subco	Subtract Carrying with Overflow Enabled	**subfco**	page 532
subco.	Subtract Carrying with Overflow Enabled and Record	**subfco.**	
subf	Subtract from		page 441
subf.	Subtract from and Record		
subfc	Subtract from Carrying		page 442
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 443
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 444
subfme	Subtract from Minus One Extended		page 445
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 441
subfo.	Subtract from with Overflow Enabled and Record		
subfze	Subtract from Zero Extended		page 446
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 532
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 447
tlbia	TLB Invalidate All		page 448
tlbre	TLB Read Entry		page 449
tlbrehi	Read TLBHI Portion of TLB Entry	**tlbre**	page 532
tlbrelo	Read TLBLO Portion of TLB Entry		
tlbsx	TLB Search Indexed		page 451
tlbsx.	TLB Search Indexed and Record		
tlbsync	TLB Synchronize		page 453
tlbwe	TLB Write Entry		page 454
tlbwehi	Write TLBHI Portion of TLB Entry	**tlbwe**	page 532
tlbwelo	Write TLBLO Portion of TLB Entry		
trap	Trap if Unconditional	**tw**	page 533
tw	Trap Word		page 456
tweq	Trap if Equal	**tw**	page 533
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 458

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 533
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 460
wrteei	Write External Enable Immediate		page 461
xor	XOR		page 462
xor.	XOR and Record		
xori	XOR Immediate		page 463
xoris	XOR Immediate Shifted		page 464

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 522 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 525 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 521.

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 72 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** **r**A, **r**S, *n*, *b* (*n* > 0)	**rlwinm** **r**A, **r**S, *b*, 0, *n*–1
	extlwi. **r**A, **r**S, *n*, *b* (*n* > 0)	**rlwinm.** **r**A, **r**S, *b*, 0, *n*–1
Extract and Right Justify Immediate	**extrwi** **r**A, **r**S, *n*, *b* (*n* > 0)	**rlwinm** **r**A, **r**S, *b*+*n*, 32–*n*, 31
	extrwi. **r**A, **r**S, *n*, *b* (*n* > 0)	**rlwinm.** **r**A, **r**S, *b*+*n*, 32–*n*, 31
Insert from Left Immediate	**inslwi** **r**A, **r**S, *n*, *b* (*n* > 0)	**rlwimi** **r**A, **r**S, 32–*b*, *b*, (*b*+*n*)–1
	inslwi. **r**A, **r**S, *n*, *b* (*n* > 0)	**rlwimi.** **r**A, **r**S, 32–*b*, *b*, (*b*+*n*)–1
Insert from Right Immediate	**insrwi** **r**A, **r**S, *n*, *b* (*n* > 0)	**rlwimi** **r**A, **r**S, 32–(*b*+*n*), *b*, (*b*+*n*)–1
	insrwi. **r**A, **r**S, *n*, *b* (*n* > 0)	**rlwimi.** **r**A, **r**S, 32–(*b*+*n*), *b*, (*b*+*n*)–1
Rotate Left Immediate	**rotlwi** **r**A, **r**S, *n*	**rlwinm** **r**A, **r**S, *n*, 0, 31
	rotlwi. **r**A, **r**S, *n*	**rlwinm.** **r**A, **r**S, *n*, 0, 31
Rotate Right Immediate	**rotrwi** **r**A, **r**S, *n*	**rlwinm** **r**A, **r**S, 32–*n*, 0, 31
	rotrwi. **r**A, **r**S, *n*	**rlwinm.** **r**A, **r**S, 32–*n*, 0, 31
Rotate Left	**rotlw** **r**A, **r**S, **r**B	**rlwnm** **r**A, **r**S, **r**B, 0, 31
	rotlw. **r**A, **r**S, **r**B	**rlwnm.** **r**A, **r**S, **r**B, 0, 31
Shift Left Immediate	**slwi** **r**A, **r**S, *n* (*n* < 32)	**rlwinm** **r**A, **r**S, *n*, 0, 31–*n*
	slwi. **r**A, **r**S, *n* (*n* < 32)	**rlwinm.** **r**A, **r**S, *n*, 0, 31–*n*
Shift Right Immediate	**srwi** **r**A, **r**S, *n* (*n* < 32)	**rlwinm** **r**A, **r**S, 32–*n*, *n*, 31
	srwi. **r**A, **r**S, *n* (*n* < 32)	**rlwinm.** **r**A, **r**S, 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 rD, *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. rA, **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** **r**A, **r**S	**nor** **r**A, **r**S, **r**S
	not. **r**A, **r**S	**nor.** **r**A, **r**S, **r**S

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** **r**S	**mtcrf** 0xFF, **r**S

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 538 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 538 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 540 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 538). 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 72 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 215, 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 148 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.

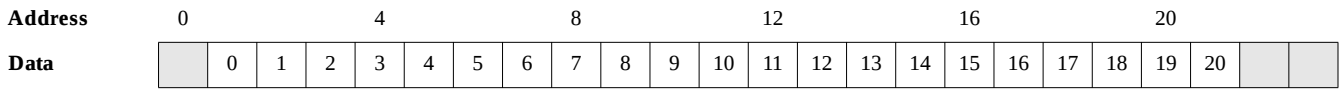


Figure D-1: **String Access Example**

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.

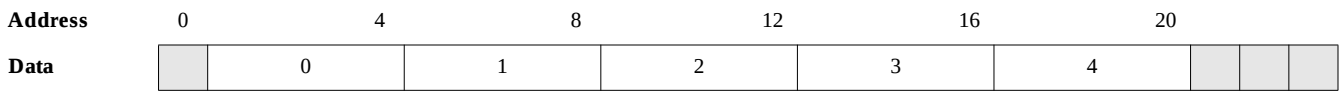


Figure D-2: **Multiple-Word Access Example**

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.

XILINX®

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 551 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 498, 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.

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 498, 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

J

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z