PPC405 Processor User's Manual

Printed in the United States of America, Monday, September 10, 2007

Applied Micro Circuits Corporation
215 Moffett Park Drive, Sunnyvale, CA 94089

Phone: (408) 542-8600 — (800) 840-6055 — Fax: (408) 542-8601

http://www.amcc.com

Contents

Figures

Tables

About This Book

This user’s manual provides the architectural overview, programming model, and detailed information about the registers, the instruction set, and operations of the AMCC PowerPC™ 405 (PPC405) embedded processor. This device contains a 32-bit reduced instruction set computer (RISC) processor.

The PPC405 RISC embedded processor features:

- PowerPC Architecture™
- Single-cycle execution for most instructions
- Instruction cache unit and data cache unit
- Support for little endian operation
- Interrupt interface for one critical and one non-critical interrupt signal
- JTAG interface

Who Should Use This Book

This book is for system hardware and software developers, and for application developers who need to understand the PPC405. The audience should understand network processor design, network system design, operating systems, RISC processing, and design for testability.

How to Use This Book

This book describes the PPC405 device architecture, programming model, external interfaces, internal registers, and instruction set. This book is organized as follows:

This book contains the following appendixes:

To help readers find material in these chapters, the book contains:

Conventions

The following is a list of notational conventions frequently used in this manual.

$\overline{\text{ActiveLow}}$	An overbar indicates an active-low signal.
n	A decimal number
0x*n*	A hexadecimal number
0b*n*	A binary number
=	Assignment
$\wedge$	AND logical operator
$\neg$	NOT logical operator
$\vee$	OR logical operator
$\oplus$	Exclusive-OR (XOR) logical operator
+	Twos complement addition
–	Twos complement subtraction, unary minus
$\times$	Multiplication
$\div$	Division yielding a quotient
%	Remainder of an integer division; (33 % 32) = 1.
$\Vert$	Concatenation
$=, \neq$	Equal, not equal relations
<, >	Signed comparison relations
$\overset{u}{<}, \overset{u}{>}$	Unsigned comparison relations
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.
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.
leave	Leave innermost do loop or do loop specified in a leave statement.
FLD	An instruction or register field
FLD_{b}	A bit in a named instruction or register field
$\text{FLD}_{\text{b:b}}$	A range of bits in a named instruction or register field
$\text{FLD}_{\text{b,b, . . .}}$	A list of bits, by number or name, in a named instruction or register field
REG_{b}	A bit in a named register
$\text{REG}_{\text{b:b}}$	A range of bits in a named register
$\text{REG}_{\text{b,b, . . .}}$	A list of bits, by number or name, in a named register
REG[FLD]	A field in a named register
REG[FLD, FLD . . .]	A list of fields in a named register
REG[FLD:FLD]	A range of fields in a named register

GPR(r)	General Purpose Register (GPR) r, where $0 \le r \le 31$.
(GPR(r))	The contents of GPR r, where $0 \le r \le 31$.
DCR(DCRN)	A Device Control Register (DCR) specified by the DCRF field in an **mfdcr** or **mtdcr** instruction
SPR(SPRN)	An SPR specified by the SPRF field in an **mfspr** or **mtspr** instruction
TBR(TBRN)	A Time Base Register (TBR) specified by the TBRF field in an **mftb** instruction
GPRs	RA, RB, . . .
(Rx)	The contents of a GPR, where *x* is A, B, S, or T
(RA|0)	The contents of the register RA or 0, if the RA field is 0.
CR_{FLD}	The field in the condition register pointed to by a field of an instruction.
$c_{0:3}$	A 4-bit object used to store condition results in compare instructions.
^{n}b	The bit or bit value *b* is replicated *n* times.
xx	Bit positions which are don't-cares.
CEIL(x)	Least integer $\ge$ x.
EXTS(x)	The result of extending *x* on the left with sign bits.
PC	Program counter.
RESERVE	Reserve bit; indicates whether a process has reserved a block of storage.
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.
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.
MS(addr, n)	The number of bytes represented by *n* at the location in main storage represented by *addr*.
EA	Effective address; the 32-bit address, derived by applying indexing or indirect addressing rules to the specified operand, that specifies a location in main storage.
EA_b	A bit in an effective address.
$EA_{b:b}$	A range of bits in an effective address.
ROTL((RS),n)	Rotate left; the contents of RS are shifted left the number of bits specified by *n*.
MASK(MB,ME)	Mask having 1s in positions MB through ME (wrapping if MB > ME) and 0s elsewhere.
instruction(EA)	An instruction operating on a data or instruction cache block associated with an EA.

1. Overview

This document describes the PowerPC™ 405 fixed-point, 32-bit RISC processor, referred to as the PPC405.

This section describes:

- PPC405 processor features
- PPC405 as a 32-bit implementation of Book-E Enhanced PowerPC Architecture.
- Organization of the PPC405 core, including a block diagram and descriptions of the functional units.
- PPC405 core interfaces.

1.1 PPC405 Processor Features

The PPC405 provides high performance and low-power consumption executing at sustained speeds approaching one cycle per instruction. On-chip instruction and data caches reduce chip count and design complexity in systems and improve system throughput. The CPU provides an ideal foundation for systems incorporating system-on-a-chip (SOC) designs. This section provides a list of features that are implemented in the PPC405.

- Five-stage pipeline with single-cycle execution of most instructions, including loads and stores
- Unaligned load/store support to cache arrays, main memory, and on-chip memory (OCM)
- Thirty-two 32-bit general purpose registers (GPRs)
- Static branch prediction
- Hardware multiply/divide for faster integer arithmetic (4-cycle multiply, 35-cycle divide)
- Multiply-accumulate instructions
- Enhanced string and multiple-word handling
- True little endian operation
- Forward and reverse trace from a trigger event
- Storage control
  - Separate, configurable, two-way set-associative 16KB instruction and data cache units
  - Eight words (32 bytes) per cache line
  - Instruction cache unit (ICU) non-blocking during line fills, data cache unit (DCU) non-blocking during line fills and flushes
  - Read and write line buffers
  - Instruction fetch hits are supplied from line buffer
  - Data load/store hits are supplied to line buffer
  - Programmable ICU prefetching of next sequential line into line buffer
  - Programmable ICU prefetching of non cacheable instructions, full line (eight words) or half line (four words)
  - Write-back or write-through DCU write strategies
  - Programmable allocation on loads and stores
  - Operand forwarding during cache line fills
  - Parity detection and reporting for the instruction cache, data cache, and translation lookaside buffer (TLB)
  - Double word instruction fetch from cache
  - Translation of the 4 GB logical address space into physical addresses
- On-Chip Memory (OCM) interface
- Memory management
  - Translation of the 4GB logical address space into physical addresses
  - Independent enabling of instruction and data translation/protection

- Page-level access control using the translation mechanism
- Software control of page replacement strategy
- Additional control over protection using zones
- WIU0GE (write-through, cachability, compressed user-defined 0, guarded, endian) storage attribute control for each virtual memory region
- WIMU0GE storage attribute control for thirty-two real 128MB regions

- Timer support
  - 64-bit time base
  - Programmable interval timer (PIT), fixed interval timer (FIT), and watchdog timers
  - Synchronous external time base clock input
- Debug support
  - Enhanced debug support with logical operators
  - Four instruction address compares (IACs)
  - Two data address compares (DACs)
  - Two data value compares (DVCs)
  - JTAG instruction to write to ICU
  - Forward or backward instruction tracing
- Minimized interrupt latency
- Advanced power management support
- PowerPC User Instruction Set Architecture (UISA) and extensions for embedded applications
- 32-bit DCR interface

1.2 PowerPC Architecture

The PowerPC Architecture comprises three levels of standards:

- PowerPC User Instruction Set Architecture (UISA), including the base user-level instruction set, user-level registers, programming model, data types, and addressing modes. This is referred to as Book I of the PowerPC Architecture.
- PowerPC Virtual Environment Architecture, describing the memory model, cache model, cache control instructions, address aliasing, and related issues. While accessible from the user level, these features are intended to be accessed from within library routines provided by the system software. This is referred to as Book II of the PowerPC Architecture.
- PowerPC Operating Environment Architecture, including the memory management model, supervisor-level registers, and the exception model. These features are not accessible from the user level. This is referred to as Book III of the PowerPC Architecture.

Book I and Book II define the instruction set and facilities available to the application programmer. Book III defines features, such as system-level instructions, that are not directly accessible by user applications. The PowerPC Architecture is described in The PowerPC Architecture: A Specification for a New Family of RISC Processors.

The PowerPC Architecture provides compatibility of PowerPC Book I application code across all PowerPC implementations to help maximize the portability of applications developed for PowerPC processors. This is accomplished through compliance with the first level of the architectural definition, the PowerPC UISA, which is common to all PowerPC implementations.

1.3 PPC405 as a PowerPC Implementation

The PPC405 implements the PowerPC UISA, user-level registers, programming model, data types, addressing modes, and 32-bit fixed-point operations. The PPC405 fully complies with the PowerPC UISA. The UISA 64-bit and floating point operations are not implemented. The floating point operations, which cause exceptions, can then be emulated by software.

Most of the features of the PPC405 processor core are compatible with the PowerPC Virtual Environment and Operating Environment Architectures. The PPC405 processor core also provides a number of optimizations and extensions to these layers of the PowerPC Architecture. The full architecture of the PPC405 is defined by the PowerPC Embedded Environment and the PowerPC User Instruction Set Architecture.

The primary extensions of the PowerPC Architecture defined in the Embedded Environment are:

- A simplified memory management mechanism with enhancements for embedded applications
- An enhanced, dual-level interrupt structure
- An architected DCR address space for integrated peripheral control
- The addition of several instructions to support these modified and extended resources

Some of the specific implementation features of the PPC405 are beyond the scope of the PowerPC Architecture. These features are included to enhance performance, integrate functionality, and reduce system complexity in embedded control applications.

1.4 RISC Processor Core Organization

The processor core consists of a 5-stage pipeline, separate instruction and data cache units, virtual memory management unit (MMU), debug, and interfaces to other functions.

1.4.1 Instruction and Data Cache Controllers

The PPC405 processor core uses a 16-KB instruction cache unit (ICU) and an 16-KB data cache unit (DCU) to enable concurrent accesses and minimize pipeline stalls. Both cache units are two-way set-associative and use a 32-byte line size. The instruction set provides a rich assortment of cache control instructions, including instructions to read tag information and data arrays. See Chapter 4, “Cache Operations,” for detailed information about the ICU and DCU.

1.4.1.1 Instruction Cache Unit

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

The ICU can forward any or all of the words of a line fill to the EXU to minimize pipeline stalls caused by cache misses. The ICU aborts speculative fetches abandoned by the EXU, eliminating unnecessary line fills and enabling the ICU to handle the next EXU fetch. Aborting abandoned requests also eliminates unnecessary PLB activity to increase PLB availability for other on-chip cores, such as the DMA controller.

1.4.1.2 Data Cache Unit

The DCU transfers 1, 2, 3, 4, or 8 bytes per cycle, depending on the number of byte enables presented by the CPU. The DCU contains a single-element command and store data queue to reduce pipeline stalls; this queue enables the DCU to independently process load/store and cache control instructions. Dynamic PLB request prioritization reduces pipeline stalls even further. When the DCU is busy with a low-priority request while a subsequent storage operation requested by the CPU is stalled, the DCU automatically increases the priority of the current request to the PLB.

The DCU uses a two-line flush queue to minimize pipeline stalls caused by cache misses. Line flushes are postponed until after a line fill is completed. Registers comprise the first position of the flush queue; the line buffer built into the output of the array for manufacturing test serves as the second position of the flush queue. Pipeline stalls are further reduced by forwarding the requested word to the CPU during the line fill. Single-queued flushes are non-blocking. When a flush operation is pending, the DCU can continue to access the array to determine subsequent load or store hits. Under these conditions, load hits can occur concurrently with store hits to write-back memory without stalling the pipeline. Requests abandoned by the CPU can also be aborted by the cache controller.

Additional DCU features enable the programmer to tailor performance for a given application. The DCU can function in write-back or write-through mode, as controlled by the Data Cache Write-through Register (DCWR) or the translation look-aside buffer (TLB). DCU performance can be tuned to balance performance and memory coherency. Store-without-allocate, controlled by the SWOA field of the Core Configuration Register 0 (CCR0), can inhibit line fills caused by store misses to further reduce potential pipeline stalls and unwanted external bus traffic. Similarly, load-without-allocate, controlled by CCR0[LWOA], can inhibit line fills caused by load misses.

1.4.2 Memory Management Unit

The 4GB address space of the PPC405 is presented as a flat address space.

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

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

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

The translation lookaside buffer (TLB) is the hardware resource that controls translation and protection. It consists of 64 entries, each specifying a page to be translated. The TLB is fully associative; a page entry can be placed anywhere in the TLB. The translation function of the MMU occurs pre-cache for data accesses. Cache tags and indexing use physical addresses for data accesses; instruction fetches are virtually indexed and physically tagged.

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

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

To improve performance, 4 instruction-side and 8 data-side TLB entries are kept in shadow arrays. The shadow arrays prevent TLB contention. Hardware manages the replacement and invalidation of shadow-TLB entries; no system software action is required. The shadow arrays can be thought of as level 1 TLBs, with the main TLB serving as a level 2 TLB.

When address translation is enabled, the translation mechanism provides a basic level of protection. Physical addresses not mapped by a page entry are inaccessible when translation is enabled. Read access is implied by the existence of the valid entry in the TLB. The EX and WR bits in the TLB entry further define levels of access for the page, by permitting execute and write access, respectively.

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

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

When address translation is enabled (MSR[IR, DR] = 1), storage attribute control bits in the TLB control the storage attributes associated with the current page. When address translation is disabled (MSR[IR, DR] = 0), bits in each storage attribute control register control the storage attributes associated with storage regions. Each storage attribute control register contains 32 fields. Each field sets the associated storage attribute for a 128MB memory region. See the topic *Real-Mode Storage Attribute Control* in the *PPC405 Processor User's Manual* for details about the storage attribute control registers.

1.4.3 Debug

The processor core debug facilities include debug modes for the various types of debugging used during hardware and software development. Also included are debug events that allow developers to control the debug process. Debug modes and debug events are controlled using debug registers in the chip. 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 test.

The debug modes, events, controls, and interfaces provide a powerful combination of debug facilities for hardware and software development tools.

1.4.3.1 Development Tool Support

The PPC405 is supported by a wide range of hardware and software development tools.

An operating system debugger is an example of an operating system-aware debugger, implemented using software traps.

1.4.3.2 Debug Modes

The internal, external, real-time-trace, and debug wait modes support a variety of debug tool used in embedded systems development. These debug modes are described in detail in *Debug Modes* on page 139.

1.4.4 Processor Core Interfaces

The processor core provides a range of I/O interfaces.

1.4.4.1 Processor Local Bus

The PLB-compliant interface provides separate 32-bit address and 64-bit data buses for the instruction and data sides.

1.4.4.2 Device Control Register Bus

The Device Control Register (DCR) bus interface provides access to on-chip registers for configuration and status of peripherals such as OCM and DMA.

These registers are accessed using the **mfdcr** and **mtdcr** instructions.

1.4.4.3 Clock and Power Management

This interface supports several methods of clock distribution and power management.

1.4.4.4 JTAG

The JTAG port is enhanced to support the attachment of a debug tool such as the RISCWatch product. Through the JTAG test access port, a debug tool can single-step the processor and interrogate internal processor state to facilitate software debugging. The enhancements comply with the IEEE 1149.1 specification for vendor-specific extensions, and are therefore compatible with standard JTAG hardware for boundary-scan system testing.

1.4.4.5 Interrupts

The PPC405 provides an interface to the UIC, an on-chip interrupt controller that is logically outside the processor. The UIC combines asynchronous interrupt inputs from on-chip and off-chip sources and presents them to the processor core using a pair of interrupt signals: critical and non-critical.

1.4.4.6 On-Chip Memory

The on-chip memory (OCM) interface supports the implementation of instruction- and data-side memory that can be accessed at performance levels matching the cache arrays.

1.5 Processor Programming Model

The programming model is described in detail in *Programming Model* on page 31.

The PowerPC instruction set and Special Purpose Registers (SPRs) provide a high degree of user control over configuration and operation of the processor core functional units.

1.5.1 Data Types

Processor core operands are bytes, halfwords, and words. Multiple words or strings of bytes can be transferred using the load/store multiple and load/store string instructions. Data is represented in twos complement notation or in unsigned fixed-point format.

The address of a multibyte operand is always the lowest memory address occupied by that operand. Byte ordering can be selected as big endian (the lowest memory address of an operand contains its most significant byte) or as little endian (the lowest memory address of an operand contains its least significant byte).

1.5.2 Processor Register Set Summary

The processor core registers can be grouped into basic categories based on function and access mode: General Purpose Registers (GPRs), Special Purpose Registers (SPRs), the Machine State Register (MSR), the Condition Register (CR), and Device Control Registers (DCRs).

Register Summary on page 353 provides lists of all registers provided by the processor..

1.5.2.1 General Purpose Registers

The processor core contains 32 GPRs; each register contains 32 bits. The contents of the GPRs can be transferred from memory using load instructions and stored to memory using store instructions. GPRs, which are specified as operands in many instructions, can also receive instruction results and the contents of other registers.

1.5.2.2 Special Purpose Registers

Special Purpose Registers (SPRs), which are part of the PowerPC Architecture, are accessed using the mtspr and mfspr instructions. SPRs control the use of the debug facilities, timers, interrupts, storage control attributes, and other architected processor resources.

All SPRs are privileged (unavailable to user-mode programs), except the Count Register (CTR), the Link Register (LR), SPR General Purpose Registers (SPRG4–SPRG7, read-only), and the Fixed Point Exception Register (XER). Note that access to the Time Base Lower (TBL) and Time Base Upper (TBU) registers, when addressed as SPRs, is write-only and privileged. However, when addressed as Time Base Registers (TBRs), read access to these registers is not privileged. See *The Time Base* on page 41 for more information.

1.5.2.3 Machine State Register

The PPC405 processor core contains a 32-bit Machine State Register (MSR). The contents of a GPR can be written to the MSR using the **mtmsr** instruction, and the MSR contents can be read into a GPR using the **mfmsr** instruction. The MSR contains fields that control the operation of the processor core.

1.5.2.4 Condition Register

The PPC405 processor core contains a 32-bit Condition Register (CR). These bits are grouped into eight 4-bit fields, CR[CR0]–CR[CR7]. Instructions are provided to perform logical operations on CR fields and bits within fields and to test CR bits within fields. The CR fields, which are set by compare instructions, can be used to control branches. CR[CR0] can be set implicitly by arithmetic instructions.

1.5.2.5 Device Control Registers

DCRs, which are architecturally outside of the processor core, are accessed using the **mtdcr** and **mfdcr** instructions. DCRs are used to control, configure, and hold status for various functional units that are not part of the processor core.

The **mtdcr** and **mfdcr** instructions are privileged, for all DCRs. Therefore, all accesses to DCRs are privileged. See *User and Supervisor Modes* on page 56 for details.

1.5.3 Memory-Mapped I/O Registers

The memory-mapped I/O (MMIO) registers are accessed using load and store instructions. MMIO registers, which are outside processor core and which are not architected, are used to control, configure, and hold status for various functional units that are not part of the processor core.

1.5.4 Addressing Modes

The processor core supports the following addressing modes, which enable efficient retrieval and storage of data in memory:

- Base plus displacement addressing
- Indexed addressing
- Base plus displacement addressing and indexed addressing, with update

In the base plus displacement addressing mode, an effective address (EA) is formed by adding a displacement to a base address contained in a GPR (or to an implied base of 0). The displacement is an immediate field in an instruction.

In the indexed addressing mode, the EA is formed by adding an index contained in a GPR to a base address contained in a GPR (or to an implied base of 0).

The base plus displacement and the indexed addressing modes also have a “with update” mode. In “with update” mode, the effective address calculated for the current operation is saved in the base GPR, and can be used as the base in the next operation. The “with update” mode relieves the processor from repeatedly loading a GPR with an address for each piece of data, regardless of the proximity of the data in memory.

2. Programming Model

The programming model of the PPC405 describes how the following features and operations of the processor appear to programmers:

- Memory organization and addressing, page 31
- Registers, page 32
- Data types and alignment, page 42
- Byte ordering, page 44
- Instruction processing, page 49
- Branch processing, page 50
- Speculative accesses, page 53
- Privileged mode operation, page 56.
- Synchronization, page 58
- Instruction set, page 61

2.1 User and Privileged Programming Models

The PPC405 executes programs in two modes, also referred to as states. Programs running in privileged mode (also referred to as the supervisor state) can access any register and execute any instruction. These instructions and registers comprise the privileged programming model. In user mode, certain registers and instructions are unavailable to programs. This is also called the problem state. Those registers and instructions that are available comprise the user programming model.

Privileged mode provides operating system software access to all processor resources. Because access to certain processor resources is denied in user mode, application software runs in user mode. Operating system software and other application software is protected from the effects of an errant application program.

Throughout this book, the terms user program and privileged programs are used to associate programs with one of the programming models. Registers and instructions are described as user or privileged. Privileged mode operation is described in detail in *User and Supervisor Modes* on page 56.

2.2 Storage Addressing

As a 32-bit implementation of the Book-E Enhanced PowerPC Architecture, the PPC405 implements a uniform 32-bit effective address (EA) space. Effective addresses are expanded into virtual addresses and then translated to 36-bit (64GB) real addresses by the memory management unit (see *Memory Management* on page 91 for more information on the translation process).

The PPC405 generates an effective address whenever it executes a storage access, branch, cache management, or translation look aside buffer (TLB) management instruction, or when it fetches the next sequential instruction.

2.2.1 Storage Attributes

The PowerPC Architecture defines storage attributes that control data and instruction accesses. Storage attributes are provided to control cache write-through policy (the W storage attribute), cachability (the I storage attribute), memory coherency in multiprocessor environments (the M storage attribute), and guarding against speculative memory accesses (the G storage attribute). The PowerPC Embedded Environment defines additional storage attributes for storage compression (the U0 storage attribute) and byte ordering (the E storage attribute).

The PPC405 provides two control mechanisms for the W, I, U0, G, and E attributes. Because the PPC405 does not provide hardware support for multiprocessor environments, the M storage attribute, when present, has no effect.

When the PPC405 operates in virtual mode (address translation is enabled), each storage attribute is controlled by the W, I, U0, G, and E fields in the translation lookaside buffer (TLB) entry for each memory page. The size of memory pages, and hence the size of storage attribute control regions, is variable. Multiple sizes can be in effect simultaneously on different pages.

When the PPC405 operates in real mode (address translation is disabled), storage attribute control registers control the corresponding storage attributes. These registers are:

- Data Cache Write-through Register (DCWR)
- Data Cache Cachability Register (DCCR)
- Instruction Cache Cachability Register (ICCR)
- Storage Guarded Register (SGR)
- Storage Little-Endian Register (SLER)
- Storage User-defined 0 Register (SU0R)

Each storage attribute control register contains 32 bits; each bit controls one of thirty-two 128MB storage attribute control regions. Bit 0 of each register controls the lowest-order region, with ascending bits controlling ascending regions in memory. The storage attributes in each storage attribute region are set independently of each other and of the storage attributes for other regions.

2.3 Registers

All PPC405 registers are identified in this section. Some of the frequently-used registers are described in detail. Other registers are covered in their respective topic chapters (for example, the cache registers are described in *Cache Operations* on page 69). All processor registers are summarized in *Register Summary* on page 353.

The registers are grouped into categories: General Purpose Registers (GPRs), Special Purpose Registers (SPRs), Time Base Registers (TBRs), the Machine State Register (MSR), the Condition Register (CR), Device Control Registers (DCRs), and memory-mapped I/O registers (MMIO). Different instructions are used to access each category of registers.

Processor registers are covered in this book. The DCRs ands MMIO registrers are covered in the user’s manual for the chip in which this processor is instantiated.

For all registers with fields marked as reserved, the reserved fields should be written as 0 and read as undefined. That is, when writing to a register with a reserved field, write a 0 to the reserved field. When reading from a register with a reserved field, ignore that field.

Programming Note: Programming Note: A good coding practice is to perform the initial write to a register with reserved fields as described, and to perform all subsequent writes to the register using a read-modify-write strategy: read the register, use logical instructions to alter defined fields, leaving reserved fields unmodified, and write the register.

Figure 2-1 illustrates the registers in the user and supervisor programming models.

Figure 2-1. PPC405 Programming Model—Registers

User Model

General-Purpose Registers

GPR0	
GPR1	
• • •	
GPR31	

SPR General Registers (read-only)

SPRG4	SPR 0x104
SPRG5	SPR 0x105
SPRG5	SPR 0x106
SPRG7	SPR 0x107

User SPR General Register 0 (read/write)

USPRG0	SPR 0x100

Condition Register

CR	

Fixed-Point Exception Register

XER	SPR 0x001

Link Register

LR	SPR 0x008

Count Register

CTR	SPR 0x009

Time Base Registers (read-only)

TBL	TBR 0x10C
TBU	TBR 0x10D

Storage Attribute Control Registers

DCCR	SPR 0x3FA
DCWR	SPR 0x3BA
ICCR	SPR 0x3FB
SGR	SPR 0x3B9
SLER	SPR 0x3BB
SU0R	SPR 0x3BC

Supervisor Model

Machine State Register

MSR	

Core Configuration Register

CCR0	SPR 0x3B3

SPR General Registers

SPRG0	SPR 0x110
SPRG1	SPR 0x111
SPRG2	SPR 0x112
SPRG3	SPR 0x113
SPRG4	SPR 0x114
SPRG5	SPR 0x115
SPRG6	SPR 0x116
SPRG7	SPR 0x117

Exception Handling Registers

Exception Vector Prefix Register

EVPR	SPR 0x3D5

Exception Syndrome Register

ESR	SPR 0x3D4

Data Exception Address Register

DEAR	SPR 0x3D5

Save/Restore Registers

SRR0	SPR 0x01A
SRR1	SPR 0x01B
SRR2	SPR 0x3DE
SRR3	SPR 0x3DF

Memory Management Registers

Process ID

PID	SPR 0x3B1

Zone Protection Register

ZPR	SPR 0x3B0

Processor Version Register

PVR	SPR 0x11F

Timer Facilities

Time Base Registers

TBL	SPR 0x11C
TBU	SPR 0x11D

Timer Control Register

TCR	SPR 0x3DA

Timer Status Register

TSR	SPR 0x3D8

Programmable Interval Timer

PIT	SPR 0x3DB

Debug Registers

Debug Status Register

DBSR	SPR 0x3F0

Debug Control Registers

DBCR0	SPR 0x3F2
DBCR1	SPR 0x3BD

Data Address Compares

DAC1	SPR 0x3F6
DAC2	SPR 0x3F7

Data Value Compares

DVC1	SPR 0x3B6
DVC2	SPR 0x3B7

Instruction Address Compares

IAC1	SPR 0x3F4
IAC2	SPR 0x3F5
IAC3	SPR 0x3B4
IAC4	SPR 0x3B5

Instruction Cache Debug Data Register

ICDBR	SPR 0x3D3

2.3.1 General Purpose Registers (GPR0-GPR31)

The PPC405 contains thirty-two 32-bit general purpose registers (GPRs). Data from memory can be read into GPRs using load instructions and the contents of GPRs can be written to memory using store instructions. Most integer instructions use GPRs for source and destination operands. See *Table 10-1* on page 353 for the numbering of the GPRs.

Figure 2-3. General Purpose Registers (GPR0-GPR31)

0:31		General Purpose Register data

2.3.2 Special Purpose Registers (SPR)

Special purpose registers (SPRs), which are part of the PowerPC Architecture and the PowerPC Embedded Environment, are accessed using the **mtspr** and **mfspr** instructions.

SPRs control the operation of debug facilities, timers, interrupts, storage control attributes, and other architected processor resources. *Table 10-3* on page 354 shows the mnemonic, name, and number for each SPR. *Table 2-1* on page 36, lists the PPC405 SPRs by function and indicates the pages where the SPRs are described more fully.

Except for the Link Register (LR), the Count Register (CTR), the Fixed-point Exception Register (XER), User SPR General 0 (USPRG0, and read access to SPR General 4–7 (SPRG4–SPRG7), all SPRs are privileged. As SPRs, the registers TBL and TBU are privileged write-only; as TBRs, these registers can be read in user mode. Unless used to access non-privileged SPRs, attempts to execute **mfspr** and **mtspr** instructions while in user mode cause privileged violation program interrupts. See *Privileged SPRs* on page 57.

Table 2-1. PPC405 SPRs

Function	Register				Access	Page
Configuration	CCR0				Privileged	77
Branch Control	CTR				User	36
	LR				User	37
Debug	DAC1	DAC2			Privileged	147
	DBCR0	DBCR1			Privileged	143
	DBSR				Privileged	145
	DVC1	DVC2			Privileged	147
	IAC1	IAC2	IAC3	IAC4	Privileged	147
	ICDBDR				Privileged	80
Fixed-point Exception	XER				User	37
General-Purpose SPR	SPRG0	SPRG1	SPRG2	SPRG3	Privileged	39
	SPRG4	SPRG5	SPRG6	SPRG7	User read, privileged write	39
	USPRG0				User	39
Interrupts and Exceptions	DEAR				Privileged	118
	ESR				Privileged	116
	EVPR				Privileged	116
	SRR0	SRR1			Privileged	115
	SRR2	SRR3			Privileged	115
Processor Version	PVR				Privileged, read-only	39
Storage Attribute Control	DCCR				Privileged	106
	DCWR				Privileged	106
	ICCR				Privileged	107
	SGR				Privileged	107
	SLER				Privileged	107
	SU0R				Privileged	107
Timer Facilities	TBL	TBU			Privileged, write-only	130
	PIT				Privileged	131
	TCR				Privileged	135
	TSR				Privileged	135
Zone Protection	ZPR				Privileged	103

2.3.2.1 Count Register (CTR)

The CTR is written from a GPR using **mtspr**. The CTR contents can be used as a loop count that is decremented and tested by some branch instructions. Alternatively, the CTR contents can specify a target address for the **bcctr** instruction, enabling branching to any address.

The CTR is in the user programming model.

Figure 2-4. Count Register (CTR)

0:31		Count	Used as count for branch conditional with decrement instructions, or as address for branch-to-counter instructions.

2.3.2.2 Link Register (LR)

The LR is written from a GPR using mtspr, and by branch instructions that have the LK bit set to 1. Such branch instructions load the LR with the address of the instruction following the branch instruction. Thus, the LR contents can be used as the return address for a subroutine that was called using the branch.

The LR contents can be used as a target address for the bclr instruction. This allows branching to any address.

When the LR contents represent an instruction address, LR30:31 are assumed to be 0, because all instructions must be word-aligned. However, when LR is read using mfspr, all 32 bits are returned as written.
The LR is in the user programming model.

Figure 2-5. Link Register (LR)

0:31		Link Register contents	If (LR) represents an instruction address, $LR_{30:31}$ should be 0.

2.3.2.3 Fixed Point Exception Register (XER)

The XER records overflow and carry conditions generated by integer arithmetic instructions.

The Summary Overflow (SO) field is set to 1 when instructions cause the Overflow (OV) field to be set to 1. The SO field does not necessarily indicate that an overflow occurred on the most recent arithmetic operation, but that an overflow occurred since the last clearing of XER[SO]. **mtspr**(XER) sets XER[SO, OV] to the value of bit positions 0 and 1 in the source register, respectively.

Once set, XER[SO] is not reset until an **mtspr**(XER) is executed with data that explicitly puts a 0 in the SO bit, or until an **mcrxr** instruction is executed.

XER[OV] is set to indicate whether an instruction that updates XER[OV] produces a result that “overflows” the 32-bit target register. XER[OV] = 1 indicates overflow. For arithmetic operations, this occurs when an operation has a carry-in to the most-significant bit of the result that does not equal the carry-out of the most-significant bit (that is, the exclusive-or of the carry-in and the carry-out is 1).

The following instructions set XER[OV] differently. The specific behavior is indicated in the instruction descriptions in Chapter 24, “Instruction Set.”

- Move instructions:

  mcrxr, mtspr(XER)

- Multiply and divide instructions:

  mullwo, mullwo., divwo, divwo., divwuo, divwuo

The Carry (CA) field is set to indicate whether an instruction that updates XER[CA] produces a result that has a carry-out of the most-significant bit. XER[CA] = 1 indicates a carry.

The following instructions set XER[CA] differently.The specific behavior is indicated in the instruction descriptions in Chapter 24, “Instruction Set.”

- Move instructions

  mcrxr, mtspr(XER)

- • Shift-algebraic operations

  sraw, srawi

The Transfer Byte Count (TBC) field is the byte count for load/store string instructions.

The XER is part of the user programming model.

Figure 2-6. Fixed Point Exception Register (XER)

0	SO	Summary Overflow 0 No overflow has occurred. 1 Overflow has occurred.	Can be *set* by **mtspr** or by using "o" form instructions; can be *reset* by **mtspr** or by **mcrxr**.
1	OV	Overflow 0 No overflow has occurred. 0 Overflow has occurred.	Can be *set* by **mtspr** or by using "o" form instructions; can be *reset* by **mtspr**, by **mcrxr**, or "o" form instructions.
2	CA	Carry 0 Carry has not occurred. 1 Carry has occurred.	Can be *set* by **mtspr** or arithmetic instructions that update the CA field; can be *reset* by **mtspr**, by **mcrxr**, or by arithmetic instructions that update the CA field.
3:24		Reserved	
25:31	TBC	Transfer Byte Count	Used by **lswx** and **stswx**; written by **mtspr**.

Table 2-2 and *Table 2-3* list the PPC405 instructions that update the XER. In the tables, the syntax "[o]" indicates that the instruction has an "o" form that updates XER[SO,OV], and a "non-o" form. The syntax "[.]" indicates that the instruction has a "record" form that updates CR[CR0] (see "Condition Register (CR)" on page 39), and a "non-record" form.

Table 2-2. XER[CA] Updating Instructions

Integer Arithmetic		Integer Shift	Processor Control
Add	**Subtract**	**Shift Right Algebraic**	**Register Management**
addc[**o**][**.**] **adde**[**o**][**.**] **addic**[**.**] **addme**[**o**][**.**] **addze**[**o**][**.**]	**subfc**[**o**][**.**] **subfe**[**o**][**.**] **subfic** **subfme**[**o**][**.**] **subfze**[**o**][**.**]	**sraw**[**.**] **srawi**[**.**]	mtspr mcrxr

Table 2-3. XER[SO,OV] Updating Instructions

Integer Arithmetic					Auxiliary Processor		Processor Control
Add	**Subtract**	**Multiply**	**Divide**	**Negate**	**Multiply-Accumulate**	**Negative Multiply-Accumulate**	**Register Management**
addo[**.**] **addco**[**.**] **addeo**[**.**] **addmeo**[**.**] **addzeo**[**.**]	**subfo**[**.**] **subfco**[**.**] **subfeo**[**.**] **subfmeo**[**.**] **subfzeo**[**.**]	**mullwo**[**.**]	**divwo**[**.**] **divwuo**[**.**]	**nego**[**.**]	**macchwo**[**.**] **macchws**o[**.**] **macchwsu**o[**.**] **macchwu**o[**.**] **machhw**o[**.**] **machhws**o[**.**] **machhwsu**o[**.**] **machhwu**o[**.**] **maclhw**o[**.**] **maclhws**o[**.**] **maclhwsu**o[**.**] **maclhwu**o[**.**]	**nmacchw**o[**.**] **nmacchws**o[**.**] **nmachhw**o[**.**] **nmachhws**o[**.**] **nmaclhw**o[**.**] **nmaclhws**o[**.**]	mtspr mcrxr

2.3.2.4 Special Purpose Registers (USPRG0 and SPRG0–SPRG7)

USPRG0 and SPRG0–SPRG7 are provided for general purpose software use. For example, these registers are used as temporary storage locations. For example, an interrupt handler might save the contents of a GPR to an SPRG, and later restore the GPR from it. This is faster than a save/restore to a memory location. These registers are written using **mtspr** and read using **mfspr**.

Access to USPRG0 is non-privileged for both read and write.

Access to SPRG0–SPRG7 is privileged, except for read access to SPRG4–SPRG7. See *Privileged SPRs* on page 57 for more information.

Figure 2-7. Special Purpose Register General (SPRG0–SPRG7)

0:31		General data	Software value; no hardware usage.

2.3.2.5 Processor Version Register (PVR)

The PVR is a read-only register that uniquely identifies a standard product or Core+ASIC implementation. Software can examine the PVR to recognize implementation-dependent features and determine available hardware resources.

Access to the PVR is privileged. See “Privileged SPRs” on page 57 for more information.

Figure 2-8. Processor Version Register (PVR)

0:31		Assigned PVR value

2.3.3 Condition Register (CR)

The CR contains eight 4-bit fields (CR0–CR7), as shown in *Figure 2-9*. The fields contain conditions detected during the execution of integer or logical compare instructions. The CR contents can be used in conditional branch instructions.

The CR can be modified in any of the following ways:

- **mtcrf** sets specified CR fields by writing to the CR from a GPR, under control of a mask specified as an instruction field.
- **mcrf** sets a specified CR field by copying another CR field to it.
- **mcrxr** copies certain bits of the XER into a designated CR field, and then clears the corresponding XER bits.
- The “with update” forms of integer instructions implicitly update CR[CR0].
- Integer compare instructions update a specified CR field.
- The CR-logical instructions update a specified CR bit with the result of a logical operation on a specified pair of CR bit fields.
- Conditional branch instructions can test a CR bit as one of the branch conditions.

If a CR field is set by a compare instruction, the bits are set as described in the next section.

The CR is part of the user programming model.

Figure 2-9. Condition Register (CR)

0:3	CR0	Condition Register Field 0
4:7	CR1	Condition Register Field 1
8:11	CR2	Condition Register Field 2
12:15	CR3	Condition Register Field 3
16:19	CR4	Condition Register Field 4
20:23	CR5	Condition Register Field 5
24:27	CR6	Condition Register Field 6
28:31	CR7	Condition Register Field 7

2.3.3.1 CR Fields After Compare Instructions

Compare instructions compare the values of two 32-bit registers. The two types of compare instructions, arithmetic and logical, are distinguished by the interpretation given to the 32-bit values. For arithmetic compares, the values are considered to be signed, where 31 bits represent the magnitude and the most-significant bit is a sign bit. For logical compares, the values are considered to be unsigned, so all 32 bits represent magnitude. There is no sign bit. As an example, consider the comparison of 0 with 0xFFFFFFFF. In an arithmetic compare, 0 is larger, because 0xFFFF FFFF represents –1; in a logical compare, 0xFFFFFFFF is larger.

A compare instruction can direct its CR update to any CR field. The first data operand of a compare instruction specifies a GPR. The second data operand specifies another GPR, or immediate data derived from the IM field of the immediate instruction form. The contents of the GPR specified by the first data operand are compared with the contents of the GPR specified by the second data operand (or with the immediate data). See descriptions of the compare instructions (page 24-34 through page 24-37) for precise details.

LT (bit 0)	The first operand is less than the second operand.
GT (bit 1)	The first operand is greater than the second operand.
EQ (bit 2)	The first operand is equal to the second operand.
SO (bit 3)	Summary overflow; a copy of XER[SO].

2.3.3.2 The CR0 Field

After the execution of compare instructions that update CR[CR0], CR[CR0] is interpreted as described in "CR Fields After Compare Instructions" on page 40. The "dot" forms of arithmetic and logical instructions also alter CR[CR0]. After most instructions that update CR[CR0], the bits of CR0 are interpreted as follows:

LT (bit 0)	Less than 0; set if the most-significant bit of the 32-bit result is 1.
GT (bit 1)	Greater than 0; set if the 32-bit result is non-zero and the most-significant bit of the result is 0.
EQ (bit 2)	Equal to 0; set if the 32-bit result is 0.
SO (bit 3)	Summary overflow; a copy of XER[SO] at instruction completion.

The CR[CR0]LT, GT, EQ subfields are set as the result of an algebraic comparison of the instruction result to 0, regardless of the type of instruction that sets CR[CR0]. If the instruction result is 0, the EQ subfield is set to 1. If the result is not 0, either LT or GT is set, depending on the value of the most significant bit of the result.

When updating CR[CR0], the most significant bit of an instruction result is considered a sign bit, even for instructions that produce results that are not usually thought of as signed. For example, logical instructions such as and., or., and nor. update CR[CR0]LT, GT, EQ using such an arithmetic comparison to 0, although the result of such a logical operation is not actually an arithmetic result.

If an arithmetic overflow occurs, the “sign” of an instruction result indicated in CR[CR0]LT, GT, EQ might not represent the “true” (infinitely precise) algebraic result of the instruction that set CR0. For example, if an add. instruction adds two large positive numbers and the magnitude of the result cannot be represented as a twos-complement number in a 32-bit register, an overflow occurs and CR[CR0]LT, SO are set, although the infinitely precise result of the add is positive.

Adding the largest 32-bit twos-complement negative number, 0x8000 0000, to itself results in an arithmetic overflow and 0x0000 0000 is recorded in the target register. CR[CR0]EQ, SO is set, indicating a result of 0, but the infinitely precise result is negative.

The CR[CR0]SO subfield is a copy of XER[SO]. Instructions that do not alter the XER[SO] bit cannot cause an overflow, but even for these instructions CR[CR0]SO is a copy of XER[SO].

Some instructions set CR[CR0] differently or do not specifically set any of the subfields. These instructions include:

- Compare instructions

  cmp, cmpi, cmpl, cmpli
- CR logical instructions

  crand, crandc, creqv, crnand, crnor, cror, crorc, crxor, mcrf
- Move CR instructions

  mtcrf, mcrxr
- stwcx.

The instruction descriptions provide detailed information about how the listed instructions alter CR[CR0].

2.3.4 The Time Base

The PowerPC Architecture provides a 64-bit time base. *Time Base* on page 130 describes the architected time base. Access to the time base is through two 32-bit time base registers (TBRs). The least-significant 32 bits of the time base are read from the Time Base Lower (TBL) register and the most-significant 32 bits are read from the Time Base Upper (TBU) register.

User-mode access to the time base is read-only, and there is no explicitly privileged read access to the time base.

The **mftb** instruction reads from TBL and TBU. Writing the time base is accomplished by moving the contents of a GPR to a pair of SPRs, which are also called TBL and TBU, using **mtspr**.

Table 2-4 shows the mnemonics and names of the TBRs.

Table 2-4. Time Base Registers

Mnemonic	Register Name	Access
TBL	Time Base Lower (Read-only)	Read-only
TBU	Time Base Upper (Read-only)	Read-only

2.3.5 Machine State Register (MSR)

The Machine State Register (MSR) controls processor core functions, such as the enabling or disabling of interrupts and address translation.

The MSR is written from a GPR using the **mtmsr** instruction. The contents of the MSR can be read into a GPR using the **mfmsr** instruction. MSR[EE] is set or cleared using the **wrtee** or **wrteei** instructions.
The MSR contents are automatically saved, altered, and restored by the interrupt-handling mechanism. See *Machine State Register (MSR)* on page 114.

2.3.6 Device Control Registers

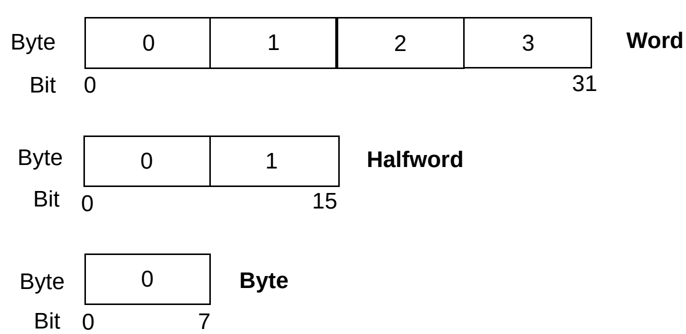
Device Control Registers (DCRs) are used to control various on-chip system functions such as the operation of on-chip buses, peripherals, and certain processor behaviors. The DCR access instructions are **mtdcr** (move-to-device control register) and **mfdcr** (move-from-device control register), which move data between GPRs and the DCRs.

Some DCRs are directly accessed, that is, they are accessed using their DCR numbers. Other DCRs are indirectly accessed. Such DCRs are accessed by writing an offset to a directly accessed DCR and then reading the data at the offset in another directly accessed DCR.

2.4 Data Types and Alignment

The data types consist of bytes (eight bits), halfwords (two bytes), words (four bytes), and strings (1 to 128 bytes). *Figure 2-10* shows the byte, halfword, and word data types and their bit and byte definitions for big endian representations of values. Note that PowerPC bit numbering is reversed from industry conventions; bit 0 represents the most significant bit of a value.

Figure 2-10. PPC405 Data Types



Data is represented in either twos-complement notation or in an unsigned integer format; data representation is independent of alignment issues.

The address of a data object is always the lowest address of any byte comprising the object.

All instructions are words, and are word-aligned (the lowest byte address is divisible by 4).

2.4.1 Alignment for Storage Reference and Cache Control Instructions

The storage reference instructions (loads and stores; see *Table 2-14*) move data to and from storage. The data cache control instructions listed in *Table 2-23*, control the contents and operation of the data cache unit (DCU). Both types of instructions form an effective address (EA). The method of calculating the EA for the storage reference and cache control instructions is detailed in the description of those instructions. See *Instruction Set* on page 157 for more information.

Cache control instructions ignore the five least significant bits of the EA; no alignment restrictions exist in the DCU because of EAs. However, storage control attributes can cause alignment exceptions. When data address translation is disabled and a dcbz instruction references a storage region that is non cacheable, or for which write-through caching is the write strategy, an alignment exception is taken. Such exceptions result from the storage control attributes, not from EA alignment.

The alignment exception enables system software to emulate the write-through function. Alignment requirements for the storage reference instructions and the dcread instruction depend on the particular instruction. *Table 2-5*, summarizes the instructions that cause alignment exceptions.

The data targets of instructions are of types that depend upon the instruction. The load/store instructions have the following "natural" alignments:

- Load/store word instructions have word targets, word-aligned.
- Load/ store halfword instructions have halfword targets, halfword-aligned.
- Load/store byte instructions have byte targets, byte-aligned (that is, any alignment).

Misalignments are addresses that are not naturally aligned on data type boundaries. An address not divisible by four is misaligned with respect to word instructions. An address not divisible by two is misaligned with respect to halfword instructions. The PPC405 implementation handles misalignments within and across word boundaries, but there is a performance penalty because additional cycles are required.

2.4.2 Alignment and Endian Operation

The endian storage control attribute does not affect alignment behavior. In little endian storage regions, the alignment of data is treated as it is in big endian storage regions; no special alignment exceptions occur when accessing data in little endian storage regions. Note that the alignment exceptions that apply to big endian region accesses also apply to little endian storage region accesses.

2.4.3 Summary of Instructions Causing Alignment Exceptions

Table 2-5 summarizes the instructions that cause alignment exceptions and the conditions under which the alignment exceptions occur.

Table 2-5. Alignment Exception Summary

Instructions Causing Alignment Exceptions	Conditions
dcbz	EA in non cacheable or write-through storage
dcread, **lwarx**, **stwcx**.	EA not word-aligned

2.5 Byte Ordering

The following discussion describes the “endianness” of the PPC405 core, which, by default and in normal use is “big endian.” The PPC405 also contains “little endian” peripherals and supports the attachment of external little endian peripherals.

If scalars (individual data items and instructions) were indivisible, there would be no such concept as “byte ordering.” It is meaningless to consider the order of bits or groups of bits within the smallest addressable unit of storage because nothing can be observed about such order. Only when scalars, which the programmer and processor regard as indivisible quantities, can comprise more than one addressable unit of storage does the question of byte order arise.

For a machine in which the smallest addressable unit of storage is the 32-bit word, there is no question of the ordering of bytes within words. All transfers of individual scalars between registers and storage are of words, and the address of the byte containing the high-order eight bits of a scalar is no different from the address of a byte containing any other part of the scalar.

For the PowerPC Architecture, as for most computer architectures currently implemented, the smallest addressable unit of storage is the 8-bit byte. Other scalars are halfwords, words, or doublewords, which consist of groups of bytes. When a word-length scalar is moved from a register to storage, the scalar is stored in four consecutive byte addresses. It thus becomes meaningful to discuss the order of the byte addresses with respect to the value of the scalar: that is, which byte contains the highest-order eight bits of the scalar, which byte contains the next-highest-order eight bits, and so on.

Given a scalar that contains multiple bytes, the choice of byte ordering is essentially arbitrary. There are 4! = 24 ways to specify the ordering of four bytes within a word, but only two of these orderings are sensible:

- The ordering that assigns the lowest address to the highest-order (“leftmost”) eight bits of the scalar, the next sequential address to the next-highest-order eight bits, and so on.

  This ordering is called *big endian* because the “big end” (most significant end) of the scalar, considered as a binary number, comes first in storage.

- The ordering that assigns the lowest address to the lowest-order (“rightmost”) eight bits of the scalar, the next sequential address to the next-lowest-order eight bits, and so on.

  This ordering is called *little endian* because the “little end” (least significant end) of the scalar, considered as a binary number, comes first in storage.

2.5.1 Structure Mapping Examples

The following C language structure, s, contains an assortment of scalars and a character string. The comments show the value assumed to be in each structure element; these values show how the bytes comprising each structure element are mapped into storage.

```
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 each scalar aligned at its natural boundary. 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.

2.5.1.1 Big Endian Mapping

The big endian mapping of structure *s* follows. (The data is highlighted in the structure mappings. Addresses, in hexadecimal, are below the data stored at the address. The contents of each byte, as defined in structure *s*, is shown as a (hexadecimal) number or character (for the string elements).

11 0x00	**12** 0x01	**13** 0x02	**14** 0x03	0x04	0x05	0x06	0x07
21 0x08	**22** 0x09	**23** 0x0A	**24** 0x0B	**25** 0x0C	**26** 0x0D	**27** 0x0E	**28** 0x0F
31 0x10	**32** 0x11	**33** 0x12	**34** 0x13	**'A'** 0x14	**'B'** 0x15	**'C'** 0x16	**'D'** 0x17
'E' 0x18	**'F'** 0x19	**'G'** 0x1A	0x1B	**51** 0x1C	**52** 0x1D	0x1E	0x1F
61 0x20	**62** 0x21	**63** 0x22	**64** 0x23	0x24	0x25	0x26	0x27

2.5.1.2 Little Endian Mapping

Structure *s* is shown mapped little endian.

14 0x00	**13** 0x01	**12** 0x02	**11** 0x03	0x04	0x05	0x06	0x07
28 0x08	**27** 0x09	**26** 0x0A	**25** 0x0B	**24** 0x0C	**23** 0x0D	**22** 0x0E	**21** 0x0F
34 0x10	**33** 0x11	**32** 0x12	**31** 0x13	**'A'** 0x14	**'B'** 0x15	**'C'** 0x16	**'D'** 0x17
'E' 0x18	**'F'** 0x19	**'G'** 0x1A	0x1B	**52** 0x1C	**51** 0x1D	0x1E	0x1F
64 0x20	**63** 0x21	**62** 0x22	**61** 0x23	0x24	0x25	0x26	0x27

2.5.2 Support for Little Endian Byte Ordering

Except as noted, this book describes the processor as if it operated only in a big endian fashion. In fact, the PowerPC Embedded Environment also supports little endian operation.

The PowerPC little endian mode, defined in the PowerPC Architecture, is not implemented.

2.5.3 Endian (E) Storage Attribute

The endian (E) storage attribute supports direct connection of the PPC405 to little endian peripherals and to memory containing little endian instructions and data. For every storage reference (instruction fetch or load/store access), an E storage attribute is associated with the storage region of the reference. The E attribute specifies whether that region is organized as big endian (E = 0) or little endian (E = 1).

When address translation is enabled (MSR[IR] = 1 or MSR[DR] = 1), the E field in the corresponding TLB entry controls the endianness of a memory region. When address translation is disabled (MSR[IR] = 0 or MSR[DR] = 0), the SLER controls the endianness of a memory region.

Bytes in storage that are accessed as little endian are arranged in true little endian format. The PPC405 does not support the little endian mode defined in the PowerPC architecture and used in PPC401xx and PPC403xx processors. Furthermore, no address modification is performed when accessing storage regions programmed as little endian. Instead, the PPC405 reorders the bytes as they are transferred between the processor and memory.

The on-the-fly reversal of bytes in little endian storage regions is handled in one of two ways, depending on whether the storage access is an instruction fetch or a data access (load/store). The following sections describe byte reordering for the two kinds of storage accesses.

2.5.3.1 Fetching Instructions from Little Endian Storage Regions

Instructions are words (four bytes) that are aligned on word boundaries in memory. As such, instructions in a big endian memory region are arranged with the most significant byte (MSB) of the instruction word at the lowest address.

Consider the big endian mapping of instruction p at address 00, where, for example, *p* = add r7, r7, r4:

Table 2-6. Big Endian Mapping

MSB			**LSB**
0x00	0x01	0x02	0x03

On the other hand, in the little endian mapping instruction p is arranged with the least significant byte (LSB) of the instruction word at the lowest numbered address:

Table 2-7. Little Endian Mapping

LSB			**MSB**
0x00	0x01	0x02	0x03

When an instruction is fetched from memory, the instruction must be placed in the instruction queue in the proper order. The execution unit assumes that the MSB of an instruction word is at the lowest address. Therefore, when instructions are fetched from little endian storage regions, the four bytes of an instruction word are reversed before the instruction is decoded. In the PPC405, the byte reversal occurs between memory and the instruction cache unit (ICU). The ICU always stores instructions in big endian format, regardless of whether the memory region containing the instruction is programmed as big endian or little endian. Thus, the bytes are already in the proper order when an instruction is transferred from the ICU to the decode stage of the pipeline.

If a storage region is reprogrammed from one endian format to the other, the storage region must be reloaded with program and data structures in the appropriate endian format. If the endian format of instruction memory changes, the ICU must be made coherent with the updates. The ICU must be invalidated and the updated instruction memory using the new endian format must be fetched so that the proper byte ordering occurs before the new instructions are placed in the ICU.

2.5.3.2 Accessing Data in Little Endian Storage Regions

Unlike instruction fetches from little endian storage regions, data accesses from little endian storage regions are not byte-reversed between memory and the DCU. Data byte ordering, in memory, depends on the data type (byte, halfword, or word) of a specific data item. It is only when moving a data item of a specific type from or to a GPR that it becomes known what type of byte reversal is required. Therefore, byte reversal during load/store accesses is performed between the DCU and the GPR.

When accessing data in a little endian storage region:

- 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.

Note that this applies, regardless of data alignment.

The big endian and little endian mappings of the structure s, shown in “Structure Mapping Examples” on page 44, demonstrate how the size of an 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.

Note that 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 0x14 in both the big endian and little endian mappings.

In little endian storage regions, the alignment of data is treated as it is in big endian storage regions. Unlike PowerPC little endian mode, no special alignment exceptions occur when accessing data in little endian storage regions.

2.5.3.3 PowerPC Byte-Reverse Instructions

For big endian storage regions, normal load/store instructions move the more significant bytes of a register to and from the lower-numbered memory addresses. The load/store with byte-reverse instructions move the more significant bytes of the register to and from the higher numbered memory addresses.

As *Figure 2-11* through *Figure 2-14* illustrate, a normal store to a big endian storage region is the same as a byte-reverse store to a little endian storage region. Conversely, a normal store to a little endian storage region is the same as a byte-reverse store to a big endian storage region.

Figure 2-11 illustrates the contents of a GPR and memory (starting at address 00) after a normal load/store in a big endian storage region.

Figure 2-11. Normal Word Load or Store (Big Endian Storage Region)

MSB			**LSB**	
11	**12**	**13**	**14**	**GPR**
11 0x00	**12** 0x01	**13** 0x02	**14** 0x03	**Memory**

Note that the results are identical to the results of a load/store with byte-reverse in a little endian storage region, as illustrated in *Figure 2-12*.

Figure 2-12. Byte-Reverse Word Load or Store (Little Endian Storage Region)

MSB			**LSB**	
11	**12**	**13**	**14**	**GPR**
11 0x00	**12** 0x01	**13** 0x02	**14** 0x03	**Memory**

Figure 2-13 illustrates the contents of a GPR and memory (starting at address 00) after a load/store with byte-reverse in a big endian storage region.

Figure 2-13. Byte-Reverse Word Load or Store (Big Endian Storage Region)

MSB			**LSB**	
11	**12**	**13**	**14**	**GPR**
14 0x00	**13** 0x01	**12** 0x02	**11** 0x03	**Memory**

Note that the results are identical to the results of a normal load/store in a little endian storage region, as illustrated in *Figure 2-14*.

Figure 2-14. Normal Word Load or Store (Little Endian Storage Region)

MSB			LSB	
11	**12**	**13**	**14**	**GPR**
14 0x00	**13** 0x01	**12** 0x02	**11** 0x03	**Memory**

The E storage attribute augments the byte-reverse load/store instructions in two important ways:

- The load/store with byte-reverse instructions do not solve the problem of fetching instructions from a storage region in little endian format.
  Only the endian storage attribute mechanism supports the fetching of little endian program images.
- Typical compilers cannot make general use of the byte-reverse load/store instructions, so these instructions are ordinarily used only in device drivers written in hand-coded assembler.
  Compilers can, however, take full advantage of the endian storage attribute mechanism, enabling application programmers working in a high-level language, such as C, to compile programs and data structures into little endian format.

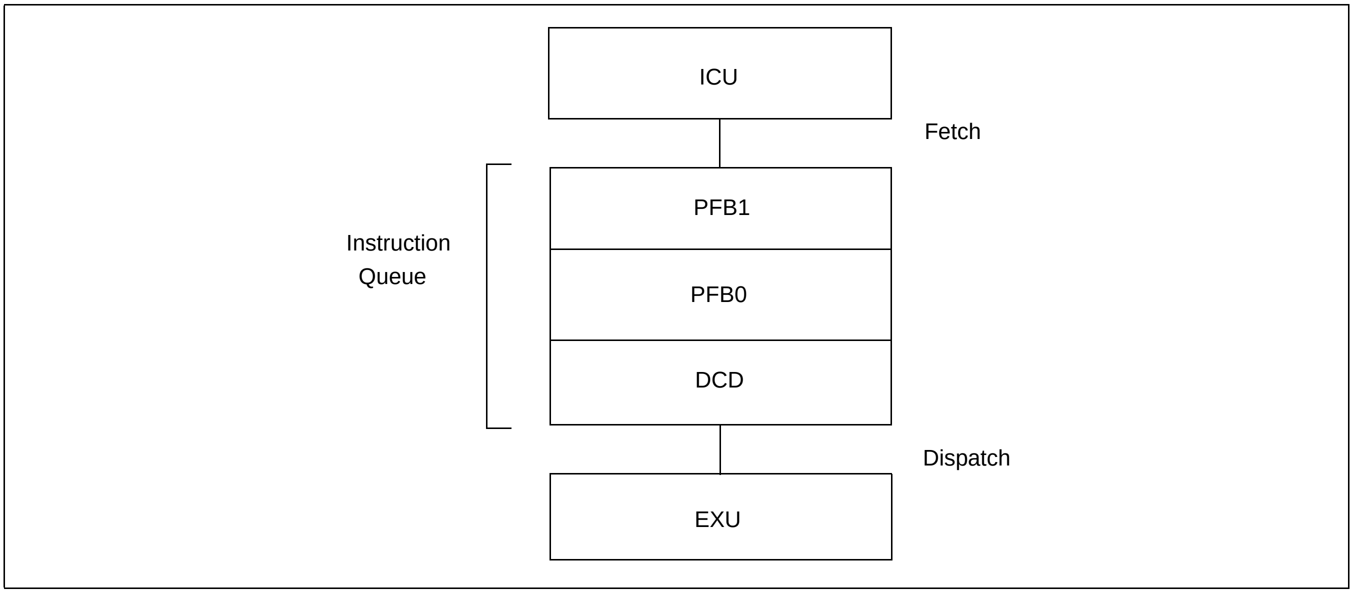
2.6 Instruction Processing

The instruction pipeline, illustrated in *Figure 2-15*, contains three queue locations: prefetch buffer 1 (PFB1), prefetch buffer 0 (PFB0), and decode (DCD). This queue implements a pipeline with the following functional stages: fetch, decode, execute, write-back and load write-back. Instructions are fetched from the instruction cache unit (ICU), placed in the instruction queue, and eventually dispatched to the execution unit (EXU).

Instructions are fetched from the ICU at the request of the EXU. Cacheable instructions are forwarded directly to the instruction queue and stored in the ICU cache array. Non cacheable instructions are also forwarded directly to the instruction queue, but are not stored in the ICU cache array. Fetched instructions drop to the empty queue location closest to the EXU. When there is room in the queue, instructions can be returned from the ICU two at a time. If the queue is empty and the ICU is returning two instructions, one instruction drops into DCD while the other drops into PFB0. PFB1 buffers instructions when the pipeline stalls.

Branch instructions are examined in DCD and PFB0 while all other instructions are decoded in DCD. All instructions must pass through DCD before entering the EXU. The EXU contains the execute, write-back and load write-back stages of the pipe. The results of most instructions are calculated during the execute stage and written to the GPR file during the write back stage. Load instructions write the GPR file during the load write-back stage.

Figure 2-15. PPC405 Instruction Pipeline



2.7 Branch Processing

The PPC405, which provides a variety of conditional and unconditional branching instructions, uses the branch prediction techniques described in *Branch Prediction* on page 52.

2.7.1 Unconditional Branch Target Addressing Options

The unconditional branches (**b, ba, bl, bla**) carry the displacement to the branch target address as a signed 26-bit value (the 24-bit LI field right-extended with 0b00). The displacement enables unconditional branches to cover an address range of ±32MB.

For the relative (AA = 0) forms (**b, bl**), the target address is the current instruction address (CIA, the address of the branch instruction) plus the signed displacement.

For the absolute (AA = 1) forms (**ba, bla**), the target address is 0 plus the signed displacement. If the sign bit (LI[0]) is 0, the displacement is the target address. If the sign bit is 1, the displacement is a negative value and wraps to the highest memory addresses. For example, if the displacement is 0x3FF FFFC (the 26-bit representation of –4), the target address is 0xFFFF FFFC (0 – 4B, or 4 bytes below the top of memory).

2.7.2 Conditional Branch Target Addressing Options

The conditional branches (**bc, bca, bcl, bcla**) carry the displacement to the branch target address as a signed 16-bit value (the 14-bit BD field right-extended with 0b00). The displacement enables conditional branches to cover an address range of ±32KB.

For the relative (AA = 0) forms (bc, bcl), the target address is the CIA plus the signed displacement.

For the absolute (AA = 1) forms (bca, bcla), the target address is 0 plus the signed displacement. If the sign bit (BD[0]) is 0, the displacement is the target address. If the sign bit is 1, the displacement is negative and wraps to the highest memory addresses. For example, if the displacement is 0xFFFC (the 16-bit representation of –4), the target address is 0xFFFF FFFC (0 – 4B, or 4 bytes from the top of memory).

2.7.3 Conditional Branch Condition Register Testing

Conditional branch instructions can test a CR bit. The value of the BI field specifies the bit to be tested (bit 0–31). The BO field controls whether the CR bit is tested, as described in the following section.

2.7.4 BO Field on Conditional Branches

The BO field of the conditional branch instruction specifies the conditions used to control branching, and specifies how the branch affects the CTR.

Conditional branch instructions can test one bit in the CR. This option is selected when BO[0] = 0; if BO[0] = 1, the CR does not participate in the branch condition test. If this option is selected, the condition is satisfied (branch can occur) if CR[BI] = BO[1].

Conditional branch instructions can decrement the CTR by one, and after the decrement, test the CTR value. This option is selected when BO[2] = 0. If this option is selected, BO[3] specifies the condition that must be satisfied to allow a branch to be taken. If BO[3] = 0, CTR ≠ 0 is required for a branch to occur. If BO[3] = 1, CTR = 0 is required for a branch to occur.

If BO[2] = 1, the contents of the CTR are left unchanged, and the CTR does not participate in the branch condition test.

Table 2-8 summarizes the usage of the bits of the BO field. BO[4] is further discussed in "*Branch Prediction* on page 52.

Table 2-8. Bits of the BO Field

BO Bit	Description
BO[0]	CR Test Control 0 Test CR bit specified by BI field for value specified by BO[1] 1 Do not test CR
BO[1]	CR Test Value 0 Test for CR[BI] = 0. 1 Test for CR[BI] = 1.
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 CTR, do not 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.

Table 2-9 lists specific BO field contents, and the resulting actions; z represents a mandatory value of 0, and y is a branch prediction option discussed in *Branch Prediction* on page 52.

Table 2-9. Conditional Branch BO Field

BO Value	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and CR[BI]=0.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and CR[BI] = 0.
001*zy*	Branch if CR[BI] = 0.
0100*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and CR[BI] = 1.
0101*y*	Decrement the CTR, then branch if the decremented CTR=0 and CR[BI] = 1.
011*zy*	Branch if CR[BI] = 1.
1*z*00*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0.
1*z*01*y*	Decrement the CTR, then branch if the decremented CTR = 0.
1*z*1*zz*	Branch always.

2.7.5 Branch Prediction

Conditional branches present a problem to the instruction fetcher. A branch might be taken. The branch EXU attempts to predict whether or not a branch is taken before all information necessary to determine the branch direction is available. This decision is called a *branch prediction*. The fetcher can then prefetch instructions starting at the predicted branch target address. If the prediction is correct, time is saved because the branched-to instruction is available in the instruction queue. Otherwise, the instruction pipeline stalls while the correct instruction is fetched into the instruction queue. To be effective, branch prediction must be correct most of the time.

The PowerPC Architecture enables software to reverse the default branch prediction, which is defined as follows:

Predict that the branch is to be taken if $((BO[0] \wedge BO[2]) \vee s) = 1$

where *s* is the sign bit of the displacement for conditional branch (**bc**) instructions, and 0 for **bclr** and **bcctr** instructions.

$(BO[0] \wedge BO[2]) = 1$ only when the conditional branch tests nothing (the “branch always” condition). Obviously, the branch should be predicted taken for this case.

If the branch tests anything, $(BO[0] \wedge BO[2]) = 0$, and *s* entirely controls the prediction. The default prediction for this case was decided by considering the relative form of **bc**, which is commonly used at the end of loops to control the number of times that a loop is executed. The branch is taken every time the loop is executed except the last, so it is best if the branch is predicted taken. The branch target is the beginning of the loop, so the branch displacement is negative and *s* = 1.

If branch displacements are positive (*s* = 0), the branch is predicted not taken. If the branch instruction is any form of **bclr** or **bcctr** except the “branch always” forms, then *s* = 0, and the branch is predicted not taken.

There is a peculiar consequence of this prediction algorithm for the absolute forms of **bc** (**bca** and **bcla**). As described in *Unconditional Branch Target Addressing Options* on page 50, if the algebraic sign of the displacement is negative (*s* = 1), the branch target address is in high memory. If the algebraic sign of the displacement is positive (s = 0), the branch target address is in low memory. Because these are absolute-addressing forms, there is no reason to treat high and low memory differently. Nevertheless, for the high memory case the default prediction is taken, and for the low memory case the default prediction is not taken.

BO[4] is the *prediction reversal bit*. If BO[4] = 0, the default prediction is applied. If BO[4] = 1, the reverse of the standard prediction is applied. For the cases in Table 2-14 where BO[4] = *y*, software can reverse the default prediction. This should only be done when the default prediction is likely to be wrong. Note that for the “branch always” condition, reversal of the default prediction is not allowed.

The PowerPC Architecture requires assemblers to provide a way to conveniently control branch prediction. For any conditional branch mnemonic, a suffix may be added to the mnemonic to control prediction, as follows:

+	Predict branch to be taken
–	Predict branch to be not taken

For example, **bcctr+** causes BO[4] to be set appropriately to force the branch to be predicted taken.

2.8 Speculative Accesses

The PowerPC Architecture permits implementations to perform speculative accesses to memory, either for instruction fetching, or for data loads. A speculative access is defined as any access which is not required by a sequential execution model.

For example, prefetching instructions beyond an undetermined conditional branch is a speculative fetch; if the branch is not in the predicted direction, the program, as executed, never needs the instructions from the predicted path.

Sometimes speculative accesses are inappropriate. For example, attempting to fetch instructions from addresses that cannot contain instructions can cause problems.To protect against errant accesses to “sensitive” memory or I/O devices, the PowerPC Architecture provides the G (guarded) storage attribute, which can be used to specify memory pages from which speculative accesses are prohibited. (Actually, speculative accesses to guarded storage are allowed in certain limited circumstances; if an instruction in a cache block will be executed, the rest of the cache block can be speculatively accessed.)

2.8.1 Speculative Accesses in the PPC405

The PPC405 does not perform speculative loads.

Two methods control speculative instruction fetching. If instruction address translation is enabled (MSR[IR] = 1), the G (guarded) field in the translation lookaside buffer (TLB) entries controls speculative accesses.

If instruction address translation is disabled (MSR[IR] = 0), the Storage Guarded Register (SGR) controls speculative accesses for regions of memory. When a region is guarded (speculative fetching is disallowed), instruction prefetching is disabled for that region. A fetch request must be completely resolved (no longer speculative) before it is issued. There is a considerable performance penalty for fetching from guarded storage, so guarding should be used only when required.

Note that, following any reset, the PPC405 operates with all of storage guarded.

Note that when address translation is enabled, attempts to fetch from guarded storage result in instruction storage exceptions. Guarded memory is in most often needed with peripheral status registers that are cleared automatically after being read, because an unintended access resulting from a speculative fetch would cause the loss of status information. Because the MMU provides 64 pages with a wide range of page sizes as small as 1KB, fetching instructions from guarded storage should be unnecessary.

2.8.1.1 Prefetch Distance Down an Unresolved Branch Path

The fetcher will speculatively access up to 19 instructions down a predicted branch path, whether taken or sequential, regardless of cachability.

2.8.1.2 Prefetch of Branches to the CTR and Branches to the LR

When the instruction fetcher predicts that a bctr or blr instruction will be taken, the fetcher does not attempt to fetch an instruction from the target address in the CTR or LR if an executing instruction updates the register ahead of the branch. (See *Instruction Processing* on page 49 for a description of the instruction pipeline). The fetcher recognizes that the CTR or LR contains data left from an earlier use and that such data is probably not valid.

In such cases, the fetcher does not fetch the instruction at the target address until the instruction that is updating the CTR or LR completes. Only then are the "correct" CTR or LR contents known. This prevents the fetcher from speculatively accessing a completely "random" address. After the CTR or LR contents are known to be correct, the fetcher accesses no more than five instructions down the sequential or taken path of an unresolved branch, or at the address contained in the CTR or LR.

2.8.2 Preventing Inappropriate Speculative Accesses

A memory-mapped I/O peripheral, such as a serial port having a status register that is automatically reset when read provides a simple example of storage that should not be speculatively accessed. If code is in memory at an address adjacent to the peripheral (for example, code goes from 0x0000 0000 to 0x0000 0FFF, and the peripheral is at 0x0000 1000), prefetching past the end of the code will read the peripheral.

Guarding storage also prevents prefetching past the end of memory. If the highest memory address is left unguarded, the fetcher could attempt to fetch past the last valid address, potentially causing machine checks on the fetches from invalid addresses. While the machine checks do not actually cause an exception until the processor attempts to execute an instruction at an invalid address, some systems could suffer from the attempt to access such an invalid address. For example, an external memory controller might log an error.

System designers can avoid problems from speculative fetching without using the guarded storage attributes. The rest of this section describes ways to prevent speculative instruction fetches to sensitive addresses in unguarded memory regions.

2.8.2.1 Fetching Past an Interrupt-Causing or Interrupt-Returning Instruction

Suppose a bctr or blr instruction closely follows an interrupt-causing or interrupt-returning instruction (**sc**, **rfi**, or **rfci**). The fetcher does not prevent speculatively fetching past one of these instructions. In other words, the fetcher does not treat the interrupt-causing and interrupt-returning instructions specially when deciding whether to predict down a branch path. Instructions after an **rfi**, for example, are considered to be on the determined branch path.

To understand the implications of this situation, consider the code sequence:

```
   handler: aaa
            bbb
            rfi
subroutine: bctr
```

When executing the interrupt handler, the fetcher does not recognize the **rfi** as a break in the program flow, and speculatively fetches the target of the **bctr**, which is really the first instruction of a subroutine that has not been called. Therefore, the CTR might contain an invalid pointer.

To protect against such a prefetch, the software must insert an unconditional branch hang (b $) just after the **rfi**. This prevents the hardware from prefetching the invalid target address used by **bctr**.

Consider also the above code sequence, with the **rfi** instruction replaced by an sc instruction used to initialize the CTR with the appropriate value for the bctr to branch to, upon return from the system call. The **sc** handler returns to the instruction following the **sc**, which can’t be a branch hang. Instead, software could put a **mtctr** just before the **sc** to load a non-sensitive address into the CTR. This address will be used as the prediction address before the **sc** executes. An alternative would be to put a **mfctr** or mtctr between the **sc** and the **bctr**; the **mtctr** prevents the fetcher from speculatively accessing the address contained in the CTR before initialization.

2.8.2.2 Fetching Past tw or twi Instructions

The interrupt-causing instructions, **tw** and **twi**, do not require the special handling described in *Fetching Past an Interrupt-Causing or Interrupt-Returning Instruction* on page 54. These instructions are typically used by debuggers, which implement software breakpoints by substituting a trap instruction for the instruction originally at the breakpoint address. In a code sequence mtlr followed by **blr** (or **mtctr** followed by **bctr**), replacement of **mtlr**/**mtctr** by **tw** or **twi** leaves the LR/CTR uninitialized. It would be inappropriate to fetch from the **blr**/**bctr** target address. This situation is common, and the fetcher is designed to prevent the problem.

2.8.2.3 Fetching Past an Unconditional Branch

When an unconditional branch is in DCD in the instruction queue, the fetcher recognizes that the sequential instructions following the branch are unnecessary. These sequential addresses are not accessed. Addresses at the branch target are accessed instead.

Therefore, placing an unconditional branch just before the start of a sensitive address space (for example, at the “end” of a memory area that borders an I/O device) guarantees that addresses in the sensitive area will not be speculatively fetched.

2.8.2.4 Suggested Locations of Memory-Mapped Hardware

The preferred method of protecting memory-mapped hardware from inadvertent access is to use address translation, with hardware isolated to guarded pages (the G storage attribute in the associated TLB entry is set to 1.) The pages can be as small as 1KB. Code should never be stored in such pages.

If address translation is disabled, the preferred protection method is to isolate memory-mapped hardware into regions guarded using the SGR. Code should never be stored in such regions. The disadvantage of this method, compared to the preferred method, is that each region guarded by the SGR consumes 128MB of the address space.

Table 2-10 shows two address regions of the PPC405. Suppose a system designer can map all I/O devices and all ROM and SRAM devices into any location in either region. The choices made by the designer can prevent speculative accesses to the memory-mapped I/O devices.

Table 2-10. Example Memory Mapping

0x7800 0000 – 0x7FFF FFFF (SGR bit 15)	128MB Region 2
0x7000 0000 – 0x77FF FFFF (SGR bit 14)	128MB Region 1

A simple way to avoid the problem of speculative reads to peripherals is to map all storage containing code into Region 2, and all I/O devices into Region 1. Thus, accesses to Region 2 would only be for code and program data. Speculative fetches occurring in Region 2 would never access addresses in Region 1. Note that this hardware organization eliminates the need to use of the G storage attribute to protect Region 1. However, Region 1 could be set as guarded with no performance penalty, because there is no code to execute or variable data to access in Region 1.

The use of these regions could be reversed (code in Region 1 and I/O devices in Region 2), if Region 2 is set as guarded. Prefetching from the highest addresses of Region 1 could cause an attempt to speculatively access the bottom of Region 2, but guarding prevents this from occurring. The performance penalty is slight, under the assumption that code infrequently executes the instructions in the highest addresses of Region 1.

2.8.3 Summary

Software should take the following actions to prevent speculative accesses to sensitive data areas, if the sensitive data areas are not in guarded storage:

- Protect against accesses to “random” values in the LR or CTR on **blr** or **bctr** branches following **rfi**, **rfci**, or **sc** instructions by putting appropriate instructions before or after the **rfi**, **rfci**, or **sc** instruction. See *Fetching Past an Interrupt-Causing or Interrupt-Returning Instruction* on page 54.
- Protect against “running past” the end of memory into a bordering I/O device by putting an unconditional branch at the end of the memory area. See *Fetching Past an Unconditional Branch* on page 55.
- Recognize that a maximum of 19 words can be prefetched past an unresolved conditional branch, either down the target path or the sequential path. See *Prefetch Distance Down an Unresolved Branch Path* on page 54.

Of course, software should not code branches with known unsafe targets (either relative to the instruction counter, or to addresses contained in the LR or CTR), on the assumption that the targets are “protected” by code guaranteeing that the unsafe direction is not taken. The fetcher assumes that if a branch is predicted to be taken, it is safe to fetch down the target path.

2.9 User and Supervisor Modes

In the PowerPC Book-E architecture defines two operating states or modes,” supervisor (privileged), and user (non privileged). The mode in which the processor is operating is controlled by MSR[PR]. When MSR[PR] is 0, the processor is in supervisor mode and can execute all instructions and access all registers, including privileged ones. When MSR[PR] is 1, the processor is in user mode and can only execute non privileged instructions and access non privileged registers. An attempt to execute a privileged instruction or to access a privileged register while in user mode causes a Privileged Instruction exception type program interrupt to occur.

Note that the name “PR” for the MSR field refers to a historical alternative name for user mode, whic is “problem state.” Hence the value 1 in the field indicates “problem state,” and not “privileged” as one might expect. After a reset, MSR[PR] = 0.

2.9.1 MSR Bits and Exception Handling

The current value of MSR[PR] is saved, along with all other MSR bits, in the SRR1 (for non-critical interrupts) or SRR3 (for critical interrupts) upon any interrupt, and MSR[PR] is set to 0. Therefore, all exception handlers operate in privileged mode.

Attempting to execute a privileged instruction while in user mode causes a privileged violation program exception (see *Program Interrupt* on page 123). The PPC405 does not execute the instruction, and the program counter is loaded with EVPR[0:15] **||** 0x0700, the address of an exception processing routine.

The PRR field of the Exception Syndrome Register (ESR) is set when an interrupt was caused by a privileged instruction program exception. Software is not required to clear ESR[PPR].

2.9.2 Privileged Instructions

The instructions listed in *Table 2-11* are privileged and cannot be executed in user mode.

Table 2-11. Privileged Instructions

dcbi	
dccci	
dcread	
iccci	
icread	
mfdcr	
mfmsr	
mfspr	For all SPRs except CTR, LR, SPRG4–SPRG7, and XER. See “Privileged SPRs” on page 57
mtdcr	
mtmsr	
mtspr	For all SPRs except CTR, LR, XER. See “Privileged SPRs” on page 57
rfci	
rfi	
tlbia	
tlbre	
tlbsx	
tlbsync	
tlbwe	
wrtee	
wrteei	

2.9.3 Privileged SPRs

Most SPRs are privileged. The only defined non privileged SPRs are the LR, CTR, XER, USPRG0, and SPRG4–7 (read access only), TBU (read access only), and TBL (read access only). These registers are read using the **mftb** instruction, rather than the **mfspr** instruction. TBL and TBU are written (with different addresses) using **mtspr**, which is privileged for these registers. Except for moves to and from non privileged SPRs, attempts to execute **mfspr** and **mtspr** instructions while in user mode result in privileged violation program exceptions.

In a **mfspr** or **mtspr** instruction, the 10-bit SPRN field specifies the SPR number of the source or destination SPR. The SPRN field contains two five-bit subfields, SPRN0:4 and SPRN5:9. The assembler handles the unusual register number encoding to generate the SPRF field. In the machine code for the **mfspr** and **mtspr** instructions, the SPRN subfields are reversed (ending up as SPRF5:9 and SPRF0:4) for compatibility with the POWER Architecture.

In the PowerPC Architecture, SPR numbers having a 1 in the most-significant bit of the SPRF field are privileged.

The following example illustrates how SPR numbers appear in assembler language coding and in machine coding of the **mfspr** and **mtspr** instructions.

In assembler language coding, SRR0 is SPR 26. Note that the assembler handles the unusual register number encoding to generate the SPRF field.

```
mfspr r5,26
```

When the SPR number is considered as a binary number (0b0000011010), the most-significant bit is 0. However, the machine code for the instruction reverses the subfields, resulting in the following SPRF field: 0b1101000000. The most-significant bit is 1; SRR0 is privileged.

When an SPR number is considered as a hexadecimal number, the second digit of the three-digit hexadecimal number indicates whether an SPR is privileged. If the second digit is odd (1, 3, 5, 7, 9, B, D, F), the SPR is privileged.

For example, the SPR number of SRR0 is 26 (0x01A). The second hexadecimal digit is odd; SRR0 is privileged. In contrast, the LR is SPR 8 (0x008); the second hexadecimal digit is not odd; the LR is non-privileged.

2.9.4 Privileged DCRs

The **mtdcr** and **mfdcr** instructions themselves are privileged, in all cases. All DCRs are privileged.

2.10 Synchronization

The PPC405 supports the synchronization operations of the PowerPC Book-E architecture. There are three kinds of synchronization defined by the architecture, each of which is described in the following sections.

2.10.1 Context Synchronization

The context of a program is the environment in which the program executes. For example, the mode (user or supervisor) is part of the context, as are the address translation space and storage attributes of the memory pages being accessed by the program. Context is controlled by the contents of certain registers and other resources, such as the MSR and the translation lookaside buffer (TLB).

Under certain circumstances, it is necessary for the hardware or software to force the synchronization of a program’s context. Context synchronizing operations include all interrupts except Machine Check, as well as the **isync**, **sc**, **rfi**, and **rfci** instructions. Context synchronizing operations satisfy the following requirements:

1. The operation is not initiated until all instructions preceding the operation have completed to the point at which they have reported any and all exceptions that they will cause.
2. All instructions *preceding* the operation must complete in the context in which they were initiated. That is, they must not be affected by any context changes caused by the context synchronizing operation, or any instructions *after* the context synchronizing operation.
3. If the operation is the **sc** instruction (which causes a System Call interrupt) or is itself an interrupt, then the operation is not initiated until no higher priority interrupt is pending (see *Interrupt Handling* on page 109).
4. All instructions that *follow* the operation must be re-fetched and executed in the context that is established by the completion of the context synchronizing operation and all of the instructions which *preceded* it.

Note that context synchronizing operations do not force the completion of storage accesses, nor do they enforce any ordering amongst accesses before and/or after the context synchronizing operation. If such behavior is required, then a storage synchronizing instruction must be used (see *Storage Ordering and Synchronization* on page 60).

Also note that architecturally Machine Check interrupts are not context synchronizing. Therefore, an instruction that *precedes* a context synchronizing operation can cause a Machine Check interrupt *after* the context synchronizing operation occurs and additional instructions have completed. For the PPC405, this can only occur with Data Machine Check exceptions, and not Instruction Machine Check exceptions.

The following scenarios use pseudocode examples to illustrate these limitations of context synchronization. Subsequent text explains how software can further guarantee “storage ordering.”

1. Consider the following instruction sequence:

```
STORE non cacheable to address XYZ
isync
XYZ instruction
```

In this sequence, the **isync** instruction does not guarantee that the XYZ instruction is fetched after the STORE has occurred to memory. There is no guarantee which XYZ instruction will execute; either the old version or the new (stored) version might.

2. Consider the following instruction sequence, which assumes that the PPC405 uses DCRs to provide bus region control:

```
STORE non cacheable to address XYZ
isync
mtdcr to change a bus region containing XYZ
```

In this sequence, there is no guarantee that the STORE will occur before the **mtdcr** changing the bus region control DCR. The STORE could fail because of a configuration error.

Consider an interrupt that changes privileged mode. An interrupt is a context synchronizing operation, because interrupts cause the MSR to be updated. The MSR is part of the processor context; the context synchronizing operation guarantees that all instructions that precede the interrupt complete using the preinterrupt value of MSR[PR], and that all instructions that follow the interrupt complete using the postinterrupt value.

Consider, on the other hand, some code that uses **mtmsr** to change the value of MSR[PR], which changes the privileged mode. In this case, the MSR is changed, changing the context. It is possible, for example, that prefetched privileged instructions expect to execute after the **mtmsr** has changed the operating mode from privileged mode to user mode. To prevent privileged instruction program exceptions, the code must execute a context synchronization operation, such as **isync**, immediately after the **mtmsr** instruction to prevent further instruction execution until the **mtmsr** completes.

eieio or **sync** can ensure that the contents of memory and DCRs are synchronized in the instruction stream. These instructions guarantee storage ordering because all memory accesses that precede **eieio** or **sync** are completed before subsequent memory accesses. Neither **eieio** nor **sync** guarantee that instruction prefetching is delayed until the eieio or sync completes. The instructions do not cause the prefetch queues to be purged and instructions to be refetched. See “Storage Ordering and Synchronization” on page 60 for more information.

Instruction cache state is part of context. A context synchronization operation is required to guarantee instruction cache access ordering.

3. Consider the following instruction sequence, which is required for creating self-modifying code:

STORE	Change data cache contents
dcbst	Flush the new data cache contents to memory
sync	Guarantee that **dcbst** completes before subsequent instructions begin
icbi	Context changing operation; invalidates instruction cache contents.
isync	Context synchronizing operation; causes refetch using new instruction cache context text and new memory context, due to the previous **sync**.

If software wishes to ensure that all storage accesses are complete before executing a **mtdcr** to change a bus region (Example 2), the software must issue a **sync** after all storage accesses and before the **mtdcr**. Likewise, if the software is to ensure that all instruction fetches after the **mtdcr** use the new bank register contents, the software must issue an **isync**, after the **mtdcr** and before the first instruction that should be fetched in the new context.

isync guarantees that all subsequent instructions are fetched and executed using the context established by all previous instructions. **isync** is a context synchronizing operation; **isync** causes all subsequently prefetched instructions to be discarded and refetched.

The following example illustrates the use of **isync** with debug exceptions:

mtdbcr0	Enable an instruction address compare (IAC) event
isync	Wait for the new Debug Control Register 0 (DBCR0) context to be established
XYZ	This instruction is at the IAC address; an **isync** was necessary to guarantee that the IAC event occurs at the execution of this instruction

2.10.2 Execution Synchronization

Execution synchronization is a subset of context synchronization. An execution synchronizing operation satisfies the first two requirements of context synchronizing operations, but not the latter two. That is, execution synchronizing operations guarantee that preceding instructions execute in the “old” context, but do not guarantee that subsequent instructions operate in the “new” context.

There are three execution synchronizing operations: **eieio, mtmsr,** and **sync**. Note that all context synchronizing instructions are also implicitly execution synchronizing, since context synchronization is a superset of execution synchronization.

Because **mtmsr** is execution synchronizing, it guarantees that previous instructions complete using the old MSR value. (For example, using **mtmsr** to change the endian mode.) However, to guarantee that subsequent instructions use the new MSR value, we have to insert a context synchronization operation, such as **isync**.

Note that PowerPC Book-E imposes additional requirements on updates to MSR[EE] (the external interrupt enable bit). Specifically, if a **mtmsr**, **wrtee**, or **wrteei** instruction sets MSR[EE] = 1, and an External Input, Decrementer, or Fixed Interval Timer exception is pending, the interrupt must be taken before the instruction that follows the MSR[EE]-updating is executed. In this sense, these MSR[EE]-updating instructions can be thought of as being context synchronizing with respect to the MSR[EE] bit, in that it guarantees that subsequent instructions execute (or are prevented from executing and an interrupt taken) according to the new context of MSR[EE].

Finally, while **sync** and **eieio** are execution synchronizing, they are also more restrictive in their requirement of memory ordering. Stating that an operation is execution synchronizing does not imply storage ordering. This is an additional specific requirement of **sync** and **eieio**.

2.10.3 Storage Ordering and Synchronization

Storage synchronization enforces ordering between storage access instructions executed by the PPC405. The **sync** instruction guarantees that all previous storage references complete with respect to the PPC405 before the **sync** instruction completes (therefore, before any subsequent instructions begin to execute). The **sync** instruction is execution synchronizing. Consider the following use of **sync**:

Consider the following use of **sync**:

stw	Store to peripheral
sync	Wait for store to actually complete
mtdcr	Reconfigure device

The **eieio** instruction guarantees the order of storage accesses. All storage accesses that precede **eieio** complete before any storage accesses that follow the instruction, as in the following example:

stb X	Store to peripheral, address X; this resets a status bit in the device
eieio	Guarantee **stb** X completes before next instruction
lbz Y	Load from peripheral, address Y; this is the status register updated by **stb** X.

eieio was necessary, because the read and write addresses are different, but affect each other

The PPC405 implements both **sync** and **eieio** identically, in the manner described above for **sync**. In the PowerPC Architecture, **sync** can function across all processors in a multiprocessor environment; **eieio** functions only within its executing processor. The PPC405 does not provide hardware support for multiprocessor memory coherency, so **sync** does not guarantee memory ordering across multiple processors.

2.11 Implemented Instruction Set Summary

This section provides an overview of the various types and categories of instructions implemented within the PPC405. In addition, *Instruction Set* on page 157 provides a complete alphabetical listing of every implemented instruction.

Appendix A Instruction Summary on page 357 alphabetically lists each instruction and extended mnemonic and provides a short-form description. *Appendix B Instructions by Category* on page 395 provides short-form descriptions of instructions, grouped by the instruction categories listed in *Table 2-12*.

Table 2-12 summarizes the PPC405 instruction set functions by categories. Instructions within each category are described in subsequent sections.

Table 2-12. PPC405 Instruction Set Summary

Category	Subcategory	Instruction Types
Integer	Integer Storage Access	load, store
	Integer Arithmetic	add, subtract, negate, multiply, multiply-accumulate, multiply halfword, divide
	Integer Logical	and, andc, or, orc, xor, nand, nor, xnor, extend sign, count leading zeros
	Integer Compare	compare, compare logical, compare immediate
	Integer Rotate	rotate and insert, rotate and mask
	Integer Shift	shift left, shift right, shift right algebraic
Branch		branch, branch conditional, branch to LR, branch to CTR
Processor Control	Condition Register Logical	crand, crandc, cror, crorc, crnand, crnor, crxor, crxnor, move CR field
	Register Management	move to/from SPR, move to/from DCR, move to/from CR
	System Linkage	system call, return from interrupt, return from critical interrupt, return from machine check interrupt
	Trap	trap
	Interrupt Control	move to/from MSR, return from interrupt, return from critical interrupt, return from machine check interrupt, write to external interrupt enable bit
	Processor Synchronization	synchronize
Storage Control	Cache Management	data allocate, data invalidate, data touch, data zero, data flush, data store, data read, instruction invalidate, instruction touch
	TLB Management	read, write, search, synchronize

2.11.1 Instructions Specific to the PowerPC Embedded Environment

To support functions required in embedded real-time applications, the PowerPC processors define instructions that are not defined in the PowerPC Architecture.

Table 2-13 lists the instructions specific to PowerPC embedded processors. Programs using these instructions are not portable to PowerPC implementations that are not part of the PowerPC 400 family of embedded processors.

In the table, the syntax [s] indicates that the instruction has a signed form. The syntax [u] indicates that the instruction has an unsigned form. The syntax [.] indicates that the instruction has a “record” form that updates CR[CR0], and a “non-record” form.

Table 2-13. Implementation-specific Instructions

dccci **dcread** **iccci** **icread**	**macchw**[**s**][**u**] **machhw**[**s**][**u**] **maclhw**[**s**][**u**] **nmacchw**[**s**] **nmachhw**[**s**] **nmaclhw**[**s**]	**mulchw**[**u**] **mulhhw**[**u**] **mullhw**[**u**]	**mfdcr** **mtdcr** **rfci** **tlbre** **tlbsx**[**.**] **tlbwe** **wrtee** **wrteei**

2.11.2 Storage Reference Instructions

Table 2-14 lists the PPC405 storage reference instructions. Load/store instructions transfer data between memory and the GPRs. These instructions operate on bytes, halfwords, and words. Storage reference instructions also support loading or storing multiple registers, character strings, and bytereversed data.

In the table, the syntax [u] indicates that an instruction has an “update” form that updates the RA addressing register with the calculated address, and a “non-update” form. The syntax [x] indicates that an instruction has an “indexed” form, which forms the address by adding the contents of the RA and RB GPRs and a “base + displacement” form, in which the address is formed by adding a 16-bit signed immediate value (included as part of the instruction word) to the contents of RA GPR.

Table 2-14. Storage Reference Instructions

Loads				Stores			
Byte	Halfword	Word	Multiple/String	Byte	Halfword	Word	Multiple/String
lbz[**u**][**x**]	**lha**[**u**][**x**] **lhbrx** **lhz**[**u**][**x**]	**lwarx** **lwbrx** **lwz**[**u**][**x**]	**lmw** **lswi** **lswx**	**stb**[**u**][**x**]	**sth**[**u**][**x**] **sthbrx**	**stw**[**u**][**x**] **stwbrx** **stwcx.**	**stmw** **stswi** **stswx**

2.11.3 Arithmetic Instructions

Arithmetic operations are performed on integer operands stored in GPRs. Instructions that perform operations on two operands are defined in a three-operand format; an operation is performed on the operands, which are stored in two GPRs. The result is placed in a third, operand, which is stored in a GPR. Instructions that perform operations on one operand are defined using a two-operand format; the operation is performed on the operand in a GPR and the result is placed in another GPR. Several instructions also have immediate formats in which an operand is contained in a field in the instruction word.

Most arithmetic instructions have versions that can update CR[CR0] and XER[SO, OV], based on the result of the instruction. Some arithmetic instructions also update XER[CA] implicitly. See *Condition Register (CR)* on page 39 and *Fixed Point Exception Register (XER)* on page 37 for more information.

Table 2-15 lists the PPC405 arithmetic instructions. In the table, the syntax [o] indicates that an instruction has an “o” form that updates XER[SO,OV], and a “non-o” form. The syntax [.] indicates that the instruction has a “record” form that updates CR[CR0], and a “non-record” form.

Table 2-15. Arithmetic Instructions

Add	Subtract	Multiply	Divide	Negate
add[o][.] **addc**[o][.] **adde**[o][.] **addi** **addic**[.] **addis** **addme**[o][.] **addze**[o][.]	**subf**[o][.] **subfc**[o][.] **subfe**[o][.] **subfic** **subfme**[o][.] **subfze**[o][.]	**mulhw**[.] **mulhwu**[.] **mulli** **mullw**[o][.]	**divw**[o][.] **divwu**[o][.]	**neg**[o][.]

Table 2-16 lists additional arithmetic instructions for multiply-accumulate and multiply halfword operations. In the table, the syntax [o] indicates that an instruction has an “o” form that updates XER[SO,OV], and a “non-o” form. The syntax [.] indicates that the instruction has a “record” form that updates CR[CR0], and a “non-record” form.

Table 2-16. Multiply-Accumulate and Multiply Halfword Instructions

Multiply-Accumulate	Negative-Multiply- Accumulate	Multiply Halfword
macchw[o][.] **macchws**[o][.] **macchwsu**[o][.] **macchwu**[o][.] **machhw**[o][.] **machhws**[o][.] **machhwsu**[o][.] **machhwu**[o][.] **maclhw**[o][.] **maclhws**[o][.] **maclhwsu**[o][.] **maclhwu**[o][.]	**nmacchw**[o][.] **nmacchws**[o][.] **nmachhw**[o][.] **nmachhws**[o][.] **nmaclhw**[o][.] **nmaclhws**[o][.]	**mulchw**[.] **mulchwu**[.] **mulhhw**[.] **mulhhwu**[.] **mullhw**[.] **mullhwu**[.]

2.11.4 Logical Instructions

Table 2-17 lists the PPC405 logical instructions. In the table, the syntax [.] indicates that the instruction has a "record" form that updates CR[CR0], and a "non-record" form.

Table 2-17. Logical Instructions

And	And with Complement	Nand	Or	Or with Complement	Nor	Xor	Equivalence	Extend Sign	Count Leading Zeros
and[.] **andi.** **andis.**	**andc**[.]	**nand**[.]	**or**[.] **ori** **oris**	**orc**[.]	**nor**[.]	**xor**[.] **xori** **xoris**	**eqv**[.]	**extsb**[.] **extsh**[.]	**cntlzw**[.]

2.11.5 Compare Instructions

These instructions perform arithmetic or logical comparisons between two operands and update the CR with the result of the comparison.

Table 2-18 lists the PPC405 compare instructions

.

Table 2-18. Compare Instructions

Arithmetic	Logical
cmp **cmpi**	**cmpl** **cmpli**

2.11.6 Branch Instructions

These instructions unconditionally or conditionally branch to an address. Conditional branch instructions can test condition codes set by a previous instruction and branch accordingly. Conditional branch instructions can also decrement and test the CTR as part of branch determination, and can save the return address in the LR.The target address for a branch can be a displacement from the current instruction address (a relative address), an absolute address, or contained in the CTR or LR.

See *Branch Processing* on page 50 for more information on branch operations.

Table 2-19 lists the PPC405 branch instructions. In the table, the syntax [l] indicates that the instruction has a "link update" form that updates LR with the address of the instruction after the branch, and a "non-link update" form. The syntax [a] indicates that the instruction has an "absolute address" form, in which the target address is formed directly using the immediate field specified as part of the instruction, and a "relative" form, in which the target address is formed by adding the immediate field to the address of the branch instruction).

Table 2-19. Branch Instructions

Branch
b[**l**][**a**] **bc**[**l**][**a**] **bcctr**[**l**] **bclr**[**l**]

2.11.6.1 CR Logical Instructions

These instructions perform logical operations on a specified pair of bits in the CR, placing the result in another specified bit. These instructions can logically combine the results of several comparisons without incurring the overhead of conditional branch instructions. Software performance can significantly improve if multiple conditions are tested at once as part of a branch decision.

Table 2-20 lists the PPC405 condition register logical instructions.

Table 2-20. CR Logical Instructions

crand **crandc** **creqv** **crnand**	**crnor** **cror** **crorc** **crxor** **mcrf**

2.11.6.2 Rotate Instructions

These instructions rotate operands stored in the GPRs. Rotate instructions can also mask rotated operands.

Table 2-21 lists the PPC405 rotate instructions. In the table, the syntax [.] indicates that the instruction has a “record” form that updates CR[CR0], and a “non-record” form.

Table 2-21. Rotate Instructions

Rotate and Insert	Rotate and Mask
rlwimi[.]	**rlwinm**[.] **rlwnm**[.]

2.11.6.3 Shift Instructions

These instructions shift operands stored in the GPRs.

Table 2-22 lists the PPC405 shift instructions. Shift right algebraic instructions implicitly update XER[CA]. In the table, the syntax [.] indicates that the instruction has a “record” form that updates CR[CR0], and a “non-record” form.

Table 2-22. Shift Instructions

Shift Left	Shift Right	Shift Right Algebraic
slw[.]	**srw**[.]	**sraw**[.] **srawi**[.]

2.11.6.4 Cache Management Instructions

These instructions control the operation of the ICU and DCU. Instructions are provided to fill or invalidate instruction cache blocks. Instructions are also provided to fill, flush, invalidate, or zero data cache blocks, where a block is defined as a 32-byte cache line.

Table 2-23 lists the PPC405 cache management instructions.

Table 2-23. Cache Management Instructions

DCU	ICU
dcba **dcbf** **dcbi** **dcbst** **dcbt** **dcbtst** **dcbz** **dccci** **dcread**	**icbi** **icbt** **iccci** **icread**

2.11.7 Interrupt Control Instructions

mfmsr and **mtmsr** read and write data between the MSR and a GPR to enable and disable interrupts. **wrtee** and **wrteei** enable and disable external interrupts. **rfi** and **rfci** return from interrupt handlers. *Table 2-24* lists the PPC405 interrupt control instructions.

Table 2-24. Interrupt Control Instructions

mfmsr **mtmsr** **rfi** **rfci** **wrtee** **wrteei**

2.11.8 TLB Management Instructions

The TLB management instructions read and write entries of the TLB array in the MMU, search the TLB array for an entry which will translate a given address, and invalidate all TLB entries. There is also an instruction for synchronizing TLB updates with other processors, but because the PPC405 is for use in uniprocessor environments, this instruction performs no operation.

Table 2-25 lists the TLB management instructions. In the table, the syntax [.] indicates that the instruction has a “record” form that updates CR[CR0], and a “non-record” form.

Table 2-25. TLB Management Instructions

tlbia **tlbre** **tlbsx**[**.**] **tlbsync** **tlbwe**

2.11.9 Processor Control Instructions

These instructions move data between the GPRs and SPRs, the CR, and DCRs in the PPC405, and provide traps, system calls, and synchronization controls.

Table 2-26 lists the processor management instructions in the PPC405.

Table 2-26. Processor Control Instructions

eieio **isync** **sync**	**mcrxr** **mfcr** **mfdcr** **mfspr**	**mtcrf** **mtdcr** **mtspr** **sc** **tw** **twi**

2.11.10 Extended Mnemonics

In addition to mnemonics for instructions supported directly by hardware, the PowerPC Architecture defines numerous extended mnemonics.

An extended mnemonic translates directly into the mnemonic of a hardware instruction, typically with carefully specified operands. For example, the PowerPC Architecture does not define a “shift right word immediate” instruction, because the “rotate left word immediate then AND with mask,” (rlwinm) instruction can accomplish the same result:

rlwinm RA,RS,32–n,n,31

However, because the required operands are not obvious, the PowerPC Architecture defines an extended mnemonic:

srwi RA,RS,n

Extended mnemonics transfer the problem of remembering complex or frequently used operand combinations to the assembler, and can more clearly reflect a programmer’s intentions. Thus, programs can be more readable.

Refer to the following chapter and appendixes for lists of the extended mnemonics:

- *Instruction Set* on page 157 lists extended mnemonics under the associated hardware instruction mnemonics.
- *Instruction Summary* on page 357 lists extended mnemonics alphabetically, along with the hardware instruction mnemonics.

Table B-5 in *Instructions by Category* on page 395 lists all extended mnemonics.

3. Cache Operations

The PPC405 incorporates two internal caches, a 16-KB instruction cache and a 16-KB data cache. Instructions and data can be accessed in the caches much faster than in main memory.

The instruction cache unit (ICU) controls instruction accesses to main memory and stores frequently used instructions to reduce the overhead of instruction transfers between the instruction pipeline and external memory. Using the instruction cache minimizes access latency for frequently executed instructions.

The data cache unit (DCU) controls data accesses to main memory and stores frequently used data to reduce the overhead of data transfers between the GPRs and external memory. Using the data cache minimizes access latency for frequently used data.

3.1 ICU Features

- Programmable address pipelining and prefetching for cache misses and non cacheable lines
- Support for non-cacheable hits from lines contained in the line fill buffer
- Programmable non cacheable requests to memory as 4 or 8 words (or half line or line)
- Bypass path for critical words
- Non-blocking cache for hits during fills
- Flash invalidate (one instruction invalidates entire cache)
- Programmable allocation for fetch fills, enabling program control of cache contents using the icbt instruction
- Virtually indexed, physically tagged cache arrays
- Support for 64- and 32-bit PLB slaves
- A rich set of cache control instructions

3.2 DCU Features

- Address pipelining for line fills
- Support for load hits from non cacheable and non-allocated lines contained in the line fill buffer
- Bypass path for critical words
- Non-blocking cache for hits during fills
- Write-back and write-through write strategies controlled by storage attributes
- Programmable non cacheable load requests to memory as lines or words.
- Handling of up to two pending line flushes.
- Holding of up to three stores before stalling the core pipeline
- Physically indexed, physically tagged cache arrays
- Support for 64- and 32-bit PLB slaves
- A rich set of cache control instructions

ICU Organization on page 69 and *DCU Organization* on page 72 describe the organization and provide overviews of the ICU and the DCU.

3.3 ICU Organization

The ICU manages instruction transfers between external cacheable memory and the instruction queue in the execution unit.

The ICU contains a two-way set-associative 16-KB cache memory. Each way is organized in 256 lines of eight words (eight instructions) each.

As shown in *Table 3-1*, tag ways A and B store instruction address bits A0:21 for each line in cache ways A and B. Instruction address bits A19:26 serve as the index to the cache array. The two cache lines that correspond to the same line index (one in each way) are called a congruence class.

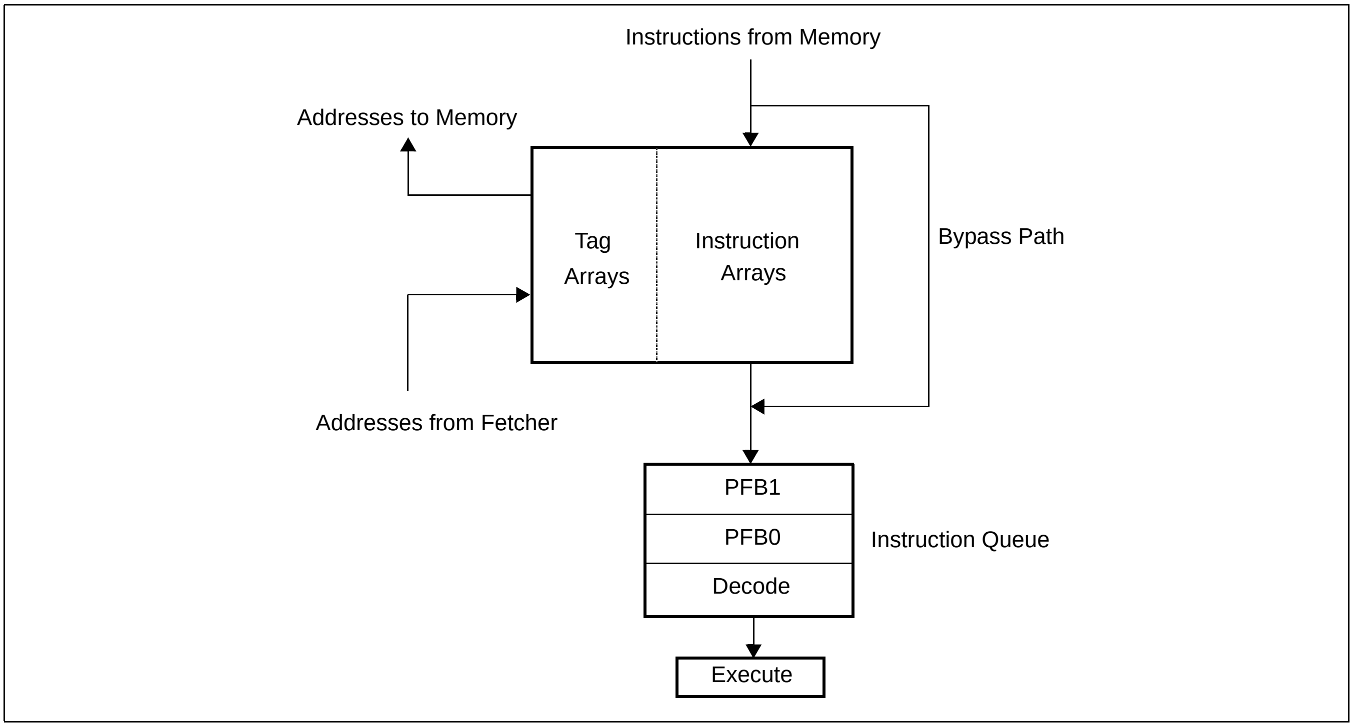
Table 3-1. Instruction Cache Organization

Tags (Two-way Set)		Instructions (Two-way Set)	
Way A	**Way B**	**Way A**	**Way B**
$A_{0:21}$ Line 0 A	$A_{0:21}$ Line 0 B	Line 0 A	Line 0 B
$A_{0:21}$ Line 1 A	$A_{0:21}$ Line 1 B	Line 1 A	Line 1 B
• • •	• • •	• • •	• • •
$A_{0:21}$ Line 254 A	$A_{0:21}$ Line 254 B	Line 254 A	Line 254 B
$A_{0:21}$ Line 255 A	$A_{0:21}$ Line 255 B	Line 255 A	Line 255 B

When a cache line is to be loaded, the cache way to receive the line is determined by using an least recently-used (LRU) policy. The index, determined by the instruction address, selects a congruence class. Within a congruence class, the line which was accessed most recently is retained, and the other line is marked as LRU, using an LRU bit in the tag array. The line to receive the incoming data is the LRU line. After the cache line fill, the LRU bit is then set to identify as least-recently-used the line opposite the line just filled.

Figure 3-1 shows the relationships between the ICU and the instruction pipeline.

Figure 3-1. Instruction Flow



3.3.1 ICU Operations

Instructions from cacheable memory regions are copied into the instruction cache array. The fetcher can access instructions much more quickly from a cache array than from memory. Cache lines are loaded either target-word-first or sequentially. Target-word-first fills start at the requested word, continue to the end of the line, and then wrap to fill the remaining words at the beginning of the line. Sequential fills start at the first word of the cache line and proceed sequentially to the last word of the line.

The bypass path handles instructions in cache-inhibited memory and improves performance during line fill operations. If a request from the fetcher obtains an entire line from memory, the queue does not have to wait for the entire line to reach the cache. The target word (the word requested by the fetcher) is sent on the bypass path to the queue while the line fill proceeds, even if the selected line fill order is not target-word-first.

Cache line fills always run to completion, even if the instruction stream branches away from the rest of the line. As requested instructions are received, they go to the fetcher from the fill register before the line fills in the cache. The filled line is always placed in the ICU; if an external memory subsystem error occurs during the fill, the line is not written to the cache. During a clock cycle, the ICU can send two instructions to the fetcher.

3.3.2 Instruction Cachability Control

When instruction address translation is enabled (MSR[IR] = 1), instruction cachability is controlled by the I storage attribute in the translation lookaside buffer (TLB) entry for the memory page. If TLB_entry[I] = 1, caching is inhibited; otherwise caching is enabled. Cachability is controlled separately for each page, which can range in size from 1KB to 16MB. *Translation Lookaside Buffer (TLB)* on page 92 describes the TLB.

When instruction address translation is disabled (MSR[IR] = 0), instruction cachability is controlled by the Instruction Cache Cachability Register (ICCR). Each field in the ICCR (ICCR[S0:S31]) controls the cachability of a 128MB region (see *Real-Mode Storage Attribute Control* on page 105). If ICCR[S*n*] = 1, caching is enabled for the specified region; otherwise, caching is inhibited.

The performance of the PPC405 is significantly lower while fetching instructions from cache inhibited regions.

Following system reset, address translation is disabled and all ICCR bits are reset to 0 so that no memory regions are cacheable. Before regions can be designated as cacheable, the ICU cache array must be invalidated. The iccci instruction must execute before the cache is enabled. Address translation can then be enabled, if required, and the TLB or the ICCR can then be configured for the required cachability.

3.3.3 Instruction Cache Synonyms

The following information applies only if instruction address translation is enabled (MSR[IR] = 1) and 1KB or 4KB page sizes are used. See *Memory Management* on page 91 for information about address translation and page sizes.

An instruction cache synonym occurs when the instruction cache array contains multiple cache lines from the same real address. Such synonyms result from combinations of:
- Cache array size
- Cache associativity
- Page size
- The use of effective addresses (EAs) to index the cache array

For example, the instruction cache array has a "way size" of 8KB (16KB array/2 ways). Thus, 11 bits ($EA_{19:29}$) are needed to select a word (instruction) in each way. For the minimum page size of 1KB, the low order eight bits ($EA_{22:29}$) address a word in a page. The high order address bits ($EA_{0:21}$) are translated to form a real address (RA), which the ICU uses to perform the cache tag match. Cache synonyms could occur because the index bits

($EA_{19:29}$) overlap the translated RA bits. For 1KB pages, overlap in $EA_{19:21}$ and $RA_{19:21}$ could result in as many as 8 synonyms. In other words, data from the same RA could occur as many as 8 locations in the cache array. Similarly, for 4KB pages, $EA_{0:19}$ are translated. Differences in EA_{19} and RA_{19} could result in as many as 2 synonyms. For the next largest page size (16KB), only EA $_{0:17}$ are translated. Because there is no overlap with index bits $EA_{19:21}$, synonyms do not occur.

In practice, cache synonyms occur when a real instruction page having multiple virtual mappings exists in multiple cache lines. For 1KB pages, all EAs differing in $EA_{19:21}$ must be cast out of cache, using an icbi instruction for each such EA (up to eight per cache line in the page). For 4KB pages, all EAs differing in EA19 must be cast out in the same manner (up to two per cache line in the page). For larger pages, cache synonyms do not occur, and casting out any of the multiple EAs removes the physical information from the cache.

Programming Note: To prevent the occurrence of cache synonyms, use only page sizes greater than the cache way size (8KB), if possible. For the PPC405, the minimum such page size is 16KB.

3.3.4 ICU Coherency

The ICU does not “snoop” external memory or the DCU. Programmers must follow special procedures for ICU synchronization when self-modifying code is used or if a peripheral device updates memory containing instructions.

The following code example illustrates the necessary steps for self-modifying code. This example assumes that *addr1* is both data and instruction cacheable.

```
stw     regN, addr1   # the data in regN is to become an instruction at addr1
dcbst   addr1         # forces data from the data cache to memory
sync                  # wait until the data actually reaches the memory
icbi    addr1         # the previous value at addr1 might already be in
                        the instruction cache; invalidate it in the cache
isync                 # the previous value at addr1 may already have been
                        pre-fetched into the queue; invalidate the queue
                        so that the instruction must be re-fetched
```

3.4 DCU Organization

The DCU manages data transfers between external cacheable memory and the general-purpose registers in the execution unit.

The DCU contains a two-way set-associative 16KB cache memory. Each way is organized in 256 lines of eight words (32 bytes) each.

As shown in Table 3-2, tag ways A and B store data address bits A0:19 for each line in cache ways A and B. Data address bits A18:26 serve as the index to the cache array. The two cache lines that correspond to the same line index (one in each way) are called a congruence class.

Table 3-2. Data Cache Organization

Tags (Two-way Set)		Data (Two-way Set)	
Way A	**Way B**	**Way A**	**Way B**
$A_{0:19}$ Line 0 A	$A_{0:19}$ Line 0 B	Line 0 A	Line 0 B
$A_{0:19}$ Line 1 A	$A_{0:19}$ Line 1 B	Line 1 A	Line 1 B
• • •	• • •	• • •	• • •
$A_{0:19}$ Line 254 A	$A_{0:19}$ Line 254 B	Line 254 A	Line 254 B
$A_{0:19}$ Line 255 A	$A_{0:19}$ Line 255 B	Line 255 A	Line 255 B

A bypass path handles data operations in cache-inhibited memory and improves performance during line fill operations.

3.4.1 DCU Operations

Data from cacheable memory regions are copied from external memory into lines in the data cache array so that subsequent cache operations result in cache hits. Loads and stores that hit in the DCU are completed in one cycle. For loads, GPRs receive the requested byte, halfword, or word of data from the data cache array. The DCU supports byte-writeability to improve the performance of byte and halfword store operations.

Cache operations require a line fill when they require data from cacheable memory regions that are not currently in the DCU. A line fill is the movement of a cache line (eight words) from external memory to the data cache array. Eight words are copied from external memory into the fill buffer, either targetword-first or sequentially. Loading order is controlled by the PLB slave. Target-word-first fills start at the requested word, continue to the end of the line, and then wrap to fill the remaining words at the beginning of the line. Sequential fills start at the first word of the cache line and proceed sequentially to the last word of the line. In both types of fills, the fill buffer, when full, is transferred to the data cache array. The cache line is marked valid when it is filled.

Loads that result in a line fill, and loads from non cacheable memory, are sent to a GPR. The requested byte, halfword, or word is sent from the DCU to the GPR from the fill buffer, using a cache bypass mechanism. Additional loads for data in the fill buffer can be bypassed to the GPR until the data is moved into the data array.

Stores that result in a line fill have their data held in the fill buffer until the line fill completes. Additional stores to the line being filled will also have their data placed in the fill buffer before being transferred into the data cache array.

To complete a line fill, the DCU must access the tag and data arrays. The tag array is read to determine the tag addresses, the LRU line, and whether the LRU line is dirty. A dirty cache line is one that was accessed by a store instruction after the line was established, and can be inconsistent with external memory. If the line being replaced is dirty, the address and the cache line must be saved so that external memory can be updated. During the cache line fill, the LRU bit is set to identify the line opposite the line just filled as LRU.

When a line fill completes and replaces a dirty line, a line flush begins. A flush copies updated data in the data cache array to main storage. Cache flushes are always sequential, starting at the first word of the cache line and proceeding sequentially to the end of the line.

Cache lines are always completely flushed or filled, even if the program does not request the rest of the bytes in the line, or if a bus error occurs after a bus interface unit accepts the request for the line fill. If a bus error occurs during a line fill, the line is filled and the data is marked valid. However, the line can contain invalid data, and a machine check exception occurs.

3.4.2 DCU Write Strategies

DCU operations can use write-back or write-through strategies to maintain coherency with external cacheable memory.

The write-back strategy updates only the data cache, not external memory, during store operations. Only modified data lines are flushed to external memory, and then only when necessary to free up locations for incoming lines, or when lines are explicitly flushed using **dcbf** or **dcbst** instructions. The write-back strategy minimizes the amount of external bus activity and avoids unnecessary contention for the external bus between the ICU and the DCU.

The write-back strategy is contrasted with the write-through strategy, in which stores are written simultaneously to the cache and to external memory. A write-through strategy can simplify maintaining coherency between cache and memory.

When data address translation is enabled (MSR[DR] = 1), the W storage attribute in the TLB entry for the memory page controls the write strategy for the page. If TLB_entry[W] = 0, write-back is selected; otherwise, write-through is selected. The write strategy is controlled separately for each page. *Translation Lookaside Buffer (TLB)* on page 92 describes the TLB.

When data address translation is disabled (MSR[DR] = 0), the Data Cache Write-through Register (DCWR) sets the storage attribute. Each bit in the DCWR (DCWR[W0:W31]) controls the write strategy of a 128MB storage region (see *Real-Mode Storage Attribute Control* on page 105). If DCWR[Wn] = 0, write-back is enabled for the specified region; otherwise, write-through is enabled.

Programming Note: The PowerPC Architecture does not support memory models in which write-through is enabled and caching is inhibited.

3.4.3 DCU Load and Store Strategies

The DCU can control whether a load receives one word or one line of data from main memory. For cacheable memory, the load without allocate (LWOA) field of the CCR0 controls the type of load resulting from a load miss. If CCR0[LWOA] = 0, a load miss causes a line fill. If CCR0[LWOA] = 1, load misses do not result in a line fill, but in a word load from external memory. For infrequent reads of non-contiguous memory, setting CCR0[LWOA] = 1 may provide a small performance improvement.

For non cacheable memory and for loads misses when CCR0[LWOA] = 1, the load word as line (LWL) field in the CCR0 affects whether load misses are satisfied with a word, or with eight words (the equivalent of a cache line) of data. If CCR0[LWL] = 0, only the target word is bypassed to the core. If CCR0[LWL] = 1, the DCU saves eight words (one of which is the target word) in the fill buffer and bypasses the target data to the core to satisfy the load word request. The fill buffer is not written to the data cache array.

Setting CCR0[LWL] = 1 provides the fastest accesses to sequential non cacheable memory. Subsequent loads from the same line are bypassed to the core from the fill buffer and do not result in additional external memory accesses. The load data remains valid in the fill buffer until one of the following occurs: the beginning of a subsequent load that requires the fill buffer, a store to the target address, a **dcbi** or **dccci** instruction issued to the target address, or the execution of a **sync** instruction. Non cacheable loads to guarded storage never cause a line transfer on the PLB even if CCR0[LWL] = 1. Subsequent loads to the same non cacheable storage are always requested again from the PLB.

For cacheable memory, the store without allocate (SWOA) field of the CCR0 controls the type of store resulting from a store miss. If CCR0[SWOA] = 0, a store miss causes a line fill. If CCR0[SWOA] = 1, store misses do not result in a line fill, but in a single word store to external memory.

3.4.4 Data Cachability Control

When data address translation is disabled (MSR[DR] = 0), data cachability is controlled by the Data Cache Cachability Register (DCCR). Each bit in the DCCR (DCCR[S0:S31]) controls the cachability of a 128MB region (see *Real-Mode Storage Attribute Control* on page 105). If DCCR[Sn] = 1, caching is enabled for the specified region; otherwise, caching is inhibited.

When data address translation is enabled (MSR[DR] = 1), data cachability is controlled by the I bit in the TLB entry for the memory page. If TLB_entry[I] = 1, caching is inhibited; otherwise caching is enabled. Cachability is controlled separately for each page, which can range in size from 1KB to 16MB. *Translation Lookaside Buffer (TLB)* on page 92 describes the TLB.

> **Programming Note:** The PowerPC Architecture does not support memory models in which write-through is enabled and caching is inhibited.

The performance of the PPC405 is significantly lower while accessing memory in cache-inhibited regions.

Following system reset, address translation is disabled and all DCCR bits are reset to 0 so that no memory regions are cacheable. The **dccci** instruction must execute 256 times before regions can be designated as cacheable. This invalidates all congruence classes before enabling the cache. Address translation can then be enabled, if required, and the TLB or the DCCR can then be configured for the desired cachability.

> **Programming Note:** If a data block corresponding to the effective address (EA) exists in the cache, but the EA is non cacheable, loads and stores (including **dcbz**) to that address are considered programming errors (the cache block should previously have been flushed). The only instructions that can legitimately access such an EA in the data cache are the cache management instructions **dcbf, dcbi, dcbst, dcbt, dcbtst, dccci,** and **dcread**.

3.4.5 DCU Coherency

The DCU does not provide snooping. Application programs must carefully use cache-inhibited regions and cache control instructions to ensure proper operation of the cache in systems where external devices can update memory.

3.5 Cache Instructions

For detailed descriptions of the instructions described in the following sections, see *Instruction Set* on page 157

In the instruction descriptions, the term "block" is synonymous with cache line. A block is the unit of storage operated on by all cache block instructions.

3.5.1 ICU Instructions

The following instructions control instruction cache operations:

icbi **Instruction Cache Block Invalidate**
Invalidates a cache block.

icbt **Instruction Cache Block Touch**
Initiates a block fill, enabling a program to begin a cache block fetch before the program needs an instruction in the block.
The program can subsequently branch to the instruction address and fetch the instruction without incurring a cache miss.
This is a privileged instruction.

iccci **Instruction Cache Congruence Class Invalidate**
Invalidates the instruction cache array.
This is a privileged instruction.

icread **Instruction Cache Read**
Reads either an instruction cache tag entry or an instruction word from an instruction cache line, typically for debugging. Fields in CCR0 control instruction behavior (see *Cache Control and Debugging Features* on page 77).
This is a privileged instruction.

3.5.2 DCU Instructions

Data cache flushes and fills are triggered by load, store and cache control instructions. Cache control instructions are provided to fill, flush, or invalidate cache blocks.

The following instructions control data cache operations:

dcba **Data Cache Block Allocate**
Speculatively establishes a line in the cache and marks the line as modified.
If the line is not currently in the cache, the line is established and marked as modified without actually filling the line from external memory.
If dcba references a non cacheable address, **dcba** is treated as a no-op.
If dcba references a cacheable address, write-through required (which would otherwise cause an alignment exception), **dcba** is treated as a no-op.

dcbf **Data Cache Block Flush**
Flushes a line, if found in the cache and marked as modified, to external memory; the line is then marked invalid.
If the line is found in the cache and is not marked modified, the line is marked invalid but is not flushed.
This operation is performed regardless of whether the address is marked cacheable.

dcbi **Data Cache Block Invalidate**
Invalidates a block, if found in the cache, regardless of whether the address is marked cacheable. Any modified data is not flushed to memory.
This is a privileged instruction.

dcbst **Data Cache Block Store**
Stores a block, if found in the cache and marked as modified, into external memory; the block is not invalidated but is no longer marked as modified.
If the block is marked as not modified in the cache, no operation is performed.
This operation is performed regardless of whether the address is marked cacheable.

dcbt **Data Cache Block Touch**
Fills a block with data, if the address is cacheable and the data is not already in the cache. If the address is non cacheable, this instruction is a no-op.

dcbtst **Data Cache Block Touch for Store**
Implemented identically to the **dcbt** instruction for compatibility with compilers and other tools.

dcbz **Data Cache Block Set to Zero**
Fills a line in the cache with zeros and marks the line as modified.
If the line is not currently in the cache (and the address is marked as cacheable and non-write-through), the line is established, filled with zeros, and marked as modified without actually filling the line from external memory. If the line is marked as either non cacheable or write-through, an alignment exception results.

dccci **Data Cache Congruence Class Invalidate**
Invalidates a congruence class (both cache ways).
This is a privileged instruction.

dcread **Data Cache Read**
Reads either a data cache tag entry or a data word from a data cache line, typically for debugging. Bits in CCR0 control instruction behavior (see *Cache Control and Debugging Features* on page 77).
This is a privileged instruction.

3.6 Cache Control and Debugging Features

Registers and instructions are provided to control cache operation and help debug cache problems. For ICU debug, the **icread** instruction and the Instruction Cache Debug Data Register (ICDBDR) are provided. See *ICU Debugging* on page 80 for more information. For DCU debug, the **dcread** instruction is provided. See *DCU Debugging* on page 81 for more information. CCR0 controls the behavior of the **icread** and the **dcread** instructions.

Figure 3-2. Core Configuration Register 0 (CCR0)

0:5		Reserved	
6	LWL	Load Word as Line 0 The DCU performs load misses or non-cacheable loads as words, halfwords, or bytes, as requested 1 For load misses or non cacheable loads, the DCU moves eight words (including the target word) into the line fill buffer	
7	LWOA	Load Without Allocate 0 Load misses result in line fills 1 Load misses do not result in a line fill, but in non cacheable loads	
8	SWOA	Store Without Allocate 0 Store misses result in line fills 1 Store misses do not result in line fills, but in non cacheable stores	
9	DPP1	DCU PLB Priority Bit 1 0 DCU PLB priority 0 on bit 1 1 DCU PLB priority 1 on bit 1	DCU logic dynamically controls DCU priority bit 0.

10:11	IPP	ICU PLB Priority Bits 0:1 00 Lowest ICU PLB priority 01 Next to lowest ICU PLB priority 10 Next to highest ICU PLB priority 11 Highest ICU PLB priority	
12	DPE	Data Cache Parity Enable 0 Disable 1 Enable	
13	DPP	Data Cache Parity Precision 0 Imprecise 1 Precise	
14	U0XE	Enable U0 Exception 0 Disables the U0 exception 1 Enables the U0 exception	
15	LDBE	Load Debug Enable 0 Disable 1 Enable	When enabled, load data is visible on data-side OCM.
16:17		Reserved	
18	IPE	Instruction Cache Parity Enable 0 Disable 1 Enable	
19	TPE	Translation Lookaside Buffer (TLB) Parity Enable 0 Disable 1 Enable	
20	PFC	ICU Prefetching for Cacheable Regions 0 Disables prefetching for cacheable regions 1 Enables prefetching for cacheable regions	
21	PFNC	ICU Prefetching for Non Cacheable Regions 0 Disables prefetching for non cacheable regions 1 Enables prefetching for non cacheable regions	
22	NCRS	Non cacheable ICU request size 0 Requests are for four-word lines 1 Requests are for eight-word lines	
23	FWOA	Fetch Without Allocate 0 An ICU miss results in a line fill. 1 An ICU miss does not cause a line fill, but results in a non cacheable fetch.	
24:26		Reserved	
27	CIS	Cache Information Select 0 Information is cache data. 1 Information is cache tag.	
28	PRS	Parity Read Select Information passed is selected by CCR0[CIS] and CCR0[CWS]. 0 Pass data or tag 1 Pass parity information	
29:30		Reserved	
31	CWS	Cache Way Select 0 Cache way is A. 1 Cache way is B.	

3.6.1 CCR0 Programming Guidelines

Several fields in CCR0 affect ICU and DCU operation. Altering these fields while the cache units are involved in PLB transfers can cause errant operation, including a processor hang.

To guarantee correct ICU and DCU operation, specific code sequences must be followed when altering CCR0 fields.

CCR0[IPP] and [FWOA] affect ICU operation. If these fields are altered, execution of the following code sequence (Sequence 1) is required:

```
!   SEQUENCE 1 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
!   sync
        sync
!   Touch code sequence into i-cache
        addis       RX,r0,seq1@h
        ori         RX,RX,seq1@l
        icbt        r0,RX
!   Call function to alter CCR0 bits
        b seq1
back:
!   Restore MSR to original value
        mtmsr       RM
                    •
                    •
                    •
!   The following function must be in cacheable memory
        .align 5                        ! Align CCR0 altering code on a cache line boundary.
        seq1:
        icbt        r0,RX               ! Repeat ICBT and execute an ISYNC to guarantee CCR0
        isync                           ! altering code has been completely fetched across the PLB.
        mfspr       RN,CCR0             ! Read CCR0.
        andi/ori    RN,RN,0xXXXX        ! Execute and/or function to change any CCR0 bits.
                                        ! Can use two instructions before having to touch
                                        ! in two cache lines.
        mtspr       CCR0, RN            ! Update CCR0.
        isync                           ! Refetch instructions under new processor context.
        b           back                ! Branch back to initialization code.
```

CCR0[DPP1] and [U0XE] affect DCU operation. If these fields are altered, execution of the following code sequence (Sequence 2) is required. Note that Sequence 1 includes Sequence 2, so Sequence 1 can be used to alter any CCR0 fields.

In the following sample code, registers RN, RM, RX, and RZ are any available GPRs.

```
!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
! sync
    sync
! Alter CCR0 bits
    mfspr       RN,CCR0             ! Read CCR0.
    andi/ori    RN,RN,0xXXXX        ! Execute and/or function to change any CCR0 bits.
    mtspr       CCR0, RN            ! Update CCR0.
    isync                           ! Refetch instructions under new processor context.
! Restore MSR to original value
    mtmsr       RM
```

CCR0[CIS, CWS] do not require special programming.

3.6.2 ICU Debugging

The **icread** instruction enables the reading of the instruction cache entries for the congruence class specified by EA18:26. The cache information is read into the ICDBDR; from there it can subsequently be moved, using an **mfspr** instruction, into a GPR. ICU tag information is placed into the ICDBDR as shown.

Figure 3-3. Instruction Cache Debug Data Register (ICDBDR)

0:21	TAG	Cache Tag	
22:26		Reserved	
27	V	Cache Line Valid Not valid Valid	
28:30		Reserved	
31	LRU	Least Recently Used (LRU) A-way LRU B-way LRU	

If CCR0[CIS] = 0, the data is a word of ICU data from the addressed line, specified by $EA_{27:29}$. If CCR0[CWS] = 0, the data is from the A-way; otherwise; the data from the B-way.

If CCR0[CIS] = 1, the cache information is the cache tag. If CCR0[CWS] = 0, the tag is from the A-way; otherwise, the tag is from the B-way.

Programming Note: The instruction pipeline does not wait for data from an **icread** instruction to arrive before attempting to use the contents the ICDBDR. The following code sequence ensures proper results:

```
icread     r5,r6     # read cache information
isync                # ensure completion of icread
mficdbdr   r7        # move information to GPR
```

3.6.3 DCU Debugging

The **dcread** instruction provides a debugging tool for reading the data cache entries for the congruence class specified by EA18:26. The cache information is read into a GPR.

If CCR0[CIS] = 0, the data is a word of DCU data from the addressed line, specified by EA27:29. If EA30:31 are not 00, an alignment exception occurs. If CCR0[CWS] = 0, the data is from the A-way; otherwise; the data is from the B-way.

If CCR0[CIS] = 1, the cache information is the cache tag. If CCR0[CWS] = 0, the tag is from the Away; otherwise the tag is from the B-way.

DCU tag information is placed into bits 0:19 of a GPR.

Note: A “dirty” cache line is one which has been accessed by a store instruction after it was established, and can be inconsistent with external memory.

3.7 DCU Performance

DCU performance depends upon the application, but, in general, cache hits complete in one cycle without stalling the CPU pipeline. Under certain conditions and limitations of the DCU, the pipeline stalls (stops executing instructions) until the DCU completes current operations.

Several factors affect DCU performance, including:
- Pipeline stalls
- DCU priority
- Simultaneous cache operations
- Sequential cache operations

3.7.1 Pipeline Stalls

The CPU issues commands for cache operations to the DCU. If the DCU can immediately perform the requested cache operation, no pipeline stall occurs. In some cases, however, the DCU cannot immediately perform the requested cache operation, and the pipeline stalls until the DCU can perform the pending cache operation.

In general, the DCU, when hitting in the cache array, can execute a load/store every cycle. If a cache miss occurs, the DCU must retrieve the line from main memory. For cache misses, the DCU stores the cache line in a line fill buffer until the entire cache line is received. The DCU can accept new DCU commands while the fill progresses. If the instruction causing the line fill is a load, the target word is bypassed to the GPR during the cycle after it becomes available in the fill buffer. When the fill buffer is full, it must be moved into the tag and data arrays. During this time, the DCU cannot begin a new cache operation and stalls the pipeline if new DCU commands are presented. Storing a line in the line fill buffer takes three cycles, unless the line being replaced has been modified. In that case, the operation takes four cycles.

The DCU can accept up to two load commands. If the data for the first load command is not immediately available, the DCU can still accept the second load command. If the load data is not required by subsequent instructions, those instructions will continue to execute. If data is required from either load command, the CPU pipeline will stall until the load data has been delivered. The pipeline will also stall until the second load has read the data array if a subsequent data cache command is issued.

In general, if the fill buffer is being used and the next load or store command requires the fill buffer, only one additional command can be accepted before causing additional DCU commands to stall the pipeline.

The DCU can accept up to three outstanding store commands before stalling the CPU pipeline for additional data cache commands.

The DCU can have two flushes pending before stalling the CPU pipeline.

DCU cache operations other than loads and stores stall the CPU pipeline until all prior data cache operations complete. Any subsequent data cache command will stall the pipeline until the prior operation is complete.

3.7.2 Cache Operation Priorities

The DCU uses a priority signal to improve performance when pipeline stalls occur. When the pipeline is stalled because of a data cache operation, the DCU asserts the priority signal to the PLB. The priority signal tells the external bus that the DCU requires immediate service, and is valid only when the data cache is requesting access to the PLB. The priority signal is asserted for all loads that require external data, or when the data cache is requesting the PLB and stalling an operation that is being presented to the data cache.

Table 3-3 provides examples of when the priority is asserted and deasserted.

Table 3-3. Priority Changes With Different Data Cache Operations

Instruction Requesting PLB	Priority	Next Instruction	Priority
Any load from external memory	1	N/A	N/A
Any store	0	Any other cache operation not being accepted by the DCU.	1
dcbf	0	Any cache hit.	0
dcbf/dcbst	0	Load non-cache.	1
dcbf/dcbst	0	Another command that requires a line flush.	1
dcbt	0	Any cache hit.	0
dcbi/dccci/dcbz	0	N/A	N/A

3.7.3 Simultaneous Cache Operations

Some cache operations can occur simultaneously to improve DCU performance. For example, combinations of line fills, line flushes, word load/stores, and operations that hit in the cache can occur simultaneously. Cache operations other than loads/stores cannot begin until the PLB completes all previous operations.

3.7.4 Sequential Cache Operations

Some common cache operations, when performed sequentially, can limit DCU performance: sequential loads/stores to non cacheable storage regions, sequential line fills, and sequential line flushes.

In the case of sequential cache hits, the most commonly occurring operations, the DCU loads or stores data every cycle. In such cases, the DCU does not limit performance.

However, when a load from a non cacheable storage region is followed by multiple loads from noncallable regions, the loads can complete no faster than every four cycles, assuming that the addresses are accepted during the same cycle in which it is requested, and that the data is returned during the cycle after the load is accepted.

Similarly, when a store to a non cacheable storage region is followed by multiple stores to non cacheable regions the fastest that the stores can complete is every other cycle. The DCU can have accepted up to three stores before additional DCU commands will stall waiting for the prior stores to complete.

Sequential line fills can limit DCU performance. Line fills occur when a load/store or **dcbt** instruction misses in the cache, and can be pipelined on the PLB interface such that up to two requests can be accepted before stalling subsequent requests. The subsequent operations will wait in the DCU until the first line fill completes. The line fills must complete in the order that they are accepted.

Sequential line flushes from the DCU to main memory also limit DCU performance. Flushes occur when a line fill replaces a valid line that is marked dirty (modified), or when a **dcbf** instruction flushes a specific line. If two flushes are pending, the DCU stalls any new data cache operations until the first flush finishes and the second flush begins.

4. On-Chip Memory (OCM)

The on-chip memory (OCM) subsystem consists of a memory controller that connects the PPC405 processor core to an SRAM array. OCM is ideal for applications requiring low latency access to critical instructions and data. OCM can provide performance that is identical to cache hits, yet, unlike a cache, the OCM never misses. Instructions and data stored in the OCM are always available because OCM contents only change under program control. Therefore, if the programmer avoids instruction-side and data-side OCM access contention, OCM can provide information availability that is superior to a cache line locking scheme. OCM is superior because it can provide single cycle performance identical to cache hits without locking down portions of the cache. This results in more effective cache utilization for the processor.

Instructions and data returned from OCM interface do not flow through the PPC405 CPU caches. The caches remain available for caching from other memory sources accessed across the PLB interface. The system designer must ensure that each address has a single access path into the PPC405 CPU for a given software process. Each address that is requested should be found in either the OCM address space or the PLB address space, but not in both.

Code to initialize OCM should execute in non-OCM address space in a region marked as non cacheable. The initialization code should invalidate the cache arrays (in the ICU and DCU, as appropriate) to ensure that no addresses to be programmed as OCM space are in the cache. After programming the OCM address and control registers, the OCM address space should remain marked as non-cacheable.

Read and write accesses to the OCM array share a single access port. OCM accesses have the following priorities:

1. Data-side OCM reads (loads)
2. Data-side OCM writes (stores)
3. Instruction-side OCM read (fetches)

Data-side OCM reads occur in one cycle. Data-side writes also complete in one cycle, though they can be preempted by higher priority data-side reads. Instruction-side OCM reads occur by default (that is, after a reset) in two cycles. However, when the Instruction-Side Two-Cycle Mode field of the OCM Instruction-Side Control Register (if it exists) is set to 0, instruction-side OCM reads occur in one cycle, unless preempted by higher priority data-side transfers. Two-cycle mode is provided for chips that cannot make instruction-side timing to the processor core. The PPC405 processor core, however, meets the timing requirement. Therefore, programmers should set the OCM Instruction-Side Control Register (if it exists) to 0 during chip initialization.

The OCM can also transfer data between the PLB and internal SRAM.

The OCM has the following features:

- Supports two non-overlapping memory banks configurable as 16 KB
- Simultaneous PLB, Instruction-side OCM and Data-side OCM access
- PLB3 slave cycles support the following
  - 64 bit slave attachment addressable by any PLB master
  - Single-beat read and write (1 to 8 bytes)
  - 4-, 8- and 16-word line read and write
  - Doubleword and word read and write bursts
  - Slave-terminated doubleword and word bursts
  - Master-terminated variable-length bursts
  - Data parity generation and checking
  - Read/Write protection per bank
- Instruction-side interface supports the following data parity checking
- Data-side interface supports the following:
  - 1-wait state OCM access with 1-deep write buffer

- Data parity generation and checking
- Read/Write protection per bank
- Processor side data port has the highest access priority (maintains predictable memory accesses to the OCM).

4.1 OCM Addressing

The address space for the instruction-side OCM and the data side OCM are defined by the OCM Instruction-Side Address Range Compare Register (OCM0_ISARC) and OCM Data-Side Address Range Compare Register (OCM0_DSARC), respectively. These registers are implemented as 6-bit registers that define the most significant address bits of the respective OCM address space. Using six bits defines a 64MB address space. The instruction side and data side can share a 64MB address space, or each can have its own 64MB address space. The address spaces are fully relocatable on 64MB boundaries within the 4GB address space of the PPC405, but the programmer must assign OCM address space to avoid conflicts with other assigned addresses. See *Programming Model* on page 31 for information about the PPC405 memory map.

Figure 4-1. OCM Address Usage

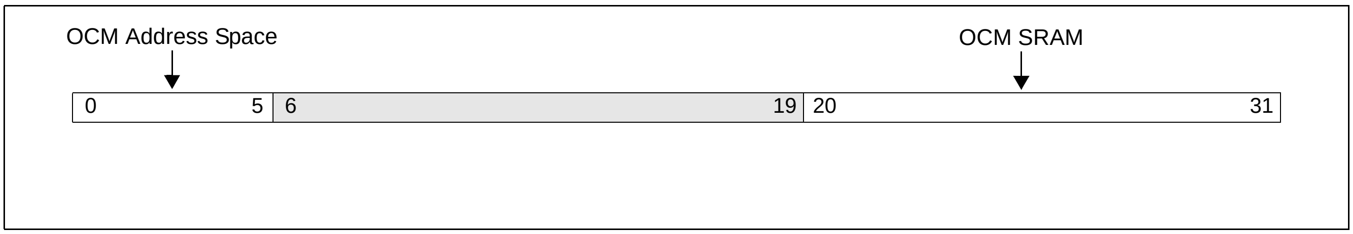


Figure 4-1 illustrates OCM address usage. The OCM SRAM array size is 4KB. Address bits 20:31 select byte addresses for data-side accesses. Address bits 30:31 are ignored for instruction-side accesses, because instruction-side accesses return either one or two words per transfer.

Note that the instruction-side and data-side OCM address spaces overlap physically, even if defined as distinct logical address spaces, because the 4KB SRAM is shared. There is no distinction between data space or instruction space, except as defined by the programmer.

Addresses in the OCM array are aliased throughout the larger OCM address spaces. The larger OCM address spaces are filled with multiple images of the 4KB SRAM. Aliased addresses refer to the same physical memory locations.

> **Programming Note:** To avoid possible memory coherency problems when using aliased addresses, align aliased addresses on 16KB boundaries rather than on 4KB boundaries. See *Store Data Bypass Behavior and Memory Coherency* on page 86 for details.

If address translation is enabled (MSR[IR, DR] = 1), one or more TLB entries for the OCM address space must exist to validate accesses. However, the virtual addresses are not translated, and 32-bit effective addresses (virtual addresses) are presented to OCM.

Data-side OCM contents can use big endian or little endian byte ordering. Instruction-side OCM contents must use big endian byte ordering. See *Byte Ordering* on page 44 for detailed information about byte ordering.

4.2 Store Data Bypass Behavior and Memory Coherency

The OCM subsystem provides only one mechanism, data-side store operations, for writing both instructions and data into the OCM array. However, two independent mechanisms request read access of OCM contents; one for instruction-side fetches and the other for data-side loads.

The following description applies only to applications that alias the OCM address space and perform a mix of data-side loads and stores. It does not apply to applications that use data-side stores only to initialize OCM with instructions.

If a data-side OCM store is followed in the next cycle by a data-side load, the load actually accesses the OCM array before the store. This is due to the nature of the processor pipeline, the cycle availability of the store data, and the fact that data-side loads have a higher priority than data-side stores. In this scenario, store data is queued in a register while the load accesses the array. Further, if the store is immediately followed by a sequence of consecutive loads, it remains in the queue until the last of the consecutive loads has accessed the OCM array. The queued store data is written into the OCM array in the first cycle that does not have a data-side load operation accessing the array.

Consider a scenario where such a situation causes store data to be held in the store data queue. If any of the loads access the same address as the address of the store operation whose data is being held in the store data queue, there is a need to bypass the store data from the store data queue to provide the correct data to the load operation.

A bypass is determined to be required by comparing the pending store address with the load address. However, the comparison is done with a 16KB address representation for the load and store operations, not the 4KB address (the physical size of the PPC405 OCM array). If the 16KB address compares, the store data is bypassed to the load operation. This implies that a bypass results for address aliasing only when the OCM addresses match at a 16KB multiple, which corresponds to a match of address bits 18:29 (a word address that is further specified by byte enables). Although the physical address space is aliased at 4KB multiples, the bypass determination is made at 16KB multiples. Therefore, if bits 18:19 of an aliased load address do not match bits 18:19 of the 16KB store address of the data being held in the store data queue, the load data will not be coherent. Instead of returning the most recently stored data, which is being held in the store data queue, the load returns “old” data previously stored in and accessed from the OCM array.

Table 4-1 provides examples that describe bypass behavior when address aliasing is used.

Table 4-1. Examples of Store Data Bypass

Example	Store Address	Load Address	4KB Aliased Address	16KB Aliased Address	Bypass
1	0x00000100	0x00000100	Same	Same	Yes
2	0x00000100	0x00000400	No	No	No
3	0x00000100	0x00001100	Yes	No, loads old data	No
4	0x00000100	0x00005100	Yes	No, loads old data	No
5	0x00000100	0x00004100	Yes	Yes	Yes
6	0x00000100	0x00008100	Yes	Yes	Yes

Example 1 provides the most basic example, in which the load and store addresses are the same. This results in the load accessing the queued store data, bypassing the OCM array to satisfy the load.

Example 2 shows two different addresses that are not aliased (both addresses are in the 4KB SRAM address space). No bypass occurs, and the load returns the correct data from the OCM array.

Examples 3 and 4 show aliased addresses that do not bypass data because the addresses do not compare within a 16KB address space. In both examples, address bits 18:19 do not match. In both examples, the load does not return the most recently stored data from the store data queue; the load returns the “old’ data from the array. To avoid such problems, alias on 16KB boundaries. If addresses are aliased on 4KB boundaries, place at least one instruction that does not access the data-side OCM between a load and a store to the same aliased address so the store data has a cycle to be written into the array.

Examples 5 and 6 bypass data out of the store data queue because the aliased addresses compare within a 16KB address space. In both examples, address bits 18:29 match, and load data is returned from the store data queue.

4.3 OCM Registers

The OCM controller uses Device Control Registers (DCRs) to store or access data in the OCM. DCRs are unique to the chip in which this processor is instantiated and are not a part of the processor. Refer to the appropriate chip user’s manual for details on the DCRs.

5. Memory Management

The PPC405 memory management unit (MMU) performs address translation and protection functions. With appropriate system software, the MMU supports:

- Translation of effective addresses to real addresses
- Independent enabling of instruction and data address translation and protection
- Page-level access control using the translation mechanism
- Software control of page replacement strategy
- Additional virtual-mode control of protection using zones
- Real-mode write protection

5.1 MMU Overview

The instruction and integer units generate 32-bit effective addresses (EAs) for instruction fetches and data accesses, respectively. Instruction EAs are generated for sequential instruction fetches, and for instruction fetches causing changes in program flow (branches and interrupts). Data EAs are generated for load/store and cache control instructions. The MMU translates EAs into real addresses; the instruction cache unit (ICU) and data cache unit (DCU) use real addresses to access memory.

The PPC405 MMU supports demand-paged virtual memory and other memory management schemes that depend on precise control of effective to real address mapping and flexible memory protection. Translation misses and protection faults cause precise interrupts. Sufficient information is available to correct the fault and restart the faulting instruction.

The MMU divides storage into pages. A page represents the granularity of EA translation and protection controls. Eight page sizes (1KB, 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB) are simultaneously supported. A valid entry for a page containing the EA to be translated must be in the translation lookaside buffer (TLB) for address translation to be performed. EAs for which no valid TLB entry exists cause TLB-miss interrupts.

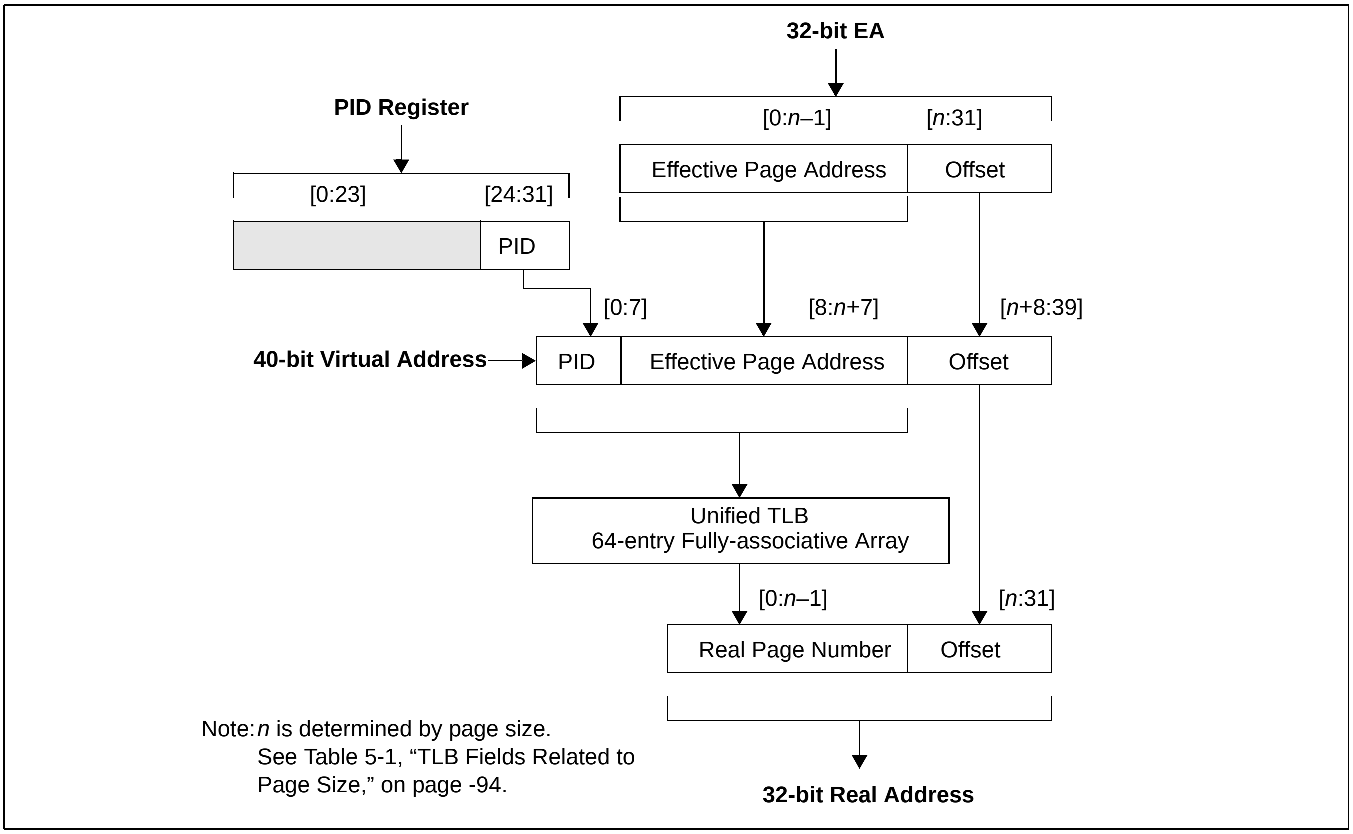
5.2 Address Translation

Fields in the Machine State Register (MSR) control the use of the MMU for address translation. The instruction relocate (IR) field of the MSR controls translation for instruction accesses. The data relocate (DR) field of the MSR controls the translation mechanism for data accesses. These fields, specified independently, can be changed at any time by a program in supervisor state. Note that all interrupts clear MSR[IR, DR] and place the processor in the supervisor state. Subsequent discussion about translation and protection assumes that MSR[IR, DR] are set, enabling address translation.

The processor references memory when it fetches an instruction, and when it executes load/store, branch, and cache control instructions. Processor accesses to memory use EAs to references a memory location. When translation is enabled, the EA is translated into a real address, as illustrated in *Figure 5-1* on page 92. The ICU or DCU uses the real address for the access. (When translation is not enabled, the EA is already a real address.)

In address translation, the EA is combined with an 8-bit process ID (PID) to create a 40-bit virtual address. The virtual address is compared to all of the TLB entries. A matching entry supplies the real address for the storage reference. *Figure 5-1* on page 92 illustrates the process.

Figure 5-1. Effective-to-Real Address Translation Flow



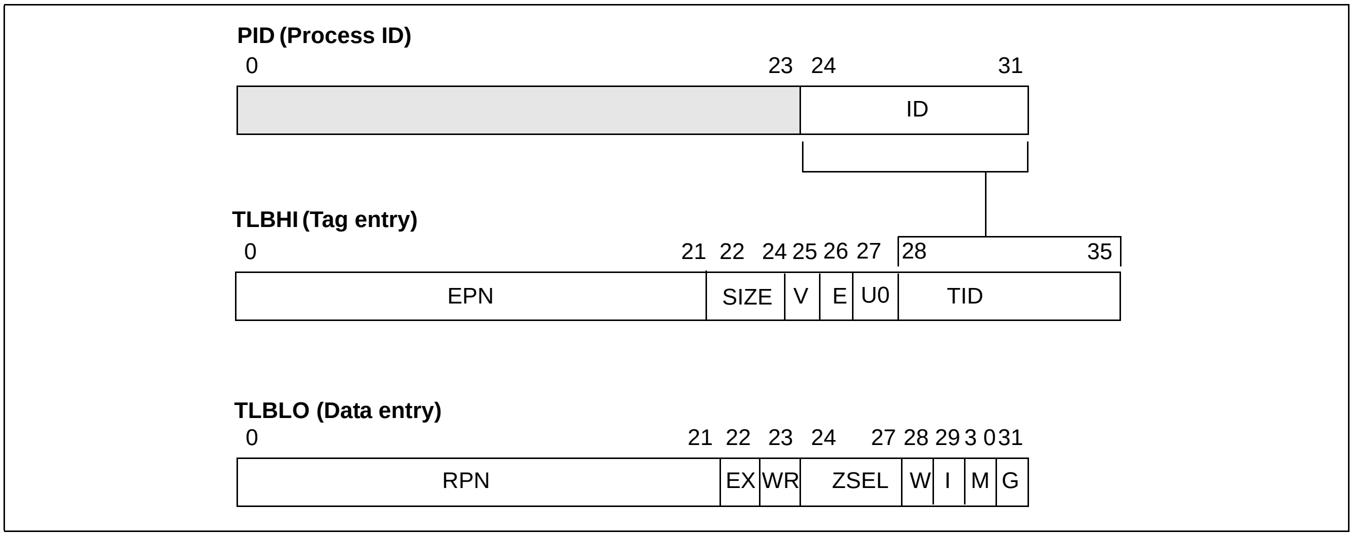
5.3 Translation Lookaside Buffer (TLB)

The TLB is hardware that controls translation, protection, and storage attributes. The instruction and data units share a unified fully-associative TLB, in which any page entry (TLB entry) can be placed anywhere in the TLB. TLB entries are maintained under program control. System software determines the TLB entry replacement strategy and the format and use of page state information. A TLB entry contains the information required to identify the page, to specify translation and protection controls, and to specify the storage attributes.

5.3.1 Unified TLB

The unified TLB (UTLB) contains 64 entries; each has a TLBHI (tag) portion and a TLBLO (data) portion, as described in *Figure 5-2* on page 93. TLBHI contains 36 bits; TLBLO contains 32 bits. When translation is enabled, the UTLB tag portion compares some or all of EA0:21 with some or all of the effective page number EPN0:21, based on the size bits SIZE0:2. All 64 entries are simultaneously checked for a match. If an entry matches, the corresponding data portion of the UTLB provides the real page number (RPN), access control bits (ZSEL, EX, WR), and storage attributes (W, I, M, G, E, U0).

Figure 5-2. TLB Entries



The virtual address space is extended by adding an 8-bit translation ID (TID) loaded from the Process ID (PID) register during a TLB access. The PID identifies one of 255 unique software entities, usually used as a process or thread ID. TLBHI[TID] is compared to the PID during a TLB look-up.

Tag and data entries are written by copying data from GPRs and the PID, using the tlbwe instruction. Tag and data entries are read by copying data to GPRs and the PID, using the tlbre instruction. Software can search for specific entries using the tlbsx instruction.

5.3.2 TLB Fields

Each TLB entry describes a page that is enabled for translation and access controls. Fields in the TLB entry fall into four categories:

- Information required to identify the page to the hardware translation mechanism
- Control information specifying the translation
- Access control information
- Storage attribute control information

5.3.2.1 Page Identification Fields

When an EA is presented to the MMU for processing, the MMU applies several selection criteria to each TLB entry to select the appropriate entry. Although it is possible to place multiple entries into the TLB to match a specific EA and PID, this is considered a programming error, and the result of a TLB lookup for such an EA is undefined. The following fields in the TLB entry identify the page. Except as noted, all comparisons must succeed to validate an entry for subsequent use.

EPN (effective page number, 22 bits)

Compared to some number of the $EA_{0:21}$ bits presented to the MMU. The number of bits corresponds to the page size.

The exact comparison depends on the page size, as shown in Table 5-1.

Table 5-1. TLB Fields Related to Page Size

Page Size	SIZE Field	*n* Bits Compared	EPN to EA Comparison	RPN Bits Set to 0
1KB	000	22	$EPN_{0:21} \leftrightarrow EA_{0:21}$	—
4KB	001	20	$EPN_{0:19} \leftrightarrow EA_{0:19}$	$RPN_{20:21}$
16KB	010	18	$EPN_{0:17} \leftrightarrow EA_{0:17}$	$RPN_{18:21}$
64KB	011	16	$EPN_{0:15} \leftrightarrow EA_{0:15}$	$RPN_{16:21}$
256KB	100	14	$EPN_{0:13} \leftrightarrow EA_{0:13}$	$RPN_{14:21}$
1MB	101	12	$EPN_{0:11} \leftrightarrow EA_{0:11}$	$RPN_{12:21}$
4MB	110	10	$EPN_{0:9} \leftrightarrow EA_{0:9}$	$RPN_{10:21}$
16MB	111	8	$EPN_{0:7} \leftrightarrow EA_{0:7}$	$RPN_{8:21}$

SIZE (page size, 3 bits)

Selects one of the eight page sizes, 1KB–16MB, listed in Table 5-1.

V (valid,1 bit)

Indicates whether a TLB entry is valid and can be used for translation.

A valid TLB entry implies read access, unless overridden by zone protection. TLB_entry[V] can be written using a **tlbwe** instruction. The **tlbia** instruction invalidates all TLB entries.

TID (translation ID, 8 bits)

Loaded from the PID register during a **tlbwe** operation. The TID value is compared with the PID value during a TLB access. The TID provides a convenient way to associate a translation with one of 255 unique software entities, typically a process or thread ID maintained by operating system software. Setting TLBHI_entry[TID] = 0x00 disables TID-PID comparison and identifies a TLB entry as valid for all processes; the value of the PID register is then irrelevant.

5.3.2.2 Translation Field

When a TLB entry is identified as matching an EA (and possibly the PID), TLBLO_entry[RPN] defines how the EA is translated.

RPN (real page number, 22 bits)

Replaces some, or all, of $EA_{0:21}$, depending on page size. For example, a 16KB page uses $EA_{0:17}$ for comparison. The translation mechanism replaces $EA_{0:17}$ with $TLBLO\_entry[RPN]_{0:17}$ to form the physical address, and $EA_{18:31}$ becomes the real page offset, as illustrated in *Figure 5-1* on page 92.

> **Programming Note:** Software must set all unused bits of RPN (as determined by page size) to 0. See Table 5-1.

5.3.2.3 Access Control Fields

Several access controls are available in the UTLB entries.

ZSEL (zone select, 4 bits)

Selects one of 16 zone fields (Z0—Z15) from the Zone Protection Register (ZPR). The ZPR field bits can modify the access protection specified by the TLB_entry[V, EX, WR] bits of a TLB entry. Zone protection is described in detail in “Zone Protection” on page 103.

EX (execute enable, 1 bit)

When set (TLBLO_entry[EX] = 1), enables instruction execution at addresses within a page. ZPR settings can override TLBLO_entry[EX]; see “Zone Protection” on page 103, for more information.

WR (write-enable 1 bit)

When set (TLBLO_entry[WR] = 1), enables store operations to addresses in a page. ZPR settings can override TLBLO_entry[WR]; see “Zone Protection” on page 103.

5.3.2.4 Storage Attribute Fields

TLB entries contain bits that control and provide information about the storage control attributes. Four of the attributes (W, I, M, and G) are defined in the PowerPC Architecture. The E storage attribute is defined in the PowerPC Embedded Environment.

W (write-through,1 bit)

When set (TLBLO_entry[W] = 1), stores are specified as write-through. If data in the referenced page is in the data cache, a store updates the cached copy of the data and the external memory location. Contrast this with a write-back strategy, which updates memory only when a cache line is flushed.

In real mode, the Data Cache Write-through Register (DCWR) controls the write strategy.

Note that the PowerPC Architecture does not support memory models in which write-through is enabled and caching is inhibited. It is considered a programming error to use these memory models; the results are undefined.

I (caching inhibited,1 bit)

When set (TLBLO_entry[I] = 1), a memory access is completed by using the location in main memory, bypassing the cache arrays. During the access, the accessed location is not put into the cache arrays.

In real mode, the Instruction Cache Cachability Register (ICCR) and Data Cache Cachability Register (DCCR) control cachability. In these registers, the setting of the bit is reversed; 1 indicates that a storage control region is cacheable, rather than caching inhibited.

Note that the PowerPC Architecture does not support memory models in which write-through is enabled and caching is inhibited. It is considered a programming error to use these memory models; the results are undefined.

It is considered a programming error if the target location of a load/store, **dcbz**, or fetch access to caching inhibited storage is in the cache; the results are undefined. It is not considered a programming error for the target locations of other cache control instructions to be in the cache when caching is inhibited.

M (memory coherent,1 bit)

For implementations that support multiprocessing, the M storage attribute improves the performance of memory coherency management. Because the PPC405 does not provide multi-processor support or hardware support for data coherency, the M bit is implemented, but has no effect.

G (guarded,1 bit)

When set (TLBLO_entry[G] = 1), indicates that the hardware cannot speculatively access the location for pre-fetching or out-of-order load access. The G storage attribute is typically used to protect memory-mapped I/O from inadvertent access. Attempted execution of an instruction from a guarded data storage address while instruction address translation is enabled results in an instruction storage interrupt because data storage and memory mapped I/O (MMIO) addresses are not used to contain instructions.

An instruction fetch from a guarded region does not occur until the execution pipeline is empty, thus guaranteeing that the access is necessary and therefore not speculative. For this reason, performance is degraded when executing out of guarded regions, and software should avoid unnecessarily marking regions of instruction storage as guarded.

In real mode, the Storage Guarded Register (SGR) controls guarding.

U0 (user-defined attribute, 1 bit)

When set (TLBLO[U0] = 1), indicates the user-defined attribute applies to the data in the associated page.

In real mode, the Storage User-defined 0 Register (SU0R) controls the setting of the U0 storage attribute.

E (endian, 1 bit)

When set (TLBLO[E] = 1), indicates that data in the associated page is stored in true little endian format.

In real mode, the Storage Little-Endian Register (SLER) controls the setting of the E storage attribute.

5.3.3 Shadow Instruction TLB

To enhance performance, four instruction-side TLB entries are kept in a four-entry fully-associative shadow array. This array, called the instruction TLB (ITLB), helps to avoid TLB contention between instruction accesses to the TLB and load/store operations. Replacement and invalidation of the ITLB entries is managed by hardware. See “Shadow TLB Consistency” on page 97 for details.

The ITLB can be considered a level-1 instruction-side TLB; the UTLB serves as the level-2 instruction-side TLB. The ITLB is used only during instruction fetches for storing instruction address translations. Each ITLB entry contains the translation information for a page. The processor uses the ITLB for address translation of instruction accesses when MSR[IR] = 1.

5.3.3.1 ITLB Accesses

The instruction unit accesses the ITLB independently of the rest of the MMU. ITLB accesses are transparent to the executing program, except that ITLB hits contribute to higher overall instruction throughput by allowing data address translations to occur in parallel. Therefore, when instruction accesses hit in the ITLB, the address translation mechanisms in the UTLB are available for use by data accesses simultaneously.

The ITLB requests a new entry from the UTLB when an ITLB miss occurs. A four-cycle latency occurs at each ITLB miss that is also a UTLB hit; the latency is longer if it is also a UTLB miss, or if there is contention for the UTLB from the data side. A round-robin replacement algorithm replaces existing entries with new entries.

5.3.4 Shadow Data TLB

To enhance performance, eight data-side TLB entries are kept in a eight-entry fully-associative shadow array. This array, called the data TLB (DTLB), helps to avoid TLB contention between instruction accesses to the TLB and load/store operations. Replacement and invalidation of the DTLB entries is managed by hardware. See “Shadow TLB Consistency” on page 97 for details.

The DTLB can be considered a level-1 data-side TLB; the UTLB serves as the level-2 data-side TLB. The DTLB is used only during instruction execute for storing data address translations. Each DTLB entry contains the translation information for a page. The processor uses the DTLB for address translation of data accesses when MSR[DR] = 1.

5.3.4.1 1 DTLB Accesses

The execute unit accesses the DTLB independently of the rest of the MMU. DTLB accesses are transparent to the executing program, except that DTLB hits contribute to higher overall instruction throughput by allowing instruction address translations to occur in parallel. Therefore, when data accesses hit in the DTLB, the address translation mechanisms in the UTLB are available for use by instruction accesses simultaneously.

The DTLB requests a new entry from the UTLB when a DTLB miss occurs. A three-cycle latency occurs at each DTLB miss that is also a UTLB hit; the latency is longer if it is also a UTLB miss. If there is contention for the UTLB from the instruction side, the data side has priority. A round-robin replacement algorithm replaces existing entries with new entries.

5.3.5 Shadow TLB Consistency

To help maintain the integrity of the shadow TLBs, the processor invalidates the ITLB and DTLB contents when the following context-synchronizing events occur:

- **isync** instruction
- Processor context switch (all interrupts, **rfi, rfci**)
- **sc** instruction

If software updates a translation/protection mechanism (UTLB, PID, ZPR, or MSR) and must synchronize these updates with the ITLB and DTLB, the software must perform the necessary context synchronization.

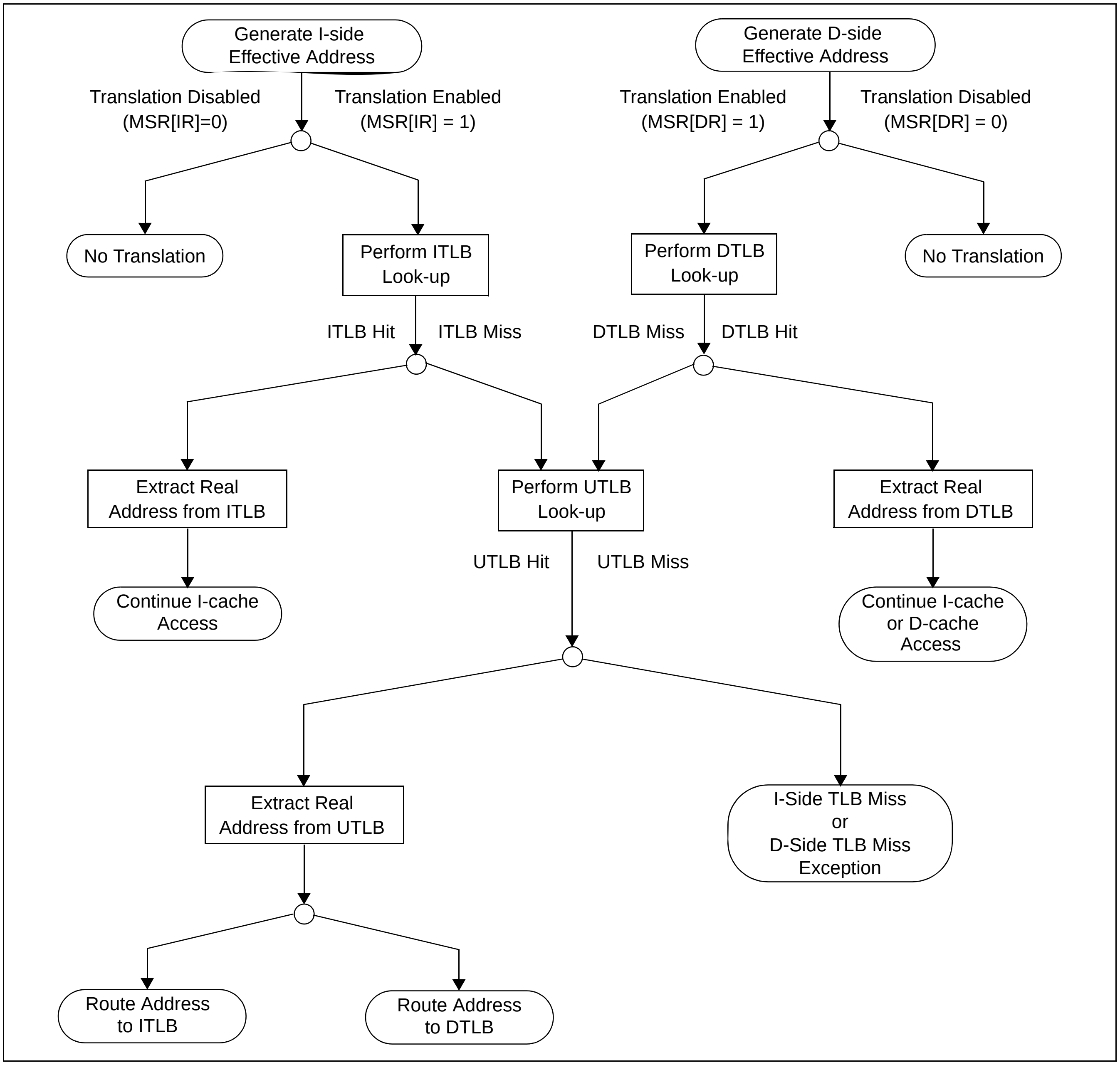
A typical example is the manipulation of the TLB by an operating system within an interrupt handler for a TLB miss. Upon entry to the interrupt handler, the contents of the ITLB and DTLB are invalidated and translation is disabled. If the operating system simply made the TLB updates and returned from the handler (using **rfi** or **rfci**), no additional explicit software action would be required to synchronize the ITLB and DTLB.

If, instead, the operating system enables translation within the handler and then performs TLB updates within the handler, those updates would not be effective in the ITLB and DTLB until rfi or rfci is executed to return from the handler. For those TLB updates to be reflected in the ITLB and DTLB within the handler, an **isync** must be issued after TLB updates finish. Failure to properly synchronize the shadow TLBs can cause unexpected behavior.

> **Programming Note:** As a rule of thumb, follow software manipulation of an translation mechanism (if performed while translation is active) with a context-synchronizing operation (usually **isync**).

Figure 5-3 illustrates the relationship of the shadow TLBs and UTLB in address translation:

Figure 5-3. ITLB/DTLB/UTLB Address Resolution



5.4 TLB-Related Interrupts

The processor relies on interrupt handling software to implement paged virtual memory, and to enforce protection of specified memory pages.

When an interrupt occurs, the processor clears MSR[IR, DR]. Therefore, at the start of all interrupt handlers, the processor operates in real mode for instruction accesses and data accesses. Note that when address translation is disabled for an instruction fetch or load/store, the EA is equal to the real address and is passed directly to the memory subsystem (including cache units). Such untranslated addresses bypass all memory protection checks that would otherwise be performed by the MMU.

When translation is enabled, MMU accesses can result in the following interrupts:

- Data storage interrupt
- Instruction storage interrupt
- Data TLB miss interrupt
- Instruction TLB miss interrupt

5.4.1 Data Storage Interrupt

A data storage interrupt is generated when data address translation is active, and the desired access to the EA is not permitted for one of the following reasons:

- In the problem state
  - **icbi**, load/store, **dcbz**, or **dcbf** with an EA whose zone field is set to no access (ZPR[Z*n*] = 00). In this case, **dcbt** and **dcbtst** no-op, rather than cause an interrupt. Privileged instructions cannot cause data storage interrupts.
  - Stores, or **dcbz**, to an EA having TLB[WR] = 0 (write access disabled) and ZPR[Z*n*] ≠ 11. (The privileged instructions **dcbi** and **dccci** are treated as “stores”, but cause program interrupts, rather than data storage interrupts.)
- In supervisor state
  - Data store, **dcbi**, **dcbz**, or **dccci** to an EA having TLB[WR] = 0 and ZPR[Z*n*] other than 11 or 10.

dcba does not cause data storage exceptions (cache line locking or protection). If conditions occur that would otherwise cause such an exception, **dcba** is treated as a no-op.

Zone Protection on page 103 describes zone protection in detail. See *Data Storage Interrupt* on page 120 for a detailed discussion of the data storage interrupt.

5.4.2 Instruction Storage Interrupt

An instruction storage interrupt is generated when instruction address translation is active and the processor attempts to execute an instruction at an EA for which fetch access is not permitted, for any of the following reasons:

- In the problem state
  - Instruction fetch from an EA with ZPR[Z*n*] = 00.
  - Instruction fetch from an EA having TLB_entry[EX] = 0 and ZPR[Z*n*] ≠ 11.
  - Instruction fetch from an EA having TLB_entry[G] = 1.

- In the supervisor state
  - Instruction fetch from an EA having TLB_entry[EX] = 0 and ZPR[Z*n*] other than 11 or 10.
  - Instruction fetch from an EA having TLB_entry[G] = 1.

See *Zone Protection* on page 103 for a detailed discussion of zone protection. See *Instruction Storage Interrupt* on page 121 for a detailed discussion of the instruction storage interrupt.

5.4.3 Data TLB Miss Interrupt

A data TLB miss interrupt is generated if data address translation is enabled and a valid TLB entry matching the EA and PID is not present. The interrupt applies to data access instructions and cache operations (excluding cache touch instructions).

See *Data TLB Miss Interrupt* on page 127 for a detailed discussion.

5.4.4 Instruction TLB Miss Interrupt

The instruction TLB miss interrupt is generated if instruction address translation is enabled and execution is attempted for an instruction for which a valid TLB entry matching the EA and PID for the instruction fetch is not present.

See *Instruction TLB Miss Interrupt* on page 127 for a detailed discussion.

5.5 TLB Management

The processor does not imply any format for the page tables or the page table entries because there is no hardware support for page table management. Software has complete flexibility in implementing a replacement strategy, because software does the replacing. For example, software can "lock" TLB entries that correspond to frequently used storage by electing to never replace them, so that those entries are never cast out of the TLB.

TLB management is performed by software with some hardware assist, consisting of:

- Storage of the missed EA in the Save/Restore Register 0 (SRR0) for an instruction-side miss, or in the Data Exception Address Register (DEAR) for a data-side miss.
- Instructions for reading, writing, searching, and invalidating the TLB, as described briefly in the following subsections. See *Instruction Set* on page 157 for detailed instruction descriptions.

5.5.1 TLB Search Instructions (tlbsx/tlbsx.)

tlbsx locates entries in the TLB, to find the TLB entry associated with an interrupt, or to locate candidate entries to cast out. tlbsx searches the UTLB array for a matching entry. The EA is the value to be matched; EA = (RA|0)+(RB).

If the TLB entry is found, its index is placed in RT26:31. RT can then serve as the source register for a **tlbre** or **tlbwe** instruction to read or write the entry, respectively. If no match is found, the contents of RT are undefined.

tlbsx. sets the Condition Register (CR) bit $CR0_{EQ}$. The value of $CR0_{EQ}$ depends on whether an entry is found: $CR0_{EQ}$ = 1 if an entry is found; $CR0_{EQ}$ = 0 if no entry is found.

5.5.2 TLB Read/Write Instructions (tlbre/tlbwe)

TLB entries can be accessed for reading and writing by **tlbre** and **tlbwe**, respectively. Separate extended mnemonics are available for the TLBHI (tag) and TLBLO (data) portions of a TLB entry.

5.5.3 TLB Invalidate Instruction (tlbia)

tlbia sets TLB_entry[V] = 0 to invalidate all TLB entries. All other TLB entry fields remain unchanged.

Using **tlbwe** to set TLB_entry[V] = 0 invalidates a specific TLB entry.

5.5.4 TLB Sync Instruction (tlbsync)

tlbsync guarantees that all TLB operations have completed for all processors in a multi-processor system. PPC405 provides no multiprocessor support, so this instruction performs no function. The instruction is included to facilitate code portability.

5.6 Recording Page References and Changes

When system software manages virtual memory, the software views physical memory as a collection of pages. Each page is associated with at least one TLB entry. To manage memory effectively, system software often must know whether a particular page has been referenced or modified. Note that this involves more than knowing whether a particular TLB entry was used to reference or alter memory, because multiple TLB entries can translate to the same page.

When system software manages a demand-paged environment, and the software needs to replace the contents of a page with other data, previously referenced pages (accessed for any purpose) are more likely to be maintained than pages that were never referenced. If the contents of a page must be replaced, and data contained in that page was modified, system software generally must write the contents of the modified page to the backing store before replacing its contents. System software must maintain records to control the environment.

Similarly, when system software manages TLB entries, the software often must know whether a particular TLB entry was referenced. When the system software must select a TLB entry to cast out, previously referenced entries are more likely to be maintained than entries which were never referenced. System software must also maintain records for this purpose.

The PPC405 does not provide hardware reference or change bits, but TLB miss interrupts and data storage interrupts enable system software to maintain reference information for TLB entries and their associated pages, respectively.

A possible algorithm follows. First, the TLB entries are built, with each TLB_entry[V, WR] = 0. System software retains the index and EPN of each entry.

The first attempt by application code to access a page causes a TLB miss interrupt, because its TLB entry is marked invalid. The TLB miss handler records the reference to the TLB entry (and to the associated page) in a data structure, then sets TLB_entry[V] = 1. (Note that TLB_entry[V] can be considered a reference bit for the TLB entry.) Subsequent read accesses to the page associated with the TLB entry proceed normally.

In the example just given for recording TLB entry references, the first write access to the page using the TLB entry, after the entry is made valid, causes a data storage interrupt because write access was turned off. The TLB miss handler records the write to the page in a data structure, for use as a “changed” flag, then sets TLB_entry[WR] = 1 to enable write access. (Note that TLB_entry[WR] can be considered a change bit for the page.) Subsequent write accesses to the page proceed normally.

5.7 Access Protection

The PPC405 provides virtual-mode access protection. The TLB entry enables system software to control general access for programs in the problem state, and control write and execute permissions for all pages. The TLB entry can specify zone protection that can override the other access control mechanisms supported in the TLB entries.

TLB entry and zone protection methods also support access controls for cache operation and string loads/stores.

5.7.1 Access Protection Mechanisms in the TLB

For MMU access protection to be in effect, one or both of MSR[IR] or MSR[DR] must be set to one to enable address translation. MSR[IR] enables protection on instruction fetches, which are inherently read-only. MSR[DR] enables protection on data accesses (loads/stores).

5.7.1.1 General Access Protection

The translation ID (TLB_entry[TID]) provides the first level of MMU access protection. This 8-bit field, if non-zero, is compared to the contents of TLB_entry[PID]. These fields must match in a valid TLB entry if any access is to be allowed. In typical use, it is assumed that a program in the supervisor state, such as a real-time operating system, sets the process ID (PID) before starting a problem state program that is subject to access control.

If TLB_entry[TID] = 0x00, the associated memory page is accessible to all programs, regardless of their PID. This enables multiple processes to share common code and data. The common area is still subject to all other access protection mechanisms. *Figure 5-4* illustrates the Process ID Register.

Figure 5-4. Process ID (PID)

0:23		Reserved	
24:31		Process ID	

5.7.1.2 Execute Permissions

If instruction address translation is enabled, instruction fetches are subject to MMU translation and have MMU access protection. Fetches are inherently read-only, so write protection is not needed. Instead, using TLB_entry[EX], a memory page is marked as executable (contains instructions) or not executable (contains only data or memory-mapped control hardware).

If an instruction is pre-fetched from a memory page for which TLB_entry[EX] = 0, the instruction is tagged as an error. If the processor subsequently attempts to execute this instruction, an instruction storage interrupt results. This interrupt is precise with respect to the attempted execution. If the fetcher discards the instruction without attempting to execute it, no interrupt will result.

Zone protection can alter execution protection.

5.7.1.3 Write Permissions

If MSR[DR] = 1, data loads and stores are subject to MMU translation and are afforded MMU access protection. The existence of a TLB entry describing a memory page implies read access; write access is controlled by TLB_entry[WR].

If a store (including those caused by dcbz, dcbi, or dccci) is made to an EA having TLB_entry[WR] = 0, a data storage interrupt results. This interrupt is precise.

Zone protection can alter write protection (see “Zone Protection” on page 103). In addition, only zone protection can prevent read access of a page defined by a TLB entry.

5.7.1.4 Zone Protection

Each TLB entry contains a 4-bit zone select (ZSEL) field. A zone is an arbitrary identifier for grouping TLB entries (memory pages) for purposes of protection. As many as 16 different zones may be defined. Any zone can have any number of member pages.

Each zone is associated with a 2-bit field (Z0-Z15) in the ZPR. The values of the field define how protection is applied to all pages that are member of that zone. Changing the value of the ZPR field can alter the protection attributes of all pages in the zone. Without ZPR, the change would require finding, reading, altering, and rewriting the TLB entry for each page in a zone, individually. The ZPR provides a much faster means of altering the protection for groups of memory pages.

The ZSEL values 0-15 select ZPR fields Z0-Z15, respectively.

The fields are defined within the ZPR as follows:

While it is common for TLB_entry[EX, WR] to be identical for all member pages in a group, this is not required. The ZPR field alters the protection defined by TLB_entry[EX] and TLB_entry[WR], on a page-by-page basis, as shown in the ZPR illustration. An application program (presumed to be running in the problem state) can have execute and write permissions as defined by TLB_entry[EX] and TLB_entry[WR] for the individual pages, or no access (denies loads, as well as stores and execution), or complete access. *Figure 5-5* shows the Zone Protection Register.

Figure 5-5. Zone Protection Register (ZPR)

0:1	Z0	TLB page access control for all pages in this zone. In the problem state (MSR[PR] = 1): 00 No access 01 Access controlled by applicable TLB_entry[EX, WR] 10 Access controlled by applicable TLB_entry[EX, WR] 11 Accessed as if execute and write permissions (TLB_entry[EX, WR]) are granted	In the supervisor state (MSR[PR] = 0): 00 Access controlled by applicable TLB_entry[EX, WR] 01 Access controlled by applicable TLB_entry[EX, WR] 10 Accessed as if execute and write permissions (TLB_entry[EX, WR]) are granted 11 Accessed as if execute and write permissions (TLB_entry[EX, WR]) are granted
2:3	Z1	See the description of Z0.	
4:5	Z2	See the description of Z0.	
6:7	Z3	See the description of Z0.	
8:9	Z4	See the description of Z0.	
10:11	Z5	See the description of Z0.	
12:13	Z6	See the description of Z0.	
14:15	Z7	See the description of Z0.	
16:17	Z8	See the description of Z0.	
18:19	Z9	See the description of Z0.	
20:21	Z10	See the description of Z0.	
22:23	Z11	See the description of Z0.	
24:25	Z12	See the description of Z0.	

26:27	Z13	See the description of Z0.	
28:29	Z14	See the description of Z0.	
30:31	Z15	See the description of Z0.	

Setting ZPR[Zn] = 00 for a ZPR field is the only way to deny read access to a page defined by an otherwise valid TLB entry. TLB_entry[EX] and TLB_entry[WR] do not support read protection. Note that the icbi instruction is considered a load with respect to access protection; executed in user mode, it causes a data storage interrupt if MSR[DR] = 1 and ZPR[Zn] = 00 is associated with the EA.

For a given ZPR field value, a program in supervisor state always has equal or greater access than a program in the problem state. System software can never be denied read (load) access for a valid TLB entry.

5.7.2 Access Protection for Cache Control Instructions

Architecturally the instructions **dcba, dcbi,** and **dcbz** are treated as "stores" because they can change data, or cause loss of data by invalidating a dirty line (a modified cache block).

Table 5-2 summarizes the conditions under which the cache control instructions can cause data storage interrupts.

Table 5-2. Protection Applied to Cache Control Instructions

Instruction	Possible Data Storage interrupt	
	When ZPR[Zn] = 00	**When TLB_entry[WR] = 0**
dcba	No (instruction no-ops)	No (instruction no-ops)
dcbf	Yes	No
dcbi	No	Yes
dcbst	Yes	No
dcbt	No (instruction no-ops)	No
dcbtst	No (instruction no-ops)	No
dcbz	Yes	Yes
dccci	No	Yes
dcread	No	No
icbi	Yes	No
icbt	No (instruction no-ops)	No
iccci	No	No
icread	No	No

If data address translation is enabled, and write permission is denied (TLB_entry[WR] = 0), **dcbi** and **dcbz** can cause data storage interrupts. **dcbz** can cause a data storage interrupt when executed in the problem state and all access is denied (ZPR[Zn] = 00); **dcbi** cannot cause a data storage interrupt because it is a privileged instruction.

The **dcba** instruction enables "speculative" line establishment in the cache arrays; the established lines do not cause a line fill. Because the effects of **dcba** are speculative, interrupts that would otherwise result when ZPR[Zn] = 00 or TLB_entry[WR] = 0 do not occur. In such cases, **dcba** is treated as a no-op.

The **dccci** instruction can also be considered a "store" because it can change data by invalidating a dirty line; however, **dccci** is not address-specific (it affects an entire congruence class regardless of the operand address of the instruction). To restrict possible damage from an instruction which can change data and yet avoids the protection mechanism, the dccci instruction is privileged.

If data address translation is enabled, **dccci** can cause data storage interrupts when TLB_entry[WR] = 0; the operand is treated as if it were address-specific. **dccci** cannot cause a data storage interrupt when ZPR[Zn] = 00, because it is a privileged instruction.

Because **dccci** can cause data storage and TLB -miss interrupts, use of **dccci** is not recommended when MSR[DR] = 1; if **dccci** is used. Note that the specific operand address can cause an interrupt.

Architecturally, **dcbt** and **dcbtst** are treated as "loads" because they do not change data; they cannot cause data storage interrupts when TLB_entry[WR] = 0.

The cache block touch instructions **dcbt** and **dcbtst** are considered "speculative" loads; therefore, if a data storage interrupt would otherwise result from the execution of **dcbt** or **dcbtst** when ZPR[Zn] = 00, the instruction is treated as a no-op and the interrupt does not occur. Similarly, TLB miss interrupts do not occur for these instructions.

Architecturally, **dcbf** and **dcbst** are treated as "loads". Flushing or storing a line from the cache is not architecturally considered a "store" because a store was performed to update the cache, and **dcbf** or **dcbst** only update main memory. Therefore, neither dcbf nor **dcbst** can cause data storage interrupts when TLB_entry[WR] = 0. Because neither instruction is privileged, they can cause data storage interrupts when ZPR[Zn] = 00 and data address translation is enabled.

dcread is a "load" from a non-specific address, and is privileged. Therefore, it cannot cause data storage interrupts when ZPR[Zn] = 00 or TLB_entry[WR] = 0.

icbi and **icbt** are considered "loads" and cannot cause data storage interrupts when TLB_entry[WR] = 0. **icbi** can cause data storage interrupts when ZPR[Zn] = 00.

The **iccci** instruction cannot change data; an instruction cache line cannot be dirty. The **iccci** instruction is privileged and is considered a load. It does not cause data storage interrupts when ZPR[Zn] = 00 or TLB_entry[WR] = 0.

Because **iccci** can cause a TLB miss interrupt, using **iccci** is not recommended when data address translation is enabled; if it is used, note that the specific operand address can cause an interrupt.

icread is considered a "load" from a non-specific address, and is privileged. Therefore, it cannot cause data storage interrupts when ZPR[Zn] = 00 or TLB_entry[WR] = 0.

5.7.3 Access Protection for String Instructions

The **stswx** instruction with string length equal to 0(XER[TBC] = 0) is a no-op.

When data address translation is enabled and the Transfer Byte Count (TBC) field of the Fixed Point Exception Register (XER) is 0, neither **lswx** nor **stswx** can cause TLB miss interrupts, or data storage interrupts when ZPR[Zn] = 0 or TLB_entry[WR] = 0.

5.8 Real-Mode Storage Attribute Control

The PowerPC Architecture and the PowerPC Embedded Environment define several SPRs to control the following storage attributes in real mode: W, I, G,U0, and E. Note that the U0 and E attributes are not defined in the PowerPC Architecture. The E attribute is defined in the PowerPC Embedded Environment, and the U0 attribute is implementation-specific. No storage attribute control register is implemented for the M storage attribute because the PPC405 does not provide multi-processor support or hardware support for data coherency.

These SPRs, called storage attribute control registers, control the various storage attributes when address translation is disabled. When address translation is enabled, these registers are ignored, and the storage attributes supplied by the TLB entry are used (see *TLB Fields* on page 93).

The storage attribute control registers divide the 4GB real address space into thirty-two 128MB regions. In a storage attribute control register, bit 0 controls the lowest addressed 128MB region, bit 1 the next higher-addressed 128MB region, and so on. EA0:4 specify a storage control region.

For detailed information on the function of the storage attributes, see “Storage Attribute Fields” on page 95.

5.8.1 Storage Attribute Control Registers

Figure 5-6 shows a generic storage attribute control register. The storage attribute control registers have the same bit numbering and address ranges.

Figure 5-6. Generic Storage Attribute Control Register

Bit	Address Range	Bit	Address Range
0	0x0000 0000–0x07FF FFFF	16	0x8000 0000–0x87FF FFFF
1	0x0800 0000–0x0FFF FFFF	17	0x8800 0000–0x8FFF FFFF
2	0x1000 0000–0x17FF FFFF	18	0x9000 0000–0x97FF FFFF
3	0x1800 0000–0x1FFF FFFF	19	0x9800 0000–0x9FFF FFFF
4	0x2000 0000–0x27FF FFFF	20	0xA000 0000–0xA7FF FFFF
5	0x2800 0000–0x2FFF FFFF	21	0xA800 0000–0xAFFF FFFF
6	0x3000 0000–0x37FF FFFF	22	0xB000 0000–0xB7FF FFFF
7	0x3800 0000–0x3FFF FFFF	23	0xB800 0000–0xBFFF FFFF
8	0x4000 0000–0x47FF FFFF	24	0xC000 0000–0xC7FF FFFF
9	0x4800 0000–0x4FFF FFFF	25	0xC800 0000–0xCFFF FFFF
10	0x5000 0000–0x57FF FFFF	26	0xD000 0000–0xD7FF FFFF
11	0x5800 0000–0x5FFF FFFF	27	0xD800 0000–0xDFFF FFFF
12	0x6000 0000–0x67FF FFFF	28	0xE000 0000–0xE7FF FFFF
13	0x6800 0000–0x6FFF FFFF	29	0xE800 0000–0xEFFF FFFF
14	0x7000 0000–0x77FF FFFF	30	0xF000 0000–0xF7FF FFFF
15	0x7800 0000–0x7FFF FFFF	31	0xF800 0000–0xFFFF FFFF

5.8.1.1 Data Cache Write-through Register (DCWR)

The DCWR controls write-through policy (the W storage attribute) for the data cache unit (DCU). Write-through is not applicable to the instruction cache unit (ICU).

After any reset, all DCWR bits are set to 0, which establishes a write-back write strategy for all regions.

The PowerPC Architecture does not support memory models in which write-through is enabled and caching is inhibited.

5.8.1.2 Data Cache Cachability Register (DCCR)

The DCCR controls the I storage attribute for data accesses and cache management instructions. Note that the polarity of the bits in this register is opposite to that of the I attribute in the TLB; DCCR[Sn] = 1 enables caching, while TLB_entry[I] = 1 inhibits caching.

After any reset, all DCCR bits are set to 0. No memory regions are cacheable. Before memory regions can be designated as cacheable in the DCCR, it is necessary to execute the dccci instruction once for each congruence class in the DCU cache array. This procedure invalidates all congruence classes. The DCCR can then be reconfigured, and the DCU can begin normal operation.

The PowerPC Architecture does not support memory models in which write-through is enabled and caching is inhibited.

5.8.1.3 Instruction Cache Cachability Register (ICCR)

The ICCR controls the I storage attribute for instruction fetches. Note that the polarity of the bits in this register is opposite of that of the I attribute (ICCR[Sn] = 1 enables caching, while TLB_entry[I] = 1 inhibits caching).

After any reset, all ICCR bits are set to 0. No memory regions are cacheable. Before memory regions can be designated as cacheable in the ICCR, it is necessary to execute the iccci instruction. This procedure invalidates all congruence classes. The ICCR can then be reconfigured, and the ICU can begin normal operation.

5.8.1.4 Storage Guarded Register (SGR)

The SGR controls the G storage attribute for instruction and data accesses.

This attribute does not affect data accesses; the PPC405 does not perform speculative loads or stores.

After any reset, all SGR bits are set to 1, marking all storage as guarded. For best performance, system software should clear the guarded attribute of appropriate regions as soon as possible. If MSR[IR] = 1, the G attribute comes from the TLB entry. Attempting to execute from a guarded region in translate mode causes an instruction storage interrupt. See *Instruction Storage Interrupt* on page 121 for more information.

5.8.1.5 Storage User-defined 0 Register (SU0R)

The Storage User-defined 0 Register (SU0R) controls the user-defined (U0) storage attribute for instruction and data accesses.

After any reset, all SU0R bits are set to 0.

5.8.1.6 Storage Little-Endian Register (SLER)

The SLER controls the E storage attribute for instruction and data accesses.

This attribute determines the byte ordering of storage. *Byte Ordering* on page 44 provides a detailed description of byte ordering in the PowerPC Embedded Environment.

After any reset, all SLER bits are set to 0 (big endian).

6. Interrupt Handling

An interrupt is the action in which the processor saves its old context (MSR and instruction pointer) and begins execution at a pre-determined interrupt-handler address, with a modified MSR. *Exceptions* are events which, if enabled, cause the processor to take an interrupt. Exceptions are generated by signals from internal and external peripherals, instructions, internal timer facilities, debug events, or error conditions.

Table 6-2 on page 113 lists the interrupts handled by the PPC405 in the order of interrupt vector offsets. Detailed descriptions of each interrupt follow, in the same order. *Table 6-2* also provides an index to the descriptions.

Several registers support interrupt handling and control.*General Interrupt Handling Registers* on page 114 describes the general interrupt handling registers:

- Data Exception Address Register (DEAR)
- Exception Syndrome Register (ESR)
- Exception Vector Prefix Register (EVPR)
- Machine State Register (MSR)
- Save/Restore Registers (SRR0–SRR3)

6.1 Architectural Definitions and Behavior

Precise interrupts are those for which the instruction pointer saved by the interrupt must be either the address of the excepting instruction or the address of the next sequential instruction. Imprecise interrupts are those for which it is possible (but not required) for the saved instruction pointer to be something else, possibly prohibiting guaranteed software recovery.

Note that "precise" and "imprecise" are defined assuming that the interrupts are unmasked (enabled to occur) when the associated exception occurs. Consider an exception that would cause a precise interrupt, if the interrupt was enabled at the time of the exception, but that occurs while the interrupt is masked. Some exceptions of this type can cause the interrupt to occur later, immediately upon its enabling. In such a case, the interrupt is not considered precise with respect to the enabling instruction, but imprecise ("delayed precise") with respect to the cause of the exception.

Asynchronous interrupts are caused by events which are independent of instruction execution. All asynchronous interrupts are precise, and the following rules apply:

1. All instructions prior to the one whose address is reported to the interrupt handling routine (in the save/restore register) have completed execution. However, some storage accesses generated by these preceding instructions may not have completed.
2. No subsequent instruction has begun execution, including the instruction whose address is reported to the interrupt handling routine.
3. The instruction having its address reported to the interrupt handler may appear not to have begun execution, or may have partially completed.

Synchronous interrupts are caused directly by the execution (or attempted execution) of instructions. Synchronous interrupts can be either precise or imprecise.

For synchronous precise interrupts, the following rules apply:

1. The save/restore register addresses either the instruction causing the exception or the next sequential instruction. Which instruction is addressed is determined by the interrupt type and status bits.

2. All instructions preceding the instruction causing the exception have completed execution. However, some storage accesses generated by these preceding instructions may not have completed.
3. The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have partially completed, or may have completed, depending on the interrupt type.
4. No subsequent instruction has begun execution.

Refer to *PowerPC Embedded Environment* for an architectural description of imprecise interrupts.

Machine check interrupts are a special case typically caused by some kind of hardware or storage subsystem failure, or by an attempt to access an invalid address. A machine check can be indirectly caused by the execution of an instruction, but not recognized or reported until long after the processor has executed past the instruction that caused the machine check. As such, machine check interrupts cannot properly be thought of as synchronous, nor as precise or imprecise. For machine checks, the following general rules apply:

1. No instruction following the one whose address is reported to the machine check handler in the save/restore register has begun execution.
2. The instruction whose address is reported to the machine check handler in the save/restore register, and all previous instructions, may or may not have completed successfully. All previous instructions that would ever complete have completed, within the context existing before the machine check interrupt. No further interrupt (other than possible additional machine checks) can occur as a result of those instructions.

6.2 Behavior of the PPC405 Implementation

All interrupts, except for machine checks, are handled precisely. Precise handling implies that the address of the excepting instruction (for synchronous exceptions other than the system call exception), or the address of the next instruction to be executed (asynchronous exceptions and the system call exception), is passed to an interrupt handling routine. Precise handling also implies that all instructions that precede the instruction whose address is reported to the interrupt handling routine have executed and that no subsequent instruction has begun execution. The specific instruction whose address is reported may not have begun execution or may have partially completed, as specified for each precise interrupt type.

Synchronous precise interrupts include most debug event interrupts, program interrupts, instruction and data storage interrupts, TLB miss interrupts, system call interrupts, and alignment interrupts.

Asynchronous precise interrupts include the critical and noncritical external interrupts, and can be caused by on-chip peripherals, timer facility interrupts, and some debug events.

In the PPC405, machine checks are handled as critical interrupts (see *Critical and Noncritical Interrupts* on page 112). If a machine check is associated with an instruction fetch, the critical interrupt save/restore register contains the address of the instruction being fetched when the machine check occurred.

The synchronism of instruction-side machine checks (errors that occur while attempting to fetch an instruction from external memory) requires further explanation. Fetch requests to cacheable memory that miss in the instruction cache unit (ICU) cause an instruction cache line fill (eight words). If any instructions (words) in the fetched line are associated with an exception, an interrupt occurs upon attempted execution and the cache line is invalidated.

It is improper to declare an exception when an erroneous word is passed to the fetcher; the address could be the result of an incorrect speculative access. It is quite likely that no attempt will be made to execute an instruction from the erroneous address. An instruction-side machine check interrupt occurs only when execution is attempted. If an exception occurs, execution is suppressed, SRR2 contains the erroneous address, and the indicates that an instruction-side machine check occurred. Although such an interrupt is clearly asynchronous to the erroneous memory access, it is handled synchronously with respect to the attempted execution from the erroneous address.

Except for machine checks, all PPC405 interrupts are handled precisely:

- The address of the excepting instruction (for synchronous exceptions, other than the system call exception) or the address of the next sequential instruction (for asynchronous exceptions and the system call exception) is passed to the interrupt handling routine.
- All instructions that precede the instruction whose address is reported to the interrupt handling routine have completed execution and that no subsequent instruction has begun execution. The specific instruction whose address is reported might not have begun execution or might have partially completed, as specified for each interrupt type.

6.3 Interrupt Handling Priorities

The PPC405 processor handles only one interrupt at a time. Multiple simultaneous interrupts are handled in the priority order shown in *Table 6-1* (assuming, of course, that the interrupt types are enabled). Multiple interrupts can exist simultaneously, each of which requires the generation of an interrupt. The architecture does not provide for simultaneously reporting more than one interrupt of the same class (critical or non-critical). Therefore, interrupts are ordered with respect to each other. A masking mechanism is available for certain persistent interrupt types.

When an interrupt type is masked, and an event causes an exception which would normally generate an interrupt of that type, the exception persists as a *status* bit in a register. However, no interrupt is generated. Later, if the interrupt type is enabled (unmasked), and the exception status has not been cleared by software, the interrupt due to the original exception event is finally generated.

All asynchronous interrupt types can be masked. In addition, certain synchronous interrupt types can be masked.

Table 6-1. Interrupt Handling Priorities

Priority	Interrupt Type	Critical or Noncritical	Causative Conditions
1	Machine check—data	Critical	External bus error during data-side access
2	Debug—IAC	Critical	IAC debug event (in internal debug mode)
3	Machine check—instruction	Critical	Attempted execution of instruction for which an external bus error occurred during fetch
4	Debug—EXC, UDE	Critical	EXC or UDE debug event (in internal debug mode)
5	Critical interrupt input	Critical	Active level on the critical interrupt input by the UIC
6	Watchdog timer—first time-out	Critical	Posting of an enabled first time-out of the watchdog timer in the TSR
7	Instruction TLB Miss	Noncritical	Attempted execution of an instruction at an address and process ID for which a valid matching entry was not found in the TLB
8	Instruction storage — ZPR[Z*n*] = 00	Noncritical	Instruction translation is active, execution access to the translated address is not permitted because ZPR[Z*n*] = 00 in user mode, and an attempt is made to execute the instruction
9	Instruction storage — TLB_entry[EX] = 0	Noncritical	Instruction translation is active, execution access to the translated address is not permitted because TLB_entry[EX] = 0, and an attempt is made to execute the instruction
	Instruction storage — TLB_entry[G] = 1 or SGR[G*n*] = 1	Noncritical	Instruction translation is active, the page is marked guarded, and an attempt is made to execute the instruction
10	Program	Noncritical	Attempted execution of illegal instructions, TRAP instruction, privileged instruction in problem state
	System call	Noncritical	Execution of the **sc** instruction

Table 6-1. Interrupt Handling Priorities (Continued)

Priority	Interrupt Type	Critical or Noncritical	Causative Conditions
11	Data TLB miss	Noncritical	Valid matching entry for the effective address and process ID of an attempted data access is not found in the TLB
12	Data storage—ZPR[Z*n*] = 00	Noncritical	Data translation is active and data-side access to the translated address is not permitted because ZPR[Z*n*] = 00 in user mode
13	Data storage—TLB_entry[WR] = 0	Noncritical	Data translation is active and write access to the translated address is not permitted because TLB_entry[WR] = 0
	Data storage—TLB_entry[U0] = 1 or SU0R[U*n*] = 1	Noncritical	Data translation is active and write access to the translated address is not permitted because TLB_entry[U0] = 1 or SU0R[U*n*] = 1
14	Alignment	Noncritical	**dcbz** to non cacheable address or write-through storage; non-word aligned **dcread**, **lwarx**, and **stwcx**, as described in Table 6-10
15	Debug—BT, DAC, DVC, IC, TIE	Critical	BT, DAC, DVC, IC, TIE debug event (in internal debug mode)
16	External interrupt input	Noncritical	Active level on the external interrupt input by the UIC
17	Fixed Interval Timer (FIT)	Noncritical	Posting of an enabled FIT interrupt in the TSR
18	Programmable Interval Timer (PIT)	Noncritical	Posting of an enabled PIT interrupt in the TSR

6.4 Critical and Noncritical Interrupts

The PPC405 processes interrupts as noncritical and critical. The following interrupts are defined as *noncritical*: data storage, instruction storage, an active external interrupt input, alignment, program, system call, programmable interval timer (PIT), fixed interval timer (FIT), data TLB miss, and instruction TLB miss. The following interrupts are defined as critical: machine check interrupts (instruction- and data-side), debug interrupts, interrupts caused by an active critical interrupt input, and the first time-out from the watchdog timer.

When a *noncritical* interrupt is taken, Save/Restore Register 0 (SRR0) is written with the address of the excepting instruction (most synchronous interrupts) or the next sequential instruction to be processed (asynchronous interrupts and system call).

If the PPC405 was executing a multicycle instruction (multiply, divide, or cache operation), the instruction is terminated and its address is written in SRR0.

Aligned scalar loads/stores that are interrupted do not appear on the PLB. An aligned scalar load/store cannot be interrupted after it is requested on the PLB, so the Guarded (G) storage attribute does not need to prevent the interruption of an aligned scalar load/store.

To enhance performance, the DCU can respond to non cacheable load requests by retrieving a line instead of a word. This is controlled by CCR0[LWL]. Note, however, that If CCR0[LWL] = 1, and the target non cacheable region is also marked as guarded (the G storage attribute is set to 1), that the DCU will request on the PLB only those bytes requested by the CPU.

Load/store multiples, load/store string, and misaligned scalar loads/stores that cross a word boundary can be interrupted and restarted upon return from the interrupt handler.

When load instructions terminate, the addressing registers are not updated. This ensures that the instructions can be restarted; if the addressing registers were in the range of registers to be loaded, this would be an invalid form in any event. Some target registers of a load instruction may have been written by the time of the interrupt; when the instruction restarts, the registers will simply be written again. Similarly, some of the target memory of a store instruction may have been written, and is written again when the instruction restarts.

Save/Restore Register 1 (SRR1) is written with the contents of the MSR; the MSR is then updated to reflect the new machine context. The new MSR contents take effect beginning with the first instruction of the interrupt handling routine.

Interrupt handling routine instructions are fetched at an address determined by the interrupt type. The address of the interrupt handling routine is formed by concatenating the 16 high-order bits of the EVPR and the interrupt vector offset. (A user must initialize the EVPR contents at power-up using an mtspr instruction.)

Table 6-2 on page 113 shows the interrupt vector offsets for the interrupt types. Note that there can be multiple sources of the same interrupt type; interrupts of the same type are mapped to the same interrupt vector, regardless of source. In such cases, the interrupt handling routine must examine status registers to determine the exact source of the interrupt.

At the end of the interrupt handling routine, execution of an rfi instruction forces the contents of SRR0 and SRR1 to be written to the program counter and the MSR, respectively. Execution then begins at the address in the program counter.

Critical interrupts are processed similarly. When a critical interrupt is taken, Save/Restore Register 2 (SRR2) and Save/Restore Register 3 (SRR3) hold the next sequential address to be processed when returning from the interrupt, and the contents of the MSR, respectively. At the end of the critical interrupt handling routine, execution of an **rfci** instruction writes the contents of SRR2 and SRR3 into the program counter and the MSR, respectively.

Table 6-2. Interrupt Vector Offsets

Offset	Interrupt Type	Interrupt Class	Category	Page
0x0100	Critical input interrupt	Asynchronous precise	Critical	118
0x0200	Machine check—data	—	Critical	118
	Machine check—instruction	—	Critical	118
0x0300	Data storage interrupt— MSR[DR]=1 and ZPR[Z*n*] = 0 or TLB_entry[WR] = 0 or TLB_entry[U0] = 1 or SU0R[U*n*] = 1	Synchronous precise	Noncritical	120
0x0400	Instruction storage interrupt	Synchronous precise	Noncritical	121
0x0500	External interrupt (external to the processor core)	Asynchronous precise	Noncritical	122
0x0600	Alignment	Synchronous precise	Noncritical	123
0x0700	Program	Synchronous precise	Noncritical	123
0x0C00	System Call	Synchronous precise	Noncritical	124
0x1000	PIT	Asynchronous precise	Noncritical	125
0x1010	FIT	Asynchronous precise	Noncritical	125
0x1020	Watchdog timer	Asynchronous precise	Critical	126
0x1100	Data TLB miss	Synchronous precise	Noncritical	127
0x1200	Instruction TLB miss	Synchronous precise	Noncritical	127
0x2000	Debug—BT, DAC, DVC, IAC, IC, TIE	Synchronous precise	Critical	128
	Debug—EXC, UDE	Asynchronous precise	Critical	

6.5 General Interrupt Handling Registers

The general interrupt handling registers are the Machine State Register (MSR), SRR0–SRR3, the Exception Vector Prefix Register (EVPR), the Exception Syndrome Register (ESR), and the Data Exception Address Register (DEAR).

6.5.1 Machine State Register (MSR)

The MSR is a 32-bit register that holds the current context of the PPC405. When a noncritical interrupt is taken, the MSR contents are written to SRR1; when a critical interrupt is taken, the MSR contents are written to SRR3. When an **rfi** or **rfci** instruction executes, the contents of the MSR are read from SRR1 or SRR3, respectively.

Programming Note: The **rfi** and **rfci** instructions can alter reserved MSR fields.

The MSR contents can be read into a General Purpose Register (GPR) using an **mfmsr** instruction. The contents of a GPR can be written to the MSR using an **mtmsr** instruction. The MSR[EE] bit may be set/cleared atomically using the **wrtee** or **wrteei** instructions.

Figure 6-1. Machine State Register (MSR)

0:12		Reserved	
13	WE	Wait State Enable 0 The processor is not in the wait state. 1 The processor is in the wait state.	If MSR[WE] = 1, the processor remains in the wait state until an interrupt is taken, a reset occurs, or an external debug tool clears WE.
14	CE	Critical Interrupt Enable 0 Critical interrupts are disabled. 1 Critical interrupts are enabled.	Controls the critical interrupt input and watchdog timer first time-out interrupts.
15		Reserved	
16	EE	External Interrupt Enable 0 Asynchronous interrupts (external to the processor core) are disabled. 1 Asynchronous interrupts are enabled.	Controls the non-critical external interrupt input, PIT, and FIT interrupts.
17	PR	Problem State 0 Supervisor state (all instructions allowed). 1 Problem state (some instructions not allowed).	
18		Reserved	
19	ME	Machine Check Enable 0 Machine check interrupts are disabled. 1 Machine check interrupts are enabled.	
20		Reserved	
21	DWE	Debug Wait Enable 0 Debug wait mode is disabled. 1 Debug wait mode is enabled.	
22	DE	Debug Interrupts Enable 0 Debug interrupts are disabled. 1 Debug interrupts are enabled.	
23:25		Reserved	
26	IR	Instruction Relocate 0 Instruction address translation is disabled. 1 Instruction address translation is enabled.	

27	DR	Data Relocate 0 Data address translation is disabled. 1 Data address translation is enabled.	
28:31		Reserved	

6.5.2 Save/Restore Registers 0 and 1 (SRR0–SRR1)

SRR0 and SRR1 are 32-bit registers that hold the interrupted machine context when a noncritical interrupt is processed. On interrupt, SRR0 is set to the current or next instruction address and the contents of the MSR are written to SRR1. When an rfi instruction is executed at the end of the interrupt handler, the program counter and the MSR are restored from SRR0 and SRR1, respectively.

The contents of SRR0 and SRR1 can be written into GPRs using the **mfspr** instruction. The contents of GPRs can be written to SRR0 and SRR1 using the **mtspr** instruction.

Figure 6-2. Save/Restore Register 0 (SRR0)

0:29		SRR0 receives an instruction address when a non-critical interrupt is taken; the Program Counter is restored from SRR0 when the **rfi** instruction executes.	
30:31		Reserved	

Figure 6-3. Save/Restore Register 1 (SRR1)

0:31		SRR1 receives a copy of the MSR when an interrupt is taken; the MSR is restored from SRR1 when **rfi** executes.	

6.5.3 Save/Restore Registers 2 and 3 (SRR2–SRR3)

SRR2 and SRR3 are 32-bit registers that hold the interrupted machine context when a critical interrupt is processed. On interrupt, SRR2 is set to the current or next instruction address and the contents of the MSR are written to SRR3. When an **rfci** instruction is executed at the end of the interrupt handler, the program counter and the MSR are restored from SRR2 and SRR3, respectively.

The contents of SRR2 and SRR3 can be written to GPRs using the **mfspr** instruction. The contents of GPRs can be written to SRR2 and SRR3 using the **mtspr** instruction.

Figure 6-4. Save/Restore Register 2 (SRR2)

0:29		SRR2 receives an instruction address when a critical interrupt is taken; the Program Counter is restored from SRR2 when **rfci** executes.	
30:31		Reserved	

Figure 6-5. Save/Restore Register 3 (SRR3)			
0:31		SRR3 receives a copy of the MSR when a critical interrupt is taken; the MSR is restored from SRR3 when **rfci** executes.	

Because critical interrupts do not automatically clear MSR[ME], SRR2 and SRR3 can be corrupted by a machine check interrupt, if the machine check occurs while SRR2 and SRR3 contain valid data that has not yet been saved by the critical interrupt handler.

Because critical interrupts do not automatically clear MSR[ME], SRR2 and SRR3 can be corrupted by a machine check interrupt, if the machine check occurs while SRR2 and SRR3 contain valid data that has not yet been saved by the critical interrupt handler.

6.5.4 Exception Vector Prefix Register (EVPR)

The EVPR is a 32-bit register whose high-order 16 bits contain the prefix for the address of an interrupt handling routine. The 16-bit interrupt vector offsets (shown in *Table 6-2*) are concatenated to the right of the high-order 16 bits of the EVPR to form the 32-bit address of an interrupt handling routine.

The contents of the EVPR can be written to a GPR using the **mfspr** instruction. The contents of a GPR can be written to EVPR using the **mtspr** instruction.

Figure 6-6. Exception Vector Prefix Register (EVPR)			
0:15	EVP	Exception Vector Prefix	
16:31		Reserved	

6.5.5 Exception Syndrome Register (ESR)

The ESR is a 32-bit register whose bits help to specify the exact cause of various synchronous interrupts. These interrupts include instruction side machine checks, data storage interrupts, and program interrupts, instruction storage interrupts, and data TLB miss interrupts.

Instruction Machine Check Handling on page 119 describes instruction machine checks. *Data Storage Interrupt* on page 120 describes data storage interrupts. *Program Interrupt* on page 123 describes program interrupts.

Although interrupt handling routines are not required to reset the ESR, it is recommended that instruction machine check handlers reset the ESR; *Instruction Machine Check Handling* on page 119 describes why such resets are recommended.

The contents of the ESR can be written to a GPR using the **mfspr** instruction. The contents of a GPR can be written to the ESR using the **mtspr** instruction.

Figure 6-7. Exception Syndrome Register (ESR)			
0	MCI	Machine check—instruction 0 Instruction machine check did not occur. 1 Instruction machine check occurred.	
1:3		Reserved	

4	PIL	Program interrupt—illegal 0 Illegal Instruction error did not occur. 1 Illegal Instruction error occurred.	
5	PPR	Program interrupt—privileged 0 Privileged instruction error did not occur. 1 Privileged instruction error occurred.	
6	PTR	Program interrupt—trap 0 Trap with successful compare did not occur. 1 Trap with successful compare occurred.	
7		Reserved	
8	DST	Data storage interrupt—store fault 0 Excepting instruction was not a store. 1 Excepting instruction was a store (includes **dcbi**, **dcbz**, and **dccci**).	
9	DIZ	Data/instruction storage interrupt—zone fault 0 Excepting condition was not a zone fault. 1 Excepting condition was a zone fault.	
10:15		Reserved	
16	U0F	Data storage interrupt—U0 fault 0 Excepting instruction did not cause a U0 fault. 1 Excepting instruction did cause a U0 fault.	
17:31		Reserved	

In general, ESR bits are set to indicate the type of precise interrupt that occurred; other bits are cleared. However, the machine check—instruction (ESR[MCI]) bit behaves differently. Because instruction-side machine checks can occur without an interrupt being taken (if MSR[ME] = 0), ESR[MCI] can be set even while other ESR-setting interrupts (program, data storage, DTLB-miss) occurring. Thus, data storage and program interrupts leave ESR[MCI] unchanged, clear all other ESR bits, and set the bits associated with any data storage or program interrupts that occurred. Enabled instruction-side machine checks (MSR[ME] = 1) set ESR[MCI] and clear the data storage and program interrupt bits.

If a machine check—instruction interrupt occurs but is disabled (MSR[ME] = 0), it sets but leaves the data storage and program interrupt bits alone. If a machine check—instruction interrupt occurs while MSR[ME] = 0, and the instruction upon which the machine check—instruction interrupt is occurring also is some other kind of ESR-setting instruction (program, data storage, DTLB-miss, or instruction storage interrupt), ESR[MCI] is set to indicate that a machine check—instruction interrupt occurred; the other ESR bits are set or cleared to indicate the other interrupt. These scenarios are summarized in *Table 6-3*.

Table 6-3. ESR Alteration by Various Interrupts

Scenario	ECR[MCI]	$ESR_{4:}$	$ESR_{8:9,\ 16}$
Program interrupt	Unchanged	Set to type	Cleared
Data storage interrupt	Unchanged	Cleared	Set to Type
Data TLB miss interrupt	Unchanged	Cleared	Cleared
Machine check—instruction	Set to 1	Cleared	Cleared
Disabled MCI, no others	Unchanged	Unchanged	Unchanged
Disabled MCI and program interrupt	Unchanged	Set to type	Cleared

6.5.6 Data Exception Address Register (DEAR)

The DEAR is a 32-bit register that contains the address of the access for which one of the following synchronous precise errors occurred: alignment error, data TLB miss, or data storage interrupt. The contents of the DEAR can be written to a GPR using the mfspr instruction. The contents of a GPR can be written to the DEAR using the **mtspr** instruction.

Figure 6-8. Data Exception Address Register (DEAR)

0:31		Address of Data Error (synchronous)	

6.6 Critical Input Interrupts

The UICCR can be programmed so that any UIC interrupt can be presented as a critical interrupt input to the processor core. Critical interrupts are recognized only if enabled by MSR[CE].

MSR[CE] also enables the watchdog timer first-time-out interrupt. However, the watchdog interrupt has a different interrupt vector than the critical pin interrupt. See *Watchdog Timer Interrupt* on page 126.

After detecting a critical interrupt, if no synchronous precise interrupts are outstanding, the PPC405 immediately takes the critical interrupt and writes the address of the next instruction to be executed in SRR2. Simultaneously, the contents of the MSR are saved in SRR3. MSR[CE] is reset to 0 to prevent another critical interrupt or the watchdog timer first time-out interrupt from interrupting the critical interrupt handler before SRR2 and SRR3 get saved. MSR[DE] is reset to 0 to disable debug interrupts during the critical interrupt handler.

The MSR is also written with the values shown in *Table 6-4*. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0100. Interrupt processing begins at the address in the program counter.

Inside the interrupt handling routine, after the contents of SRR2/SRR3 are saved, critical interrupts can be enabled again by setting MSR[CE] = 1.

Executing an **rfci** instruction restores the program counter from SRR2 and the MSR from SRR3, and execution resumes at the address in the program counter.

Table 6-4. Register Settings during Critical Input Interrupts

SRR2	Written with the address of the next instruction to be executed
SRR3	Written with the contents of the MSR
PC	EVPR[0:15] || 0x0100

6.7 Machine Check Interrupts

When an external bus error occurs on an instruction fetch, and execution of that instruction is subsequently attempted, a machine check—instruction interrupt occurs.

When an external bus error occurs while attempting data accesses, a machine check—data interrupt occurs.

When an instruction-side machine check interrupt occurs, the PPC405 stores the address of the excepting instruction in SRR2. When a data-side machine check occurs, the PPC405 stores the address of the next sequential instruction in SRR2. Simultaneously, for all machine check interrupts, the contents of the MSR are loaded into SRR3.

The MSR Machine Check Enable bit (MSR[ME]) is reset to 0 to disable another machine check from interrupting the machine check interrupt handling routine. The other MSR bits are loaded with the values shown in *Table 6-5* and *Table 6-6* on page 120. The high-order 16 bits of the program counter are then written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x0200. Interrupt processing begins at the new address in the program counter.

Executing an **rfci** instruction restores the program counter from SRR2 and the MSR from SRR3, and execution resumes at the address in the program counter.

6.7.1 Instruction Machine Check Handling

When a machine check occurs on an instruction fetch, *and execution of that instruction is subsequently attempted*, a machine check—instruction interrupt occurs. If enabled by MSR[ME], the processor reports the machine check—instruction interrupt by vectoring to the machine check handler (EVPR[0:15] || 0x0200), setting. Note that only a bus error can cause a machine check—instruction interrupt. Taking the vector automatically clears MSR[ME] and the other MSR fields.

Note that it is improper to declare a machine check—instruction interrupt when the instruction is fetched, because the address is possibly the result of an incorrect speculation by the fetcher. It is quite likely that no attempt will be made to execute an instruction from the erroneous address. The interrupt will occur only if execution of the instruction is subsequently attempted.

When a machine check occurs on an instruction fetch, the erroneous instruction is never validated in the instruction cache unit (ICU). Fetch requests to cacheable memory that miss in the ICU cause an instruction cache line fill (eight words). If any words in the fetched line are associated with an error, an interrupt occurs upon attempted execution and the cache line is invalidated. If any word in the line is in error, the cache line is invalidated after the line fill.

is set, even if MSR[ME] = 0. This means that if a machine check—instruction interrupt occurs while running in code in which MSR[ME] is disabled, the machine check—instruction interrupt is recorded, but no interrupt occurs. Software running with MSR[ME] disabled can sample to determine whether at least one machine check—instruction interrupt occurred during the disabled execution.

If a new machine check—instruction interrupt occurs after MSR[ME] is enabled again, the new machine check—instruction interrupt is recorded, and the machine check—instruction interrupt handler is invoked. However, enabling MSR[ME] again does not cause a machine Check interrupt to occur simply due to the presence of indicating that a machine check—instruction interrupt occurred while MSR[ME] was disabled. The machine check—instruction interrupt must occur while MSR[ME] is enabled for the machine check interrupt to be taken. Software should, in general, clear the bits before returning from a machine check interrupt to avoid any ambiguity when handling subsequent machine check interrupts.

Table 6-5. Register Settings during Machine Check—Instruction Interrupts

SRR2	Written with the address that caused the machine check.
SRR3	Written with the contents of the MSR
PC	EVPR[0:15] || 0x0200
ESR	MCI ← 1. All other bits are cleared.

6.7.2 Data Machine Check Handling

When a machine check occurs on an data access, a machine check—data interrupt occurs. To determine the cause of a machine check, examine the various error reporting registers of the external PLB slaves.

Table 6-6. Register Settings during Machine Check—Data Interrupts

SRR2	Written with the address of the next sequential instruction.
SRR3	Written with the contents of the MSR
PC	EVPR[0:15] || 0x0200

6.8 Data Storage Interrupt

The data storage interrupt occurs when the desired access to the effective address is not permitted for any of the following reasons:

- A U0 fault: any store to an EA with the U0 storage attribute set and CCR0[U0XE] = 1
- In the problem state with data translation enabled:
  - A *zone fault*, which is any user-mode storage access (data load, store, **icbi, dcbz, dcbst,** or **dcbf**) with an effective address with (ZPR field) = 00. (**dcbt** and **dcbtst** will no-op in this situation, rather than cause an interrupt. The instructions **dcbi, dccci, icbt,** and **iccci**, being privileged, cannot cause zone fault data storage interrupts.)
  - Data store or **dcbz** to an effective address with the WR bit clear and (ZPR field) ¼ 11. (The privileged instructions **dcbi** and **dccci** are treated as “stores,” but will cause privileged program interrupts, rather than data storage interrupts.)
- In the supervisor state with data translation enabled:
  - Data store, **dcbi, dcbz,** or **dccci** to an effective address with the WR bit clear and (ZPR field) other than 11 or 10.

Programming Note: The icbi, icbt, and iccci instructions are treated as loads from the addressed byte with respect to address translation and protection. Instruction cache operations use MSR[DR], not MSR[IR], to determine translation of their operands. Instruction storage interrupts and Instruction-side TLB Miss Interrupts are associated with the fetching of instructions, not with the execution of instructions. Data storage interrupts and data TLB miss interrupts are associated with the execution of instruction cache operations.

When a data storage interrupt is detected, the PPC405 suppresses the instruction causing the interrupt and writes the instruction address in SRR0. The Data Exception Address Register (DEAR) is loaded with the data address that caused the access violation. ESR bits are loaded as shown in *Table 6-7* on page 121 to provide further information about the error. The current contents of the MSR are loaded into SRR1, and MSR bits are then loaded with the values shown in *Table 6-7* on page 121.

The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0300. Interrupt processing begins at the new address in the program counter. Executing the return from interrupt instruction (rfi) restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the PPC405 resumes execution at the new program counter address.

For instructions that can simultaneously generate program interrupts (privileged instructions executed in Problem State) and data storage interrupts, the program interrupt has priority.

Table 6-7. Register Settings during Data Storage Interrupts

SRR0	Written with the EA of the instruction causing the data storage interrupt
SRR1	Written with the value of the MSR at the time of the interrupt
PC	EVPR[0:15] || 0x0300
DEAR	Written with the EA of the failed access
ESR	DST ← 1 if excepting operation is a store DIZ ← 1 if access failure caused by a zone protection fault (ZPR[Z*n*] = 00 in user mode) U0F ← 1 if access failure caused by a U0 fault (the U0 storage attribute is set and CCR0[U0XE] = 1) MCI ← unchanged All other bits are cleared.

6.9 Instruction Storage Interrupt

The instruction storage interrupt is generated when instruction translation is active and execution is attempted for an instruction whose fetch access to the effective address is not permitted for any of the following reasons:

- In Problem State:
  - Instruction fetch from an effective address with (ZPR field) = 00.
  - Instruction fetch from an effective address with the EX bit clear and (ZPR field) ¼ 11.
  - Instruction fetch from an effective address contained within a Guarded region (G=1).
- In Supervisor State:
  - Instruction fetch from an effective address with the EX bit clear and (ZPR field) other than 11 or 10.
  - Instruction fetch from an effective address contained within a Guarded region (G=1).

SRR0 will save the address of the instruction causing the instruction storage interrupt.

ESR is set to indicate the following conditions:

- If ESR[DIZ] = 1, the excepting condition was a zone fault: the attempted execution of an instruction address fetched in user-mode with (ZPR field) = 00.
- If ESR[DIZ] = 0, then the excepting condition was either EX = 0 or G = 1.

The interrupt is precise with respect to the attempted execution of the instruction. Program flow vectors to EVPR[0:15] || 0x0400.

The following registers are modified to the specified values:

Table 6-8. Register Settings during Instruction Storage Interrupts

SRR0	Set to the EA of the instruction for which execute access was not permitted
SRR1	Set to the value of the MSR at the time of the interrupt
PC	EVPR[0:15] || 0x0400
ESR	DIZ ← 1 If access failure due to a zone protection fault (ZPR[Z*n*] = 00 in user mode) **Note:** If ESR[DIZ] is not set, the interrupt occurred because TBL_entry[EX] was clear in an otherwise accessible zone, or because of an instruction fetch from a storage region marked as guarded. See "Exception Syndrome Register (ESR)" on page 116 for details of ESR operation. MCI ← unchanged All other bits are cleared.

6.10 External Interrupt

External interrupts (external to the processor core) are triggered by active levels non-critical interrupts in the UIC. All external interrupting events are presented to the processor as a single external interrupt. External interrupts are enabled or disabled by MSR[EE].

> **Programming Note:** MSR[EE] also enables PIT and FIT interrupts. However, after timer interrupts, control passes to different interrupt vectors than for the interrupts discussed in the preceding paragraph. Therefore, these timer interrupts are described in *Programmable Interval Timer (PIT) Interrupt* on page 125 and *Fixed Interval Timer (FIT) Interrupt* on page 125.

6.10.1 External Interrupt Handling

When MSR[EE] = 1 (external interrupts are enabled), a noncritical external interrupt occurs, and this interrupt is the highest priority interrupt condition, the processor immediately writes the address of the next sequential instruction into SRR0. Simultaneously, the contents of the MSR are saved in SRR1.

When the processor takes a noncritical external interrupt, MSR[EE] is set to 0. This disables other external interrupts from interrupting the interrupt handler before SRR0 and SRR1 are saved. The MSR is also written with the other values shown in *Table 6-9*. The high-order 16 bits of the program counter are written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x0500. Interrupt processing begins at the address in the program counter.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter.

Table 6-9. Register Settings during External Interrupts

SRR0	Written with the address of the next sequential instruction
SRR1	Written with the contents of the MSR
PC	EVPR[0:15] || 0x0500

6.11 Alignment Interrupt

Alignment interrupts are caused by dcbz instructions to non cacheable or write-through storage and misaligned **dcread, lwarx,** or **stwx**. instructions. *Table 6-10* summarizes the instructions and conditions causing alignment interrupts.

Table 6-10. Alignment Interrupt Summary

Instructions Causing Alignment Interrupts	Conditions
dcbz	EA in non cacheable or write-through storage
dcread, **lwarx**, **stwcx**.	EA not word-aligned

Execution of an instruction causing an alignment interrupt is prohibited from completing. SRR0 is written with the address of that instruction and the current contents of the MSR are saved into SRR1. The DEAR is written with the address that caused the alignment error. The MSR bits are written with the values shown in *Table 6-11*. The high-order 16 bits of the program counter are written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x0600. Interrupt processing begins at the new address in the program counter.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter.

Alignment interrupts cannot be disabled. To avoid overwrites of SRR0 and SRR1 by alignment interrupts that occur within a handler, interrupt handlers should save these registers as soon as possible.

Table 6-11. Register Settings during Alignment Interrupts

SRR0	Written with the address of the instruction causing the alignment interrupt
SRR1	Written with the contents of the MSR
PC	EVPR[0:15] || 0x0600
DEAR	Written with the address that caused the alignment violation

6.12 Program Interrupt

Program interrupts are caused by attempting to execute:

- An illegal instruction
- A privileged instruction while in the problem state
- Executing a trap instruction with conditions satisfied

The ESR bits that differentiate these situations are listed and described in *Table 6-12*. When a program interrupt occurs, the appropriate bit is set and the others are cleared. These interrupts are not maskable.

Table 6-12. ESR Usage for Program Interrupts

Bits	Interrupts	Cause
ESR[PIL]	Illegal instruction	Opcode not recognized
ESR[PPR]	Privileged instruction	Attempt to use a privileged instruction in the problem state

Table 6-12. ESR Usage for Program Interrupts (Continued)

Bits	Interrupts	Cause
ESR[PTR]	Trap	Excepting instruction is a trap

The program interrupt handler does not need to reset the ESR.

When one of the following occurs, the PPC405 does not execute the instruction, but writes the address of the excepting instruction into SRR0:

- Attempted execution of a privileged instruction in problem state
- Attempted execution of an illegal instruction (including memory management instructions when memory management is disabled

Trap instructions can be used as a program interrupt or a debug event, or both (see *Debug Events* on page 147 for information about debug events). When a trap instruction is detected as a program interrupt, the PPC405 writes the address of the trap instruction into SRR0. See **tw** on page 341 and **twi** on page 344 (both in *Instruction Set* on page 157) for a detailed discussion of the behavior of trap instructions with various interrupts enabled.

After any program interrupt, the contents of the MSR ar MSR[APA] = 0, an attempt to execute an instruction intended for an APU causes a program interrupt if MSR[APE] = 0e written into SRR1 and the MSR bits are written with the values shown in *Table 6-13*. The high-order 16 bits of the program counter are written with the contents of the EVPR; the low-order 16 bits of the program counter are written with 0x0700. Interrupt processing begins at the new address in the program counter.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter.

Table 6-13. Register Settings during Program Interrupts

SRR0	Written with the address of the excepting instruction
SRR1	Written with the contents of the MSR
PC	EVPR[0:15] || 0x0700
ESR	Written with the type of program interrupt. (see *Table 6-12*) MCI ← unchanged All other bits are cleared.

6.13 System Call Interrupt

System call interrupts occur when a **sc** instruction is executed. The PPC405 writes the address of the instruction following the **sc** into SRR0. The contents of the MSR are written into SRR1 and the MSR bits are written with the values shown in *Table 6-14*. The high-order 16 bits of the program counter are then written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x0C00. Interrupt processing begins at the new address in the program counter.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter.

Table 6-14. Register Settings during System Call Interrupts

SRR0	Written with the address of the instruction following the **sc** instruction
SRR1	Written with the contents of the MSR
PC	EVPR[0:15] || 0x0C00

6.14 Programmable Interval Timer (PIT) Interrupt

For a discussion of the PPC405 timer facilities, see *Timer Facilities* on page 129. The PIT is described in *Programmable Interval Timer (PIT)* on page 131.

If the PIT interrupt is enabled by TCR[PIE] and MSR[EE], the PPC405 initiates a PIT interrupt after detecting a time-out from the PIT. Time-out is detected when, at the beginning of a clock cycle, TSR[PIS] = 1. (This occurs on the cycle after the PIT decrements on a PIT count of 1.) The PPC405 immediately takes the interrupt. The address of the next sequential instruction is saved in SRR0; simultaneously, the contents of the MSR are written into SRR1 and the MSR is written with the values shown in *Table 6-15*. The high-order 16 bits of the program counter are then written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x1000. Interrupt processing begins at the address in the program counter.

To clear a PIT interrupt, the interrupt handling routine must clear the PIT interrupt bit, TSR[PIS]. Clearing is performed by writing a word to TSR, using an **mtspr** instruction, that has 1 in bit positions to be cleared and 0 in all other bit positions. The data written to the TSR is not direct data, but a mask; a 1 clears the bit and 0 has no effect.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter.

Table 6-15. Register Settings during Programmable Interval Timer Interrupts

SRR0	Written with the address of the next instruction to be executed
SRR1	Written with the contents of the MSR
PC	EVPR[0:15] || 0x1000
TSR	PIS ← 1

6.15 Fixed Interval Timer (FIT) Interrupt

For a discussion of the PPC405 timer facilities, see *Timer Facilities* on page 129. The FIT is described in *Fixed Interval Timer (FIT)* on page 132.

If the FIT interrupt is enabled by TCR[FIE] and MSR[EE], the PPC405 initiates a FIT interrupt after detecting a time-out from the FIT. Time-out is detected when, at the beginning of a clock cycle, TSR[FIS] = 1. (This occurs on the second cycle after the 0 → 1 transition of the appropriate time-base bit.) The PPC405 immediately takes the interrupt. The address of the next sequential instruction is written into SRR0; simultaneously, the contents of the MSR are written into SRR1 and the MSR is written with the values shown in Table 6-16. The high-order 16 bits of the program counter are then written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x1010. Interrupt processing begins at the address in the program counter.

To clear a FIT interrupt, the interrupt handling routine must clear the FIT interrupt bit, TSR[FIS]. Clearing is performed by writing a word to TSR, using an **mtspr** instruction, that has 1 in any bit positions to be cleared and 0 in all other bit positions. The data written to the TSR is not direct data, but a mask; a 1 clears a bit and 0 has no effect.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter.

Table 6-16. Register Settings during Fixed Interval Timer Interrupts

SRR0	Written with the address of the next sequential instruction
SRR1	Written with the contents of the MSR
PC	EVPR[0:15] || 0x1010
TSR	FIS ← 1

6.16 Watchdog Timer Interrupt

For a general description of the PPC405 timer facilities, see *Timer Facilities* on page 129 The watchdog timer (WDT) is described in *Watchdog Timer* on page 133.

If the WDT interrupt is enabled by TCR[WIE] and MSR[CE], the PPC405 initiates a WDT interrupt after detecting the first WDT time-out. First time-out is detected when, at the beginning of a clock cycle, TSR[WIS] = 1. (This occurs on the second cycle after the 0→1 transition of the appropriate time-base bit while TSR[ENW] = 1 and TSR[WIS] = 0.) The PPC405 immediately takes the interrupt. The address of the next sequential instruction is saved in SRR2; simultaneously, the contents of the MSR are written into SRR3 and the MSR is written with the values shown in Table 6-17. The high-order 16 bits of the program counter are then written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x1020. Interrupt processing begins at the address in the program counter.

To clear the WDT interrupt, the interrupt handling routine must clear the WDT interrupt bit TSR[WIS]. Clearing is done by writing a word to TSR (using **mtspr**), with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The data written to the status register is not direct data, but a mask; a 1 causes the bit to be cleared, and a 0 has no effect.

Executing the return from critical interrupt instruction (**rfci**) restores the contents of the program counter and the MSR from SRR2 and SRR3, respectively, and the PPC405 resumes execution at the contents of the program counter.

Table 6-17. Register Settings during Watchdog Timer Interrupts

SRR2	Written with the address of the next sequential instruction
SRR3	Written with the contents of the MSR
PC	EVPR[0:15] || 0x1020
TSR	WIS ← 1

6.17 Data TLB Miss Interrupt

The data TLB miss interrupt is generated if data translation is enabled and a valid TLB entry matching the EA and PID is not present. The address of the instruction generating the untranslatable effective data address is saved in SRR0. In addition, the hardware also saves the data address (that missed in the TLB) in the DEAR. The ESR is set to indicate whether the excepting operation was a store (includes dcbz, dcbi, dccci). The interrupt is precise. Program flow vectors to EVPR[0:15] || 0x1100.

The following registers are modified to the values specified in Table 6-18.

Table 6-18. Register Settings during Data TLB Miss Interrupts

SRR0	Set to the address of the instruction generating the effective address for which no valid translation exists.
SRR1	Set to the value of the MSR at the time of the interrupt
PC	EVPR[0:15] || 0x1100
DEAR	Set to the effective address of the failed access
ESR	DST ← 1 if excepting operation is a store operation (includes **dcbi**, **dcbz**, and **dccci**). MCI ← unchanged All other bits are cleared.

Programming Note: Data TLB miss interrupts can happen whenever data translation is enabled. Therefore, ensure that SRR0 and SRR1 are saved before enabling translation in an interrupt handler.

6.18 Instruction TLB Miss Interrupt

The instruction TLB miss interrupt is generated if instruction translation is enabled and execution is attempted for an instruction for which a valid TLB entry matching the EA and PID for the instruction fetch is not present. The instruction whose fetch caused the TLB miss is saved in SRR0.

The interrupt is precise with respect to the attempted execution of the instruction. Program flow vectors to EVPR[0:15 || 0x1200.

The following are modified to the values specified in *Table 6-19*.

Table 6-19. Register Settings during Instruction TLB Miss Interrupts

SRR0	Set to the address of the instruction for which no valid translation exists.
SRR1	Set to the value of the MSR at the time of the interrupt
PC	EVPR[0:15] || 0x1200

Programming Note: Instruction TLB miss interrupts can happen whenever instruction translation is active. Therefore, insure that SRR0 and SRR1 are saved before enabling translation in an interrupt handler.

6.19 Debug Interrupt

Debug interrupts can be either synchronous or asynchronous. These debug events generate synchronous interrupts: branch taken (BT), data address compare (DAC), data value compare (DVC), instruction address compare (IAC), instruction completion (IC), and trap instruction (TIE). The exception (EXC) and unconditional (UDE) debug events generate asynchronous interrupts. See *Debug Events* on page 147 for more information about debug events.

For debug events, SRR2 is written with an address, which varies with the type of debug event, as shown in *Table 6-20*.

Table 6-20. SRR2 during Debug Interrupts

Debug Event	Address Saved in SRR2
BT DAC IAC TIE	Address of the instruction causing the event
DVC IC	Address of the instruction *following* the instruction that causing the event
EXC	Interrupt vector address of the initial exception that caused the exception debug event
UDE	Address of next instruction to be executed at time of UDE

SRR3 is written with the contents of the MSR and the MSR is written with the values shown in *Table 6-21*. The high-order 16 bits of the program counter are then written with the contents of the EVPR; the low-order 16 bits of the program counter are written with 0x2000. Interrupt processing begins at the address in the program counter.

Executing an **rfci** instruction restores the program counter from SRR2 and the MSR from SRR3, and execution resumes at the address in the program counter.

Table 6-21. Register Settings during Debug Interrupts

SRR2	Written with an address as described in *Table 6-20*
SRR3	Written with the contents of the MSR
PC	EVPR[0:15] || 0x2000
DBSR	Set to indicate type of debug event.

7. Timer Facilities

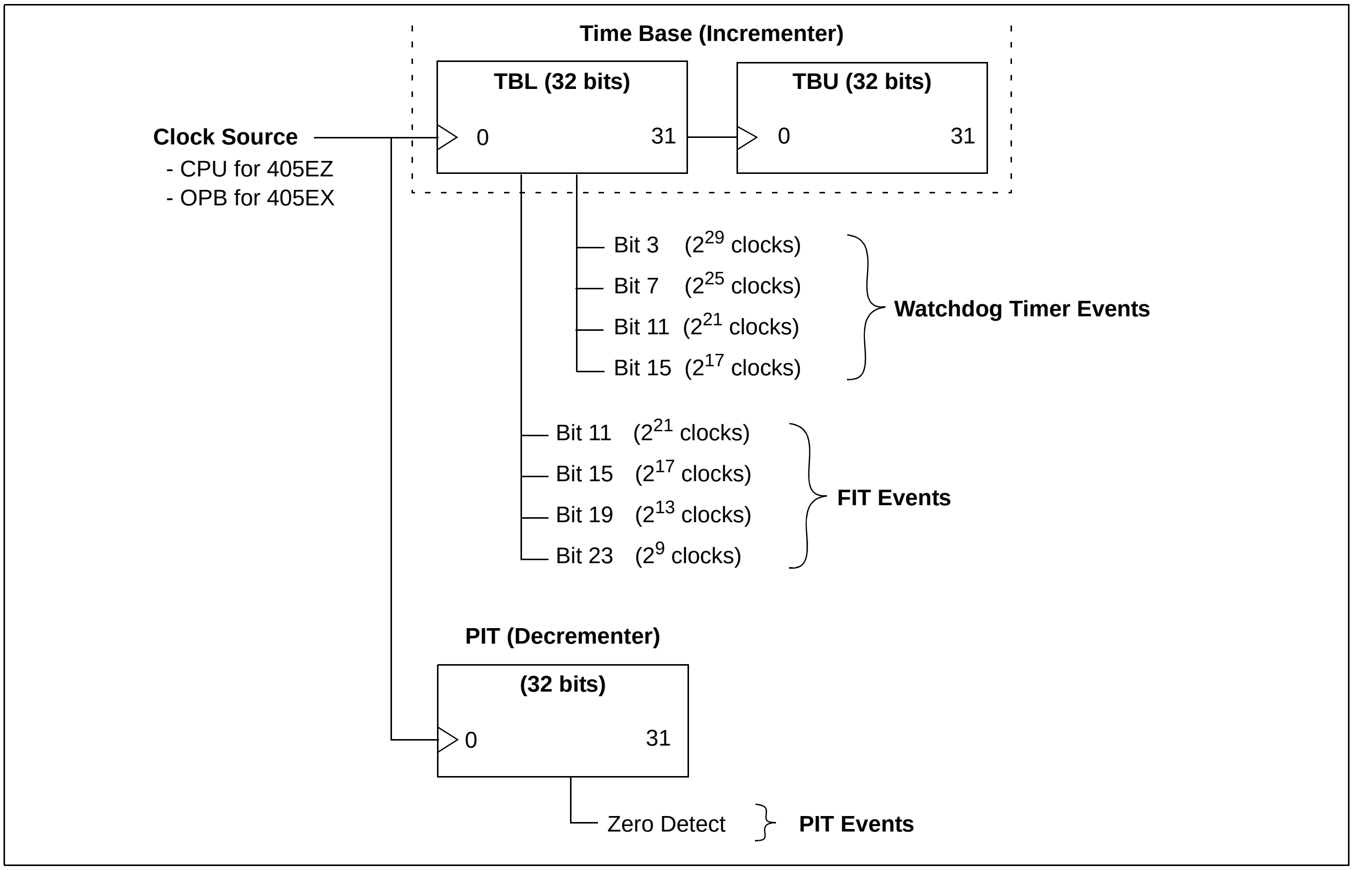
The PPC405 processor core provides four timer facilities: a time base, a Programmable Interval Timer (PIT), a fixed interval timer (FIT), and a watchdog timer. The PIT is a Special Purpose Register (SPR). These facilities, which are driven by the same base clock, can, among other things, be used for:

- Time-of-day functions
- Data logging functions
- Peripherals requiring periodic service
- Periodic task switching

Additionally, the watchdog timer can help a system to recover from faulty hardware or software.

Figure 7-1 shows the relationship of the timers and the clock source to the time base.

Figure 7-1. Relationship of Timer Facilities to the Time Base



7.1 Time Base

The PPC405 implements a 64-bit time base as required in *The PowerPC Architecture*. The time base, which increments once during each period of the source clock, provides a time reference.

Read access to the time base is through the **mftb** instruction. **mftb** provides user-mode read-only access to the time base. The TBR numbers (0x10C and 0x10D; TBL and TBU, respectively) that specify the time base registers to **mftb** are not SPR numbers. However, the PowerPC Architecture allows an implementation to handle **mftb** as **mfspr**. Accordingly, these register numbers cannot be used for other SPRs. PowerPC compilers cannot use **mftb** with register numbers other than those specified in the PowerPC Architecture as read-access time base registers (0x10C and 0x10D).

Write access to the time base, using **mtspr**, is privileged. Different register numbers are used for read access and write access. Writing the time base is accomplished by using SPR 0x11C and SPR 0x11D (TBL and TBU, respectively) as operands for **mtspr**.

The period of the 64-bit time base is approximately 2925 years for a 200 MHz clock source. The time base does not generate interrupts, even when it wraps. For most applications, the time base is set once at system reset and only read thereafter. Note that the FIT and the watchdog timer (discussed below) are driven by 0→1 transitions of bits from the TBL. Transitions caused by software alteration of TBL have the same effect as transitions caused by normal incrementing of the time base. *Figure 7-2* illustrates the TBL.

Figure 7-2. Time Base Lower (TBL)

0:31		Time Base Lower	Current count; low-order 32 bits of time base.

Figure 7-3 illustrates the TBU.

Figure 7-3. Time Base Upper (TBU)

0:31		Time Base Upper	Current count, high-order 32 bits of time base.

Table 7-1 summarizes the TBRs, instructions used to access the TBRs, and access restrictions.

Table 7-1. Time Base Access

Instructions		Register Number	Access Restrictions
• TBU • Upper 32 bits	• mftbu RT • Extended mnemonic for • mftb RT,TBU	• 0x10D	• Read-only
	• mttbu RS • Extended mnemonic for • mtspr TBU,RS	• 0x11D	• Privileged; write-only
• TBL • Lower 32 bits	• mftb RT • Extended mnemonic for • mftb RT,TBL	• 0x10C	• Read-only
	• mttbl RS • Extended mnemonic for • mtspr TBL,RS	• 0x11C	• Privileged; write-only

7.1.1 Reading the Time Base

The following code provides an example of reading the time base. **mftb** moves the low-order 32 bits of the time base to a GPR; **mftbu** moves the high-order 32 bits of the time base to a second GPR.

```
loop:
    mftbu   Rx          # load from TBU
    mftb    Ry          # load from TBL
    mftbu   Rz          # load from TBU
    cmpw    Rz, Rx      # see if old = new
    bne     loop        # loop/reread if rollover occurred
```

The comparison and loop ensure that a consistent pair of values is obtained.

7.1.2 Writing the Time Base

The following code provides an example of writing the time base. Writing the time base is privileged. **mttbl** moves the contents of a GPR to the low-order 32 bits of the time base; **mttbu** moves the contents of a second GPR to the high-order 32 bits of the time base.

```
    lwz     Rx, upper   # load 64-bit time base value into Rx and Ry
    lwz     Ry, lower
    li      Rz, 0
    mttbl   Rz          # force TBL to 0 to avoid rollover while writing TBU
    mttbu   Rx          # set TBU
    mttbl   Ry          # set TBL
```

7.2 Programmable Interval Timer (PIT)

The PIT is a 32-bit SPR that decrements at the same rate as the time base. The PIT is read and written using **mfspr** and **mtspr**, respectively. Writing to the PIT also simultaneously writes to a hidden reload register. Reading the PIT using **mfspr** returns the current PIT contents; the hidden reload register cannot be read. When a non-zero value is written to the PIT, it begins to decrement. A PIT event occurs when a decrement occurs and the PIT count is set to 1. When a PIT event occurs, the following occurs:

1. If the PIT is in auto-reload mode (the ARE field of the Timer Control Register (TCR) is 1), the PIT is loaded with the last value an **mtspr** wrote to the PIT. A decrement from a PIT count of 1 immediately causes a reload; no intermediate PIT content of 0 occurs.
   If the PIT is not in auto-reload mode (TCR[ARE] = 0), a decrement from a PIT count of 1 simply causes a PIT content of 0.
2. TSR[PIS] is set to 1.
3. If enabled (TCR[PIE] = 1 and the EE field of the Machine State Register (MSR) is 1), a PIT interrupt is taken. See "Programmable Interval Timer (PIT) Interrupt" on page 10-44 for details of register behavior during a PIT interrupt.

The interrupt handler should use software to reset the PIS field of the Timer Status Register (TSR). This is done by using **mtspr** to write a word to the TSR having a 1 in TSR[PIS] and any other bits to be cleared, and a 0 in all other bits. The data written to the TSR is not direct data, but a mask. A 1 clears a bit; a 0 has no effect.

Using **mtspr** to force the PIT to 0 does not cause a PIT interrupt. However, decrementing that was ongoing at the instant of the **mtspr** instruction can cause the appearance of an interrupt. To eliminate the PIT as a source of interrupts, write a 0 to TCR[PIE], the PIT interrupt enable bit.

To eliminate all PIT activity:

1. Write a 0 to TCR[PIE]. This prevents PIT activity from causing interrupts.
2. Write a 0 to TCR[ARE]. This disables the PIT auto-reload feature.
3. Write zeroes to the PIT to halt PIT decrementing. Although this action does not cause a pit PIT interrupt to become pending, a near-simultaneous decrement to 0 might have done so.
4. Write a 1 to TSR[PIS] (PIT Interrupt Status bit). This clears TSR[PIS] to 0 (see “Timer Status Register (TSR)” on page 11-8). This also clears any pending PIT interrupt. Because the PIT stops decrementing, no further PIT events are possible.

If the auto-reload feature is disabled (TCR[ARE] = 0) when the PIT decrements to 0, the PIT remains 0 until software uses **mtspr** to reload it.

After a reset, TCR[ARE] = 0, which disables the auto-reload feature. *Figure 7-4* illustrates the PIT.

Figure 7-4. Programmable Interval Timer (PIT)

0:31		Programmed interval remaining	Number of clocks remaining until the PIT event

7.2.1 Fixed Interval Timer (FIT)

The FIT provides timer interrupts having a repeatable period. The FIT is functionally similar to an auto-reload PIT, except that only a smaller fixed selection of interrupt periods are available.

The FIT exception occurs on 0→1 transitions of selected bits from the time base, as shown in *Table 7-2*.

Table 7-2. FIT Controls

TCR[FP]	TBL Bit	Period (Time Base Clocks)	Period (200 Mhz Clock)
0, 0	23	2^9 clocks	2.56 μsec
0, 1	19	2^{13} clocks	40.96 μsec
1, 0	15	2^{17} clocks	0.655 msec
1, 1	11	2^{21} clocks	10.49 msec

The TSR[FIS] field logs a FIT exception as a pending interrupt. A FIT interrupt occurs if TCR[FIE] and MSR[EE] are enabled at the time of the FIT exception. “Fixed Interval Timer (FIT) Interrupt” on page 10-44 describes register settings during a FIT interrupt.

The interrupt handler should reset TSR[FIS]. This is done by using **mtspr** to write a word to the TSR having a 1 in TSR[FIS] and any other bits to be cleared, and a 0 in all other bits. The data written to the TSR is not direct data, but a mask. A 1 clears a bit and a 0 has no effect.

7.3 Watchdog Timer

The watchdog timer aids system recovery from software or hardware faults.

A watchdog timeout occurs on 0→1 transitions of a selected bit from the time base, as shown in *Table 7-3*.

Table 7-3. Watchdog Timer Controls

TCR[WP]	TBL Bit	Period (Time Base Clocks)	Period (200 MHz Clock)
0,0	15	2^{17} clocks	0.655 msec
0,1	11	2^{21} clocks	10.49 msec
1,0	7	2^{25} clocks	0.168 sec
1,1	3	2^{29} clocks	2.684 sec

If a watchdog timeout occurs while TSR[WIS] = 0 and TSR[ENW] = 1, a watchdog interrupt occurs if the interrupt is enabled by TCR[WIE] and MSR[CE]. “Watchdog Timer” on page 11-6 describes register behavior during a watchdog interrupt.

The interrupt handler should reset the TSR[WIS] bit. This is done by using **mtspr** to write a word to the TSR having a 1 in TSR[WIS] and any other bits to be cleared, and a 0 in all other bits. The data written to the TSR is not direct data, but a mask. A 1 clears a bit and a 0 has no effect.

If a watchdog timeout occurs while TSR[WIS] = 1 and TSR[ENW] = 1, a hardware reset occurs if enabled by a non-zero value of TCR[WRC]. In other words, a reset can occur if a watchdog timeout occurs while a previous watchdog timeout is pending. The assumption is that TSR[WIS] was not cleared because the processor could not execute the watchdog handler, leaving reset as the only way to restart the system. Note that after TCR[WRC] is set to a non-zero value, it cannot be reset by software. This prevents errant software from disabling the watchdog timer reset capability. After a reset, the initial value of TCR[WRC] = 00.

Figure 7-5 illustrates the watchdog state machine. The values shown for ENW and WIS relate to the actions described in *Figure 7-4* and the operating mode descriptions that follow *Figure 7-4*.

Figure 7-5. Watchdog State Machine

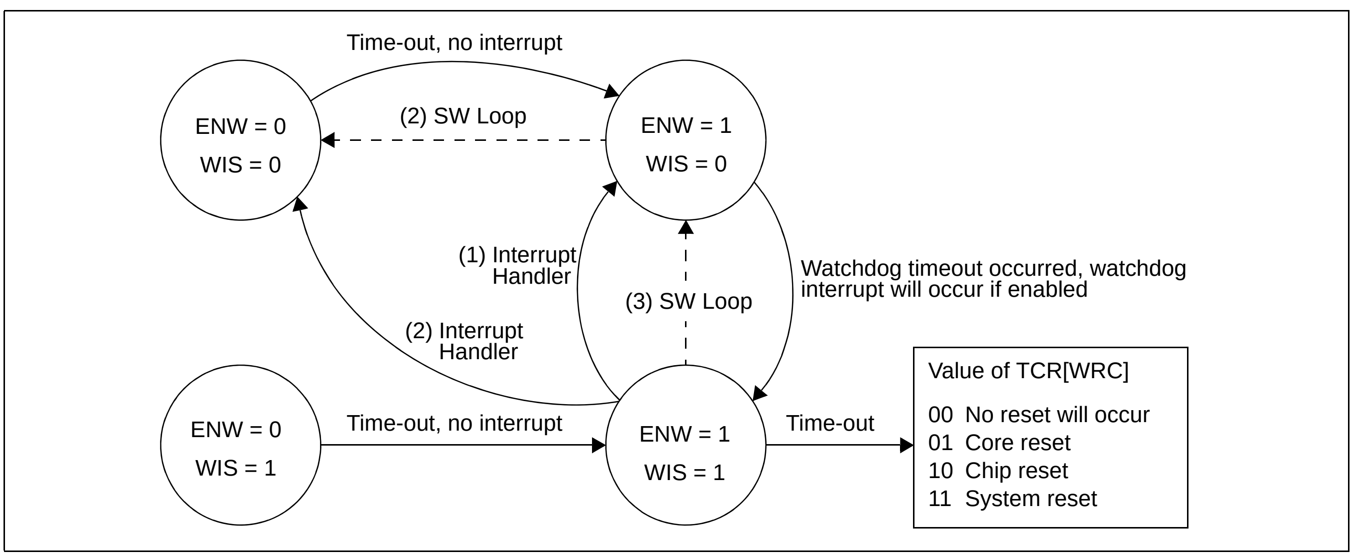


Table 7-4. Watchdog Timer State Machine

Enable Next Watchdog TSR[ENW]	Watchdog Timer Status TSR[WIS]	Action When Timer Interval Expires
0	0	Set TSR[ENW] = 1.
0	1	Set TSR[ENW] = 1.
1	0	Set TSR[WIS] = 1. If TCR[WIE] = 1 and MSR[CE] = 1, then interrupt.
1	1	Cause the watchdog reset action specified by TCR[WRC]. On reset, copy current TCR[WRC] to TSR[WRS] and clear TCR[WRC], disabling the watchdog timer.

The controls described in Figure 7-4 imply three different modes of using the watchdog timer. The modes assume that TCR[WRC] was set to allow processor reset by the watchdog timer:

1. Always take a pending watchdog interrupt, and never attempt to prevent its occurrence. (This mode is described in the preceding text.)
   a. Clear TSR[WIS] in the watchdog timer handler.
   b. Never use TSR[ENW].
2. Always take a pending watchdog interrupt, but avoid it whenever possible by delaying a reset until a second watchdog timer occurs.

   This assumes that a recurring code loop of known maximum duration exists outside the interrupt handlers, or that a FIT interrupt handler is operational. One of these mechanisms clears TSR[ENW] more frequently than the watchdog period.
   a. Clear TSR[ENW] to 0 in loop or in FIT interrupt handler.
   To clear TSR[ENW], use mtspr to write a 1 to TSR[ENW] (and to any other bits that are to be cleared), with 0 in all other bit locations.
   b. Clear TSR[WIS] in watchdog timer handler.

   It is not expected that a watchdog interrupt will occur every time, but only if an exceptionally high execution load delays clearing of TSR[ENW] in the usual time frame.
3. Never take a watchdog interrupt.

   This assumes that a recurring code loop of reliable duration exists outside the interrupt handlers, or that a FIT interrupt handler is operational. This method only guarantees one watchdog timeout period before a reset occurs.
   a. Clear TSR[WIS] in the loop or in FIT handler.
   b. Never use TSR[ENW] but have it set.

7.4 Timer Status Register (TSR)

The TSR can be accessed for read or write-to-clear.

Status registers are generally set by hardware and read and cleared by software. The mfspr instruction reads the TSR. Clearing the TSR is performed by writing a word to the TSR, using mtspr, having a 1 in all fields to be cleared and a 0 in all other fields. The data written to the TSR is not direct data, but a mask. A 1 clears the field and a 0 has no effect.

Figure 7-6. Timer Status Register (TSR)

0	ENW	Enable Next Watchdog 0 Action on next watchdog event is to set TSR[ENW] = 1. 1 Action on next watchdog event is governed by TSR[WIS].	Software must reset TSR[ENW] = 0 after each watchdog timer event.
1	WIS	Watchdog Interrupt Status 0 No Watchdog interrupt is pending. 1 Watchdog interrupt is pending.	
2:3	WRS	Watchdog Reset Status 00 No Watchdog reset has occurred. 01 Core reset was forced by the watchdog. 10 Chip reset was forced by the watchdog. 11 System reset was forced by the watchdog.	
4	PIS	PIT Interrupt Status 0 No PIT interrupt is pending. 1 PIT interrupt is pending.	
5	FIS	FIT Interrupt Status 0 No FIT interrupt is pending. 1 FIT interrupt is pending.	
6:31		Reserved	

7.5 Timer Control Register (TCR)

The TCR controls PIT, FIT, and watchdog timer operation.

The TCR[WRC] field is cleared to 0 by all processor resets. This field is set only by software. However, hardware does not allow software to clear the field after it is set. After software writes a 1 to a bit in the field, that bit remains a 1 until any reset occurs. This prevents errant code from disabling the watchdog timer reset function.

All processor resets clear TCR[ARE] to 0, disabling the auto-reload feature of the PIT.

Figure 7-7. Timer Control Register (TCR)

0:1	WP	Watchdog Period 00 2^{17} clocks 01 2^{21} clocks 10 2^{25} clocks 11 2^{29} clocks	
2:3	WRC	Watchdog Reset Control 00 No Watchdog reset will occur. 01 Core reset will be forced by the Watchdog. 10 Chip reset will be forced by the Watchdog. 11 System reset will be forced by the Watchdog.	TCR[WRC] resets to 00. This field can be set by software, but cannot be cleared by software, except by a software-induced reset.
4	WIE	Watchdog Interrupt Enable 0 Disable watchdog interrupt. 1 Enable watchdog interrupt.	
5	PIE	PIT Interrupt Enable 0 Disable PIT interrupt. 1 Enable PIT interrupt.	
6:7	FP	FIT Period 00 2^{9} clocks 01 2^{13} clocks 10 2^{17} clocks 11 2^{21} clocks	
8	FIE	FIT Interrupt Enable 0 Disable FIT interrupt. 1 Enable FIT interrupt.	
9	ARE	Auto Reload Enable 0 Disable auto reload. 1 Enable auto reload.	Disables on reset.
10:31		Reserved	

8. Debugging

The debug facilities of the PPC405 include support for debug modes for debugging during hardware and software development, and debug events that allow developers to control the debug process. Debug registers control the debug modes and debug events. The debug registers are accessed through software running on the processor or through a JTAG debug port. The debug interface is the JTAG debug port. The JTAG debug port can also be used for board test.

The debug modes, events, controls, and interface provide a powerful combination of debug facilities for a wide range of hardware and software development tools.

8.1 Development Tool Support

The RISCWatch™ product is an example of a development tool that uses the external debug mode, debug events, and the JTAG debug port to implement a hardware and software development tool. The RISCTrace™ feature of RISCWatch is an example of a development tool that uses the real-time instruction trace capability of the PPC405.

8.2 Debug Interfaces

The PPC405 provides JTAG and trace interfaces to support hardware and software test and debug. Typically, the JTAG interface connects to a debug port external to the PPC405; the debug port is typically connected to a JTAG connector on a processor board.

The trace interface connects to a trace port, also external to the PPC405, that is typically connected to a trace connector on the processor board.

8.3 IEEE 1149.1 Test Access Port (JTAG Debug Port)

The IEEE 1149.1 Test Access Port (TAP), commonly called the JTAG (Joint Test Action Group) debug port, is an architectural standard described in IEEE Std 1149.1–1990, *IEEE Standard Test Access Port and Boundary Scan Architecture*. The standard describes a method for accessing internal chip facilities using a four- or five-signal interface.

The JTAG debug port, originally designed to support scan-based board testing, is enhanced to support the attachment of debug tools. The enhancements, which are designed to the IEEE 1149.1 specifications for vendor-specific extensions, are compatible with standard JTAG hardware for boundary-scan system testing.

JTAG Signals	The JTAG debug port implements the four required JTAG signals: TCK, TMS, TDI, and TDO, and the optional $\overline{TRST}$ signal.
JTAG Clock Requirements	The frequency of the TCK signal can range from DC to one-half of the internal chip clock frequency.
JTAG Reset Requirements	The JTAG debug port logic is reset at the same time as a system reset. Upon receiving $\overline{TRST}$, the JTAG debug port returns to the Test-Logic Reset state.

8.3.1 JTAG Connector

A 16-pin male 2x8 header connector is suggested as the JTAG debug port connector. This connector definition matches the requirements of the RISCWatch debugger. The connector is described in detail in *RISCWatch Debugger User's Guide*.

8.3.2 JTAG Instructions

The JTAG debug port provides the standard *extest*, *idcode*, *sample/preload*, and *bypass* instructions and the optional *highz* and *clamp* instructions. Invalid instructions behave as the *bypass* instruction.

Table 8-1. JTAG Instructions

Instruction	Code	Comments
Extest	1111000	IEEE 1149.1 standard.
	1111001	Reserved.
Sample/Preload	1111010	IEEE 1149.1 standard.
IDCode	1111011	IEEE 1149.1 standard.
Private	xxxx100	Private instructions
HighZ	1111101	IEEE 1149.1a-1993 optional
Clamp	1111110	IEEE 1149.1a-1993 optional
Bypass	1111111	IEEE 1149.1 standard.

8.3.3 JTAG Boundary Scan

Boundary Scan Description Language (BSDL), IEEE 1149.1b-1994, is a supplement to IEEE 1149.1-1990 and IEEE 1149.1a-1993 *Standard Test Access Port and Boundary-Scan Architecture*. BSDL, a subset of the IEEE 1076-1993 Standard VHSIC Hardware Description Language (VHDL), allows a rigorous description of testability features in components which comply with the standard. BSDL is used by automated test pattern generation tools for package interconnect tests and by electronic design automation (EDA) tools for synthesized test logic and verification. BSDL supports robust extensions that can be used for internal test generation and to write software for hardware debug and diagnostics.

The primary components of BSDL include the logical port description, the physical pin map, the instruction set, and the boundary register description.

The logical port description assigns symbolic names to the pins of a chip. Each pin has a logical type of in, out, inout, buffer, or linkage that defines the logical direction of signal flow.

The physical pin map correlates the logical ports of the chip to the physical pins of a specific package. A BSDL description can have several physical pin maps; each map is given a unique name.

Instruction set statements describe the bit patterns that must be shifted into the Instruction Register to place the chip in the various test modes defined by the standard. Instruction set statements also support descriptions of instructions that are unique to the chip.

The boundary register description lists each cell or shift stage of the Boundary Register. Each cell has a unique number: the cell numbered 0 is the closest to the Test Data Out (TDO) pin; the cell with the highest number is closest to the Test Data In (TDI) pin. Each cell contains additional information, including: cell type, logical port associated with the cell, logical function of the cell, safe value, control cell number, disable value, and result value.

8.3.4 JTAG Implementation

PPC405 JTAG interface I/Os (TDI, TDO, TMs, TCK, and $\overline{\text{TRST}}$) are 5V tolerant and do not contain internal pull up resistors.

The optional JTAG instructions, *idcode* and *highz*, offer additional JTAG functionality. The *idcode* instruction returns the PPC405 JTAG ID, which is unique for each chip version. The *highz* instruction disables all chip outputs regardless of whether they are included in the JTAG boundary scan chain.

The PPC405 provides boundary scan structures on all digital I/O signals.

PPC405 boundary scan structures are defined as follows:

1. All digital pins labeled in the IOSpeclist as functional inputs are observe only.
2. All digital pins labeled as outputs are drive only and are always actively driven during JTAG except when the HIGHZ command is selected on the JTAG TAP controller.
3. All digital pins labeled as 3-state ouputs or bidirectional drive when explicitly enabled by means of the appropriate boundary scan cell. They are forced to a disabled state in the presence of the HIGHZ command. When the driver is disabled, the input state of a bidirectional signal can be observed.
4. Analog pins are not observable.

8.3.5 JTAG ID Register

In most cases, there is a register that enables manufacturing, part number, and version information to be determined through the TAP. The **mfdcr** instruction is used to read this register.

Refer to data sheet for the chip in question to see the value assigned to the JTAG ID.

8.4 Trace Port

The PPC405 implements a trace status interface to support the tracing of code running in real-time. This interface enables the connection of an external trace tool, such as RISCWatch, and allows for user-extended trace functions. A software tool with trace capability, such as RISCWatch with RISCTrace, can use the data collected from this port to trace code running on the processor. The result is a trace of the code executed, including code executed out of the instruction cache if it was enabled. Information on trace capabilities, how trace works, and how to connect the external trace tool is available in *RISCWatch Debugger User's Guide*.

8.5 Debug Modes

The PPC405 supports the following debug modes, each of which supports a type of debug tool or debug task commonly used in embedded systems development:

- Internal debug mode, which supports ROM monitors
- External debug mode, which supports JTAG debuggers
- Debug wait mode, which supports processor stopping or stepping for JTAG debuggers while servicing interrupts
- Real-time trace mode, which supports trigger events for real-time tracing

Internal and external debug modes can be enabled simultaneously. Both modes are controlled by fields in Debug Control Register 0 (DBCR0). Real-time trace mode is available only if internal, external, and debug wait modes are disabled.

8.5.1 Internal Debug Mode

Internal debug mode provides access to architected processor resources and supports setting hardware and software breakpoints and monitoring processor status. In this mode, debug events generate debug interrupts, which can interrupt normal program flow so that monitor software can collect processor status and alter processor resources.

Internal debug mode relies on exception handling software at a dedicated interrupt vector and an external communications path to debug software problems. This mode, used while the processor executes instructions, enables debugging of operating system or application programs.

In this mode, debugger software is accessed through a communications port, such as a serial port, external to the processor core.

To enable internal debug mode, the Debug Control Register 0 (DBCR0) field IDM is set to 1 (DBCR0[IDM] = 1). To enable debug interrupts, MSR[DE] = 1. A debug interrupt occurs on a debug event only if DBCR0[IDM] = 1 and MSR[DE] = 1.

8.5.2 External Debug Mode

External debug mode provides access to architected processor resources and supports stopping, starting, and stepping the processor, setting hardware and software breakpoints, and monitoring processor status. In this mode, debug events cause the processor to become architecturally frozen. While the processor is frozen, normal instruction execution stops and architected processor resources can be accessed and altered. External bus activity continues in external debug mode.

The JTAG mechanism can pass instructions to the processor for execution, allowing a JTAG debugger to display and alter processor resources, including memory.

The JTAG mechanism prevents the occurrence of a privileged exception when a privileged instruction is executed while the processor is in user mode.

Storage access control by a memory management unit (MMU) remains in effect while in external debug mode; the debugger may need to modify MSR or TLB values to access protected memory.

Because external debug mode relies only on internal processor resources, it can be used to debug system hardware and software.

In this mode, access to the processor is through the JTAG debug port.

To enable external debug mode, DBCR0[EDM] = 1. To enable debug interrupts, MSR[DE] = 1. A debug interrupt occurs on a debug event only if DBCR0[EDM] = 1 and MSR[DE] = 1.

8.5.3 Debug Wait Mode

In debug wait mode, debug events cause the PPC405 to enter a state in which interrupts can be serviced while the processor appears to be stopped.

Debug wait mode provides access to architected processor resources in a manner similar to external debug mode, except that debug wait mode allows the servicing of interrupt handlers. It supports stopping, starting, and stepping the processor, setting hardware and software breakpoints, and monitoring processor status. In this mode, if a debug event caused the processor to become architecturally frozen, an interrupt causes the processor to run an interrupt handler and return to the architecturally frozen state upon returning from the interrupt handler. While the processor is frozen, normal instruction execution stops and architected processor resources can be accessed and altered. External bus activity continues in debug wait mode.

The processor enters debug wait mode when internal and external debug modes are disabled (DBCR0[IDM, EDM] = 0), debug wait mode is enabled (MSR[DWE] = 1), debug wait is enabled by the JTAG debugger, and a debug event occurs.

For example, while the PPC405 is in debug wait mode, an external device might generate an interrupt that requires immediate service. The PPC405 can service the interrupt (vector to an interrupt handler and execute the interrupt handler code) and return to the previous stopped state.

Debug wait mode relies only on internal processor resources, so it can be used to debug both system hardware and software problems. This mode can also be used for software development on systems without a control program, or to debug control program problems.

In this mode, access to the processor is through the JTAG debug port.

8.5.4 Real-time Trace Debug Mode

Real-time trace debug mode supports the generation of trigger events for tracing the instruction stream being executed out of the instruction cache in real-time. In this mode, debug events can be used to control the collection of trace information through the use of trigger event generation. The broadcast of trace information is independent of the use of debug events as trigger events.This mode does not alter the processor performance.

A trace event occurs when internal and external debug modes are disabled (DBCR0[IDM, EDM] = 0) and a debug events occurs.

When a trace event occurs, a trace device can capture trace signals that provide the instruction trace information. Most trace events generated from debug events are blocked when internal debug, external debug, or debug wait modes are enabled

8.6 Processor Control

The PPC405 provides the following debug functions for processor control. Not all facilities are available in all debug modes.

Instruction Step	The processor is stepped one instruction at a time, while stopped, using the JTAG debug port.
Instruction Stuff	While the processor is stopped, instructions can be stuffed into the processor and executed using the JTAG debug port.
Halt	The processor can be stopped by activating an external halt signal on an external event, such as a logic analyzer trigger. This signal freezes the processor architecturally. While frozen, normal instruction execution stops and architected processor resources can be accessed and altered using the JTAG debug port. Normal execution resumes when the halt signal is deactivated.
Stop	The processor can be stopped using the JTAG debug port. Activating a stop causes the processor to become architecturally frozen. While frozen, normal instruction execution stops and the architected processor resources can be accessed and altered using the JTAG debug port.
Reset	An external reset signal, the JTAG debug port, or DBCR0 can request core, chip, and system resets.
Debug Events	A debug event triggers a debug operation. The operation depends on the debug mode. For more information and a list of debug events, see “Debug Events” on page 147.
Freeze Timers	The JTAG debug port or DBCR0 can control timer resources. The timers can be enabled to run, freeze always, or freeze on a debug event.
Trap Instructions	The trap instructions **tw** and **twi** can be used, with debug events, to implement software breakpoints.

8.7 Processor Status

The processor execution status, exception status, and most recent reset can be monitored.

Execution Status	The JTAG debug port can monitor processor execution status to determine whether the processor is stopped, waiting, or running.
Exception Status	The JTAG debug port can monitor the status of pending synchronous exceptions.
Most Recent Reset	The JTAG debug port or an **mfspr** instruction can be used to read the Debug Status Register (DBSR) to determine the type of the most recent reset.

8.8 Debug Registers

Several debug registers, available to debug tools running on the processor, are not intended for use by application code. Debug tools control debug resources such as debug events. Application code that uses debug resources can cause the debug tools to fail, as well as other unexpected results, such as program hangs and processor resets.

Application code should not use the debug resources, including the debug registers.

8.8.1 Debug Control Registers

The debug control registers (DBCR0 and DBCR1)can enable and configure debug events, reset the processor, control timer operation during debug events, enable debug interrupts, and set the processor debug mode.

8.8.1.1 Debug Control Register 0 (DBCR0)

Figure 8-1. Debug Control Register 0 (DBCR0)

0	EDM	External Debug Mode 0 Disabled 1 Enabled	
1	IDM	Internal Debug Mode 0 Disabled 1 Enabled	
2:3	RST	Reset 00 No action 01 Core reset 10 Chip reset 11 System reset	Causes a processor reset request when set by software. **Attention:** Writing 01, 10, or 11 to this field causes a processor reset request.
4	IC	Instruction Completion Debug Event 0 Disabled 1 Enabled	
5	BT	Branch Taken Debug Event 0 Disabled 1 Enabled	
6	EDE	Exception Debug Event 0 Disabled 1 Enabled	
7	TDE	Trap Debug Event 0 Disabled 1 Enabled	
8	IA1	IAC 1 Debug Event 0 Disabled 1 Enabled	
9	IA2	IAC 2 Debug Event 0 Disabled 1 Enabled	
10	IA12	Instruction Address Range Compare 1–2 0 Disabled 1 Enabled	Registers IAC1 and IAC2 define an address range used for IAC address comparisons.
11	IA12X	Enable Instruction Address Exclusive Range Compare 1–2 0 Inclusive 1 Exclusive	Selects the range defined by IAC1 and IAC2 to be inclusive or exclusive.
12	IA3	IAC 3 Debug Event 0 Disabled 1 Enabled	
13	IA4	IAC 4 Debug Event 0 Disabled 1 Enabled	
14	IA34	Instruction Address Range Compare 3–4: 0 Disabled 1 Enabled	Registers IAC3 and IAC4 define an address range used for IAC address comparisons.

15	IA34X	Instruction Address Exclusive Range Compare 3–4: 0 Inclusive 1 Exclusive	Selects range defined by IAC3 and IAC4 to be inclusive or exclusive.
16	IA12T	Instruction Address Range Compare 1-2 Toggle: 0 Disabled 1 Enable	Toggles range 12 inclusive, exclusive DBCR[IA12X] on debug event.
17	IA34T	Instruction Address Range Compare 3–4 Toggle: 0 Disabled 1 Enable	Toggles range 34 inclusive, exclusive DBCR[IA34X] on debug event.
18:30		Reserved	
31	FT	Freeze Timers on Debug Event: 0 Timers not frozen 1 Timers frozen	

8.8.1.2 Debug Control Register 1 (DBCR1)

Figure 8-2. Debug Control Register 1 (DBCR1)

0	D1R	DAC1 Read Debug Event: 0 Disabled 1 Enabled	
1	D2R	DAC 2 Read Debug Event: 0 Disabled 1 Enabled	
2	D1W	DAC 1 Write Debug Event: 0 Disabled 1 Enabled	
3	D2W	DAC 2 Write Debug Event: 0 Disabled 1 Enabled	
4:5	D1S	DAC 1 Size: 00 Compare all bits 01 Ignore lsb (least significant bit) 10 Ignore two lsbs 11 Ignore five lsbs	Address bits used in the compare: Byte address Halfword address Word address Cache line (8-word) address
6:7	D2S	DAC 2 Size: 00 Compare all bits 01 Ignore lsb (least significant bit) 10 Ignore two lsbs 11 Ignore five lsbs	Address bits used in the compare: Byte address Halfword address Word address Cache line (8-word) address
8	DA12	Enable Data Address Range Compare 1:2: 0 Disabled 1 Enabled	Registers DAC1 and DAC2 define an address range used for DAC address comparisons
9	DA12X	Data Address Exclusive Range Compare 1:2: 0 Inclusive 1 Exclusive	Selects range defined by DAC1 and DAC2 to be inclusive or exclusive
10:11		Reserved	

12:13	DV1M	Data Value Compare 1 Mode: 00 Undefined 01 AND 10 OR 11 AND-OR	Type of data comparison used: All bytes selected by DBCR1[DV1BE] must compare to the appropriate bytes of DVC1. One of the bytes selected by DBCR1[DV1BE] must compare to the appropriate bytes of DVC1. The upper halfword or lower halfword must compare to the appropriate halfword in DVC1. When performing halfword compares set DBCR1[DV1BE] = 0011, 1100, or 1111.
14:15	DV2M	Data Value Compare 2 Mode: 00 Undefined 01 AND 10 OR 11 AND-OR	Type of data comparison used All bytes selected by DBCR1[DV2BE] must compare to the appropriate bytes of DVC2. One of the bytes selected by DBCR1[DV2BE] must compare to the appropriate bytes of DVC2. The upper halfword or lower halfword must compare to the appropriate halfword in DVC2. When performing halfword compares set DBCR1[DV2BE] = 0011, 1100, or 1111.
16:19	DV1BE	Data Value Compare 1 Byte: 0 Disabled 1 Enabled	Selects which data bytes to use in data value comparison
20:23	DV2BE	Data Value Compare 2 Byte: 0 Disabled 1 Enabled	Selects which data bytes to use in data value comparison
24:31		Reserved	

8.8.2 Debug Status Register (DBSR)

The DBSR contains status on debug events and the most recent reset; the status is obtained by reading the DBSR. The status bits are normally set by debug events or by any of the three reset types.

Clearing DBSR fields is performed by writing a word to the DBSR, using the **mtdbsr** extended mnemonic, having a 1 in all bit positions to be cleared and a 0 in the all other bit positions. The data written to the DBSR is not direct data, but a mask. A 1 clears the bit and a 0 has no effect.

Application code must not use the DBSR.

Figure 8-3. Debug Status Register (DBSR)

0	IC	Instruction Completion Debug Event: 0 Event did not occur 1 Event occurred	
1	BT	Branch Taken Debug Event: 0 Event did not occur 1 Event occurred	
2	EDE	Exception Debug Event: 0 Event did not occur 1 Event occurred	

3	TIE	Trap Instruction Debug Event: 0 Event did not occur 1 Event occurred	
4	UDE	Unconditional Debug Event: 0 Event did not occur 1 Event occurred	
5	IA1	IAC1 Debug Event: 0 Event did not occur 1 Event occurred	
6	IA2	IAC2 Debug Event: 0 Event did not occur 1 Event occurred	
7	DR1	DAC1 Read Debug Event: 0 Event did not occur 1 Event occurred	
8	DW1	DAC1 Write Debug Event: 0 Event did not occur 1 Event occurred	
9	DR2	DAC2 Read Debug Event: 0 Event did not occur 1 Event occurred	
10	DW2	DAC2 Write Debug Event: 0 Event did not occur 1 Event occurred	
11	IDE	Imprecise Debug Event: 0 No circumstance that would cause a debug event (if MSR[DE] = 1) occurred 1 A debug event would have occurred, but debug exceptions were disabled (MSR[DE] = 0)	
12	IA3	IAC3 Debug Event: 0 Event did not occur 1 Event occurred	
13	IA4	IAC4 Debug Event: 0 Event did not occur 1 Event occurred	
14:21		Reserved	
22:23	MRR	Most Recent Reset: No reset has occurred since last cleared by software. 0 Core reset 1 Chip reset System reset	This field is set to a value, indicating the type of reset, when a reset occurs.
24:31		Reserved	

8.8.3 Instruction Address Compare Registers (IAC1–IAC4)

The PPC405 can take a debug event upon an attempt to execute an instruction from an address. The address, which must be word-aligned, is defined in an IAC register. The DBCR0[IA1, IA2] fields of DBCR0 controls the instruction address compare (IAC) debug event.

Figure 8-4. Instruction Address Compare Registers (IAC1–IAC4)

0:29		Instruction Address Compare Word Address	Omit two low-order bits of complete address.
30:31		Reserved	

8.8.4 Data Address Compare Registers (DAC1–DAC2)

The PPC405 can take a debug event upon storage or cache references to addresses specified in the DAC registers. The specified addresses in the DAC registers are EAs of operands of storage references or cache instructions.The fields DBCR1[D1R], [D2R] and DBCR[D1W], [D2W] control the DAC-read and DAC-write debug events, respectively.

Addresses in the DAC registers specify exact byte EAs for DAC debug events. However, one may want to take a debug event on any byte within a halfword (ignore the least significant bit (LSb) of the DAC), on any byte within a word (ignore the two LSbs of DAC), or on any byte within eight words (ignore four LSbs of DAC). DBCR1[D1S, D2S] control the addressing options.

Errors related to execution of storage reference or cache instructions prevent DAC debug events.

Figure 8-5. Data Address Compare Registers (DAC1–DAC2)

0:31		Data Address Compare (DAC) Byte Address	DBCR0[D1S] determines which address bits are examined.

8.8.5 Data Value Compare Registers (DVC1–DVC2)

The PPC405 can take a debug event upon storage or cache references to addresses specified in the DAC registers, that also require the data at that address to match the value specified in the DVC registers. The data address compare for a DVC events works the same as for a DAC event. Cache operations do not cause DVC events. If the data at the address specified matches the value in the corresponding DVC register a DVC event will occur. The fields DBCR1[DV1M, DV2M] control how the data value are compared.

Errors related to execution of storage reference or cache instructions prevent DVC debug events.

Figure 8-6. Data Value Compare Registers (DVC1–DVC2)

0:31		Data Value to Compare	

8.8.6 Debug Events

Debug events, enabled and configured by DBCR0 and DBCR1 and recorded in the DBSR, cause debug operations. A debug event occurs when an event listed in *Table 8-2* on page 148 is detected. The debug operation is performed after the debug event.

In internal debug mode, the processor generates a debug interrupt when a debug event occurs. In external debug mode, the processor stops when a debug event occurs. When internal and external debug mode are both enabled, the processor stops on a debug event with the debug interrupt pending. When external and internal debug mode are both disabled, and debug wait mode is enabled the processor stops, but can be restarted by an interrupt. When all debug modes are disabled, debug events are recorded in the DBSR, but no action is taken.

Table 8-2 lists the debug events and the related fields in DBCR0, DBCR1, and DBSR. DBCR0 and DBCR1 enable the debugs events, and the DBSR fields report their occurrence.

Table 8-2. Debug Events

Event	Enabling DBCR0, DBCR1 Fields	Reporting DBSR Fields	Description
Instruction Completion	IC	IC	Occurs after completion of an instruction.
Branch Taken	BT	BT	Occurs before execution of a branch instruction determined to be taken.
Exception Taken	EDE	EXC	Occurs after an exception.
Trap Instruction	TDE	TIE	Occurs before execution of a trap instruction where the conditions are such that the trap will occur.
Unconditional	UDE	UDE	Occurs immediately upon being set by the JTAG debug port.
Instruction Address Compare	IA1, IA2, IA3, IA4, IA12, IA12X, IA12T, IA34, IA34X, IA34T	IA1, IA2, IA3, IA4	Occurs before execution of an instruction at an address that matches an address defined by the Instruction Address Compare Registers (IAC1–IAC4).
Data Address Compare	D1R, D1W, D1S, D2R, D2W, D2S, DA12, DA12X	DR2,DW2	Occurs before execution of an instruction that accesses a data address that matches the contents of the specified DAC register.
Data Value Compare	DV1M, DV2M, DV1BE, DV2BE	DR1, DW1	Occurs after execution of an instruction that accesses a data address for which a DAC occurs, and for which the value at the address matches the value in the specified DVC register.
Imprecise		IDE	Indicates that another debug event occurred while MSR[DE] = 0

8.8.7 Instruction Complete Debug Event

This debug event occurs after the completion of an instruction. If DBCR0[IDM] = 1, DBCR0[EDM] = 0 and MSR[DE] =0 this debug event is disabled.

8.8.8 Branch Taken Debug Event

This debug event occurs before execution of a branch instruction determined to be taken. If DBCR0[IDM] = 1, DBCR0[EDM] = 0 and MSR[DE] =0 this debug event is disabled.

8.8.9 Exception Taken Debug Event

This debug event occurs after an exception. Exception debug events always include the non-critical class of exceptions. When DBCR0[IDM] = 1 and DBCR0[EDM] = 0 the critical exceptions are not included.

8.8.10 Trap Taken Debug Event

This debug event occurs before execution of a trap instruction where the conditions are such that the trap will occur. When trap is enabled for a debug event, external debug mode is enabled, internal debug mode is enabled with MSR[DE] enabled, or debug wait mode is enabled, a trap instruction will not cause a program exception.

8.8.11 Unconditional Debug Event

This debug event occurs immediately upon being set by the JTAG debug port.

8.8.12 IAC Debug Event

This debug event occurs before execution of an instruction at an address that matches an address defined by the Instruction Address Compare Registers (IAC1–IAC4). DBCR0[IA1, IA2, IA3, IA4] enable IAC debug events IAC can be defined as an exact address comparison to one of the IAC*n* registers or on a range of addresses to compare defined by a pair of IAC*n* registers.

8.8.12.1 IAC Exact Address Compare

In this mode each IAC*n* register specifies an exact address to compare. These are enabled by setting DBCR0[IA*n*] = 1 and disabling IAC range compare (DBCR0[IA12X] = 0 for IAC1 and IAC2 and DBCR0[IA23X] = 0 for IAC3 and IAC4). The corresponding DBSR[IA*n*] bit displays the results of the debug event.

8.8.12.2 IAC Range Address Compare

In this mode a pair of IAC*n* registers are used to define a range of addresses to compare:

Range 1:2 corresponds to IAC1 and IAC2
Range 3:4 corresponds to IAC3 and IAC4

To enable Range 1:2, DBCR0[IA12] = 1 and DBCR0[IA1] or DBCR0[IA2] =1. An IAC event will be seen in the DBSR[IA*n*] field that corresponds to the enabled DBCR0[IA*n*] field. If DBCR0[IA1] and DBCR0[IA2] are enabled, the results of the event are reported in both DBSR fields. Setting DBCR0[IA12] =1 prohibits IAC1 and IAC2 from being used for exact address compares.

To enable Range 3:4, DBCR0[IA34] = 1 and DBCR0[IA3] or DBCR0[IA4] =1. An IAC event will be seen in the DBSR[IA*n*] field that corresponds to the enabled DBCR0[IA*n*] field. If DBCR0[IA3] and DBCR0[IA4] are enabled, the results of the event will be reported in both DBSR fields. Setting DBCR0[IA34] =1 prohibits IAC3 and IAC4 from being used for exact address compares.

Ranges can be defined as inclusive, as shown in the preceding examples, or exclusive, using DBCR0[IA12X] (corresponding to range 1:2) and DBCR0[IA34X] (corresponding to range 3:4), as follows:

DBCR0[IA12] = 1: Range 1:2 = IAC1 ≤ range < IAC2.
DBCR0[IA12X] = 1: Range 1:2 = Range low < IAC1 or IAC2 ð≤ Range high
DBCR0[IA34] = 1: Range 3:4 = IAC3 ≤ range < IAC4.
DBCR0[IA34X] = 1: Range 3:4 = Range low < IAC3 or IAC4 ≤ Range high

Figure 8-7 shows the range selected in an inclusive IAC range address compare. Note that the address in IAC1 is considered part of the range, but the address in IAC2 is not, as shown in the preceding examples. The thick lines indicate that the indicated address is included in the compare results.

Figure 8-7. Inclusive IAC Range Address Compares

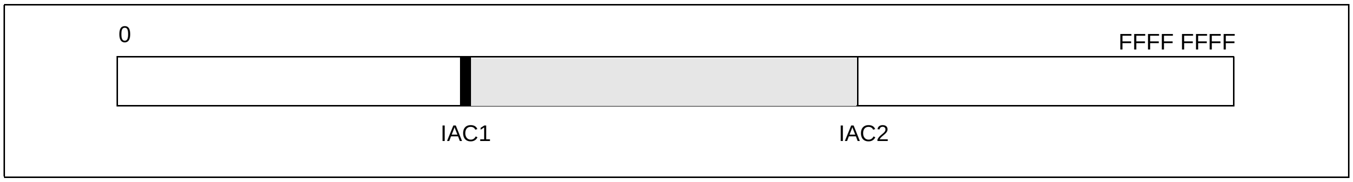
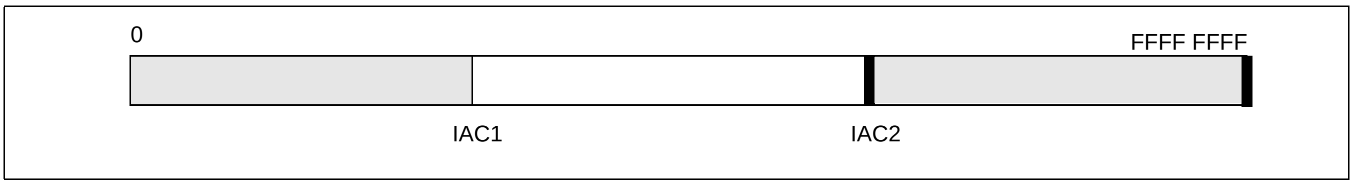


Figure 8-8 shows the range selected in an inclusive IAC range address compare. Note that the address in IAC1 is not considered part of the range, but the address in IAC2 is, along with the highest memory address, as shown in the preceding examples.

Figure 8-8. Exclusive IAC Range Address Compares



To toggle the range from inclusive to exclusive or from exclusive to inclusive on a IAC range debug event, DBCR0[IA12T] (corresponding to range 1:2) and DBCR0[IA34T] (corresponding to range 3:4) are used. If these fields are set, the DBCR0[IA12X] or DBCR0[IA34X] fields toggle on an IAC debug event, changing the defined range.

If a toggle is enabled (DBCR0[IA12T] for range 1:2 or DBCR0[IA34T] = 1 for range 3:4), and DBCR0[IDM] =1, DBCR0[EDM] = 0, and MSR[DE] = 0, IAC range comparisons for the corresponding toggle field are disabled.

8.8.13 DAC Debug Event

This debug event occurs before execution of an instruction that accesses a data address that matches the contents of the specified DAC register. DBCR1[D1R, D2R, D1W, D2W] enable DAC debug events for address comparisons on DAC1 and DAC2 for read instructions, DAC2 for read instructions, DAC1 for write instructions, DAC2 for write instructions respectively. Loads are reads and stores are writes. DAC can be defined(DBCR1[D1R, D2R])as an exact address comparison to one of the DACn registers or a range of addresses to compare defined by DAC1 and DAC2 registers.

8.8.13.1 DAC Exact Address Compare

In this mode, each DAC*n* register specifies an exact address to compare. These registers are enabled by setting one or more of DBCR1[D1R,D2R,D1W,D2W] = 1, and disabling DAC range compare DBCR1[DA12X] = 0. The corresponding DBSR[DR1,DR2,DW1,DW2] field displays the results of a DAC debug event.

The address for a DAC is the effective address (EA) of a storage reference instruction. EAs are always generated within a single aligned word of memory. Unaligned load and store, strings, and multiples generate multiple EAs to be used in DAC comparisons.

Data address compare (DAC) debug events can be set to react to any byte in a larger block of memory, in addition to reacting to a byte address match. The DAC Compare Size fields (DBCR1[D1S, D2S]) allow DAC debug events to react to byte, halfword, word, or 8-word line address by ignoring a number of LSBs in the EA.

DAC 1 Size	
00 Compare all bits	Byte address
01 Ignore LSB (least significant bit)	Halfword address
10 Ignore two LSBs	Word address
11 Ignore five LSBs	Cache line (8-word) address

The user must determine how the addresses of interest are accessed, relative to byte, halfword, word, string, and unaligned storage instructions, and adjust the DAC compare size field appropriately to cover the addresses of interest.

For example, suppose that a DAC debug event should react to byte 3 of a word-aligned target. A DAC set for exact compare would not recognize a reference to that byte by load/store word or load/store halfword instructions, because the byte address is not the EA of such instructions. In such a case, the D1S field must be set for a wider capture range (for example, to ignore the two least significant bits (LSBs) if word operations to the misaligned byte are to be detected). The wider capture range may result in excess debug events (events that are within the specified capture range, but reflect byte operations in addition to the desired byte). Such excess debug events must be handled by software.

While load/store string instructions are inherently byte addressed the processor will generate EAs containing the largest portion of an aligned word address as possible. It may not be possible to DAC on a specific individual byte using load/store string instructions.

8.8.13.2 DAC Range Address Compare

In this mode, the pair of DAC1 and DAC2 registers are used to define a range of addresses to compare.

To enable DAC range, DBCR1[DA12] = 1 and one or more of DBCR1[D1R,D2R,D1W,D2W] =1. The DAC event is seen on the DBSR[DR1,DR2,DW1,DW2] field that corresponds to the DBCR1[D1R,D2R,D1W,D2W] field that is enabled. For example, if DBCR1[D1R] and DBCR1[D2R] are enabled, the results of a DAC debug event are reported on DBSR[DR1, DR2]. Setting DBCR1[DA12] =1 prohibits DAC1 and DAC2 from being used for exact address compares.

Ranges are defined to be inclusive or exclusive, using the DBCR1[DA12X], as follows:

DBCR1[DA12] = 1: Range = DAC1 ≤ range < DAC2.
DBCR1[DA12X] = 1: Range = Range low < DAC1 or DAC2 ≤ Range high.

Figure 8-9 shows the range selected in an inclusive DAC range address compare. Note that the address in DAC1 is considered part of the range, but the address in DAC2 is not, as shown in the preceding examples. The thick lines indicate that the indicated address is included in the compare results.

Figure 8-9. Inclusive DAC Range Address Compares

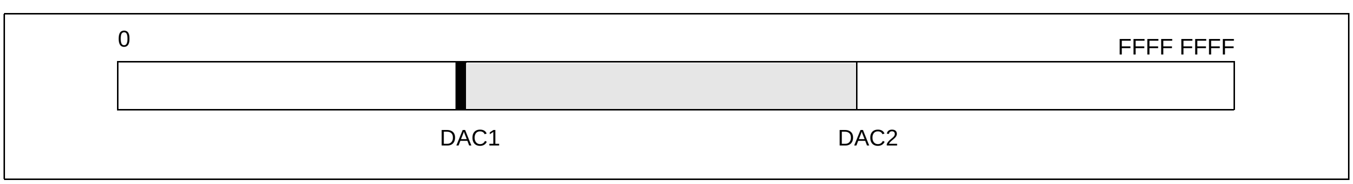
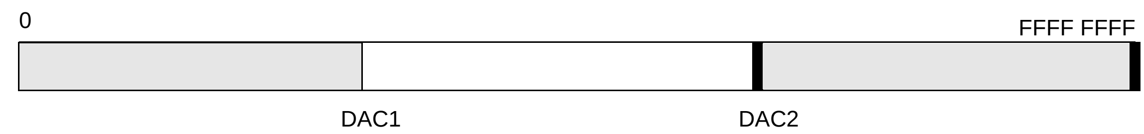


Figure 8-10 shows the range selected in an exclusive DAC range address compare. Note that the address in DAC1 is not considered part of the range, but the address in DAC2 is, along with the highest memory address, as shown in the preceding examples.

Figure 8-10. Exclusive DAC Range Address Compares



The DAC Compare Size fields (DBCR1[D1S, D2S]) are not used by DAC range comparisons.

8.8.13.3 DAC Applied to Cache Instructions

Some cache instructions can cause DAC debug events. There are several special cases.

Table 8-3 summarizes possible DAC debug events by cache instruction:

Table 8-3. DAC Applied to Cache Instructions

Instruction	Possible DAC Debug Event	
	DAC-Read	DAC-Write
dcba	No	Yes
dcbf	No	Yes
dcbi	No	Yes
dcbst	No	Yes
dcbt	Yes	No
dcbz	No	Yes
dccci	No	No
dcread	No	No
dcbtst	Yes	No
icbi	Yes	No
icbt	Yes	No
iccci	No	No
icread	No	No

Architecturally, the **dcbi** and **dcbz** instructions are "stores." These instructions can change data, or cause the loss of data by invalidating a dirty line. Therefore, they can cause DAC-write debug events.

The **dccci** instruction can also be considered a "store" because it can change data by invalidating a dirty line. However, **dccci** is not address-specific; it affects an entire congruence class regardless of the operand address of the instruction. Because it is not address-specific, **dccci** does not cause DAC-write debug events.

Architecturally, the **dcbt**, **dcbtst**, **dcbf**, and **dcbst** instructions are "loads." These instructions do not change data. Flushing or storing a cache line from the cache is not architecturally a "store" because a store had already updated the cache; the **dcbf** or **dcbst** instruction only updates the copy in main memory.

The **dcbt** and **dcbtst** instructions can cause DAC-read debug events regardless of cachability.

Although **dcbf** and **dcbst** are architecturally “loads,” these instructions can create DAC-write (but not DAC-read) debug events. In a debug environment, the fact that external memory is being written is the event of interest.

Even though **dcread** and **dccci** are not address-specific (they affect a congruence class regardless of the instruction operand address), and are considered “loads,” in the PPC405 they do not cause DAC debug events.

All ICU operations (**icbi**, **icbt**, **iccci**, and **icread**) are architecturally treated as “loads.” **icbi** and **icbt** cause DAC debug events. **iccci** and **icread** do not cause DAC debug events in the PPC405.

8.8.13.4 DAC Applied to String Instructions

An **stswx** instruction with a string length of 0 is a no-op. The **lswx** instruction with the string length equal to 0 does not alter the RT operand with undefined data, as allowed by the PowerPC Architecture. Neither **stswx** nor **lswx** with zero length causes a DAC debug event because storage is not accessed by these instructions.

8.8.14 Data Value Compare Debug Event

A data value compare (DVC) debug event can occur only after execution of a load or store instruction to an address that compares with the address in one of the DAC*n* registers and has a data value that matches the corresponding DVC*n* register. Therefore, a DVC debug event requires both the data address comparison and the data value comparison to be true. A DVCn debug event when enabled in the DBCR1 supersedes a DACn debug event since the DVCn and the DACn both use the same DACn register.

DVC1 debug events are enabled by setting the appropriate DAC enable DBCR1[D1R,D1W] to cause an address comparison and by setting any bit combination in the DBCR1[DV1BE]. DVC2 debug events are enabled by setting the appropriate DAC enable DBCR1[D2R,D2W] to cause an address comparison and by setting any bit combination in the DBCR1[DV1BE]. Each bit in DBCR1[DV1BE, DV2BE] corresponds to a byte in DVC1 and DVC2. Exact address compare and range address compare work the same for DVC as for a simple DAC.

DBSR[DR1] and DBSR[DW1] record status for DAC1 debug events. Which DBSR bit is set depends on the setting of DBCR1[D1R] and DBCR[D1W]. If DBCR1[D1R] = 1, DBSR[DR1] = 1, assuming that a DVC event occurred. Similarly, if DBCR1[D1W] = 1, DBSR[DW1] = 1, assuming that a DVC event occurred.

Similarly, DBSR[DR2] and DBSR[DW2] record status for DAC2 debug events. Which DBSR bit is set depends on the setting of DBCR1[D2R] and DBCR[D2W]. If DBCR1[D2R] = 1, DBSR[DR2] = 1, assuming that a DVC event occurred. Similarly, if DBCR1[D2W] = 1, DBSR[DW2] = 1, assuming that a DVC event occurred.

In the following example, a DVC1 event is enabled by setting DBCR1[D1R] = 1, DBCR1[D1W] = 1, DBCR1[DA12] = 0, and DBCR1[DV1BE] = 0000. When the data address and data value match the DAC1 and DVC1, a DVC1 event is recorded in DBSR[DR1] or DBSR[DW1], depending on whether the operation is a load (read) or a store (write). This example corresponds to the last line of Table 8-4.

In *Table 8-4* on page 154, *n* is 1 or 2, depending on whether the bits apply to DAC1, DAC2, DVC1, and DVC2 events. “Hold” indicates that the DBSR holds its value unless cleared by software. “RA” indicates that the operation is a read (load) and the data address compares (exact or range). “WA” indicates that the operation is a write (store) and the data address compares (exact or range). “RV” indicates that the operation is a read (load), the data address compares (exact or range), and the data value compares according to DBCR1[DVC*n*].

Table 8-4. Setting of DBSR Bits for DAC and DVC Events

DACn Event	DVCn Enabled	DVCn Event	DBCR1			DBSR	
			[DnR]	[DnW]	[DA12]	[DRn]	[DWn]
0	—	—	—	—	—	Hold	Hold
—	—	—	0	0	—	Hold	Hold
1	0	—	0	1	—	Hold	WA
1	0	—	1	0	—	RA	Hold
1	0	—	1	1	—	RA	WA
1	1	0	—	—	—	Hold	Hold
1	1	1	0	1	—	Hold	WV
1	1	1	1	0	—	RV	Hold
1	1	1	1	1	—	RV	WV

The settings of DBCR1[DV1M] and DBCR1[DV2M] are more precisely defined in *Table 8-6* on page 155 and *Table 8-7* on page 155. (*n* enables the table to apply to DBCR1[DV1M, DV2M] and DBCR1[DV1BE, DV2BE]). DV*n*BE_m indicates bytes selected (or not selected) for comparison in DBCR1[DV*n*BE].

When DBCR1[DV*n*M] = 01, the comparison is an AND; all bytes must compare to the appropriate bytes of DVC1.

When DBCR1[DV*n*M] = 10, the comparison is an OR; at least one of the selected bytes must compare to the appropriate bytes of DVC1.

When DBCR1[DV*n*M] = 11, the comparison is an AND-OR (halfword) comparison. This is intended for use when DBCR1[DV*n*BE] is set to 0011, 0111, or 1111. Other values of DBCR1[DV*n*BE] can be compared, but the results are more easily understood using the AND and OR comparisons. In *Table 8-5*, "not" is ¬, AND is ∧, and OR is ∨.

Table 8-5. Comparisons Based on DBCR1[DVnM]

DBCR1[DVnM] Setting	Operation	Comparison
00	—	Undefined
01	AND	(¬DV*n*BE_0 ∨ (DVC1[byte 0] = data[byte 0])) ∧ (¬DV*n*BE_1 ∨ (DVC1[byte 1] = data[byte 1])) ∧ (¬DV*n*BE_2 ∨ (DVC1[byte 2] = data[byte 2])) ∧ (¬DV*n*BE_3 ∨ (DVC1[byte 3] = data[byte 3]))
10	OR	(DV*n*BE_0 ∧ (DVC1[byte 0] = data[byte 0])) ∨ (DV*n*BE_1 ∧ (DVC1[byte 1] = data[byte 1])) ∨ (DV*n*BE_2 ∧ (DVC1[byte 2] = data[byte 2])) ∨ (DV*n*BE_3 ∧ (DVC1[byte 3] = data[byte 3]))
11	AND-OR	(DV*n*BE_0 ∧ (DVC1[byte 0] = data[byte 0])) ∧ (DV*n*BE_1 ∧ (DVC1[byte 1] = data[byte 1])) ∨ (DV*n*BE_2 ∧ (DVC1[byte 2] = data[byte 2])) ∧ (DV*n*BE_3 ∧ (DVC1[byte 3] = data[byte 1]))

Table 8-6 illustrates comparisons for aligned DVC accesses, that is, words, halfwords, or bytes on naturally aligned boundaries (all byte accesses are aligned).

Table 8-6. Comparisons for Aligned DVC Accesses

Access	DBCR1[DVnBE] Setting	Value	Operation
Word	All	Word value	AND
Halfword (Low-Order)	All	Halfword value replicated	AND-OR
Halfword (High-Order)	All	Halfword value replicated	AND-OR
Byte	All	Byte value replicated	OR

For halfword accesses, the halfword value is replicated in the "empty" halfword in the DVC register, for example, if the low-order halfword is to be compared, its value is stored in the low-order halfword and the high-order halfword of the register. Similarly, a byte value is replicated in each byte in the register.

Table 8-7 illustrates comparisons for misaligned DVC accesses. In the "DVC1" and "DVC2" columns, "x" indicates a don't care.

Table 8-7. Comparisons for Misaligned DVC Accesses

Access	Operation	DVC1 (Hex)	DVC2 (Hex)	DBCR1[DV1BE] Setting	DBCR1[DV2BE] Setting	DBCR1[D2S] Setting
Word (Offset 1)	AND	xx112233	44xxxxxx	123	0	01
Word (Offset 2)	AND	xxxx1122	3344xxxx	23	01	10
Word (Offset 3)	AND	xxxxxx11	223344xx	3	012	10
Halfword (Offset 1)	AND	xx1122xx		12	12	10
Halfword (Offset 3)	AND	xxxxxx11	22xxxxxx	3	0	10

Note: Misaligned accesses stop the processor on the instruction causing the compare hit. The second part of an instruction is not performed if the first part of the compare hits.

8.8.15 Imprecise Debug Event

The imprecise debug event is not an independent debug event, but indicates that a debug event occurred while MSR[DE] = 0. This is useful in internal debug mode if a debug event occurs while in a critical interrupt handler. On return from interrupt, a debug interrupt occurs if MSR[DE] = 1. If DBSR[IDE] = 1, the debug event causing the interrupt occurred sometime earlier, not immediately after a debug event.

9. Instruction Set

Descriptions of the PPC405 instructions follow. Each description contains the following elements:

- Instruction names (mnemonic and full)
- Instruction syntax
- Instruction format diagram
- Pseudocode description
- Prose description
- Registers altered
- Architecture notes identifying the associated PowerPC Architecture component

Where appropriate, instruction descriptions list invalid instruction forms and exceptions, and provide programming notes.

9.1 Instruction Set Portability

To support embedded real-time applications, the instruction sets of the PPC405 and other controllers implement the PowerPC Embedded Environment, which is not part of the PowerPC Architecture defined in *The PowerPC Architecture: A Specification for a New Family of RISC Processors*.

Programs using these instructions are not portable to PowerPC implementations that do not implement the PowerPC Embedded Environment.

The PPC405 implements a number of implementation-specific instructions that are not part of the PowerPC Architecture or the PowerPC Embedded Environment, which are listed in Table 9-1. In the table, the syntax [**o**] indicates that an instruction has an o form, which updates the XER[SO,OV] fields, and a non-o form. The syntax [**.**] indicates that an instruction has a record form, which updates CR[CR0], and a non-record form.

Table 9-1. Implementation-Specific Instructions

dccci **dcread** **iccci** **icread**	**macchw**[**o**][**.**] **macchws**[**o**][**.**] **macchwsu**[**o**][**.**] **macchwu**[**o**][**.**] **machhw**[**o**][**.**] **machhws**[**o**][**.**] **machhwsu**[**o**][**.**] **machhwu**[**o**][**.**] **maclhw**[**o**][**.**] **maclhws**[**o**][**.**] **maclhwsu**[**o**][**.**] **maclhwu**[**o**][**.**]	**mfdcr** **mtdcr** **mulchw**[**.**] **mulchwu**[**.**] **mulhhw**[**.**] **mulhhwu**[**.**] **mullhw**[**.**] **mullhwu**[**.**]	**nmacchw**[**o**][**.**] **nmacchws**[**o**][**.**] **nmachhw**[**o**][**.**] **nmachhws**[**o**][**.**] **nmaclhw**[**o**][**.**] **nmaclhws**[**o**][**.**]	**rfci** **tlbre** **tlbsx**[**.**] **tlbwe** **wrtee** **wrteei**

9.2 Instruction Formats

For more detailed information about instruction formats, including a summary of instruction field usage and instruction format diagrams for the PPC405, see “Instruction Formats” on page 157.

Instructions are four bytes long. Instruction addresses are always word-aligned.

Instruction bits 0 through 5 always contain the primary opcode. Many instructions have an extended opcode in another field. The remaining instruction bits contain additional fields. All instruction fields belong to one of the following categories:

- Defined

  These instructions contain values, such as opcodes, that cannot be altered. The instruction format diagrams specify the values of defined fields.

- Variable

  These fields contain operands, such as general purpose register selectors and immediate values, that may vary from execution to execution. The instruction format diagrams specify the operands in variable fields.

- Reserved

  Bits in a reserved field should be set to 0. In the instruction format diagrams, reserved fields are shaded.

If any bit in a defined field does not contain the expected value, the instruction is illegal and an illegal instruction exception occurs. If any bit in a reserved field does not contain 0, the instruction form is invalid and its result is architecturally undefined. Unless otherwise noted, the execute all invalid instruction forms without causing an illegal instruction exception.

9.3 Pseudocode

The pseudocode that appears in the instruction descriptions provides a semi-formal language for describing instruction operations.

The pseudocode uses the following notation:

=	Assignment
$\wedge$	AND logical operator
$\neg$	NOT logical operator
$\vee$	OR logical operator
$\oplus$	Exclusive-OR (XOR) logical operator
+	Twos complement addition
–	Twos complement subtraction, unary minus
$\times$	Multiplication
$\div$	Division yielding a quotient
%	Remainder of an integer division; (33 % 32) = 1.
$\Vert$	Concatenation
$=, \neq$	Equal, not equal relations
<, >	Signed comparison relations
$\overset{u}{<}, \overset{u}{>}$	Unsigned comparison relations
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.
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.
leave	Leave innermost do loop or do loop specified in a leave statement.
n	A decimal number

0xn	A hexadecimal number
0bn	A binary number
FLD	An instruction or register field
FLD_b	A bit in a named instruction or register field
$FLD_{b:b}$	A range of bits in a named instruction or register field
$FLD_{b,b,...}$	A list of bits, by number or name, in a named instruction or register field
REG_b	A bit in a named register
$REG_{b:b}$	A range of bits in a named register
$REG_{b,b,...}$	A list of bits, by number or name, in a named register
REG[FLD]	A field in a named register
REG[FLD, FLD $_{...}$]	A list of fields in a named register
REG[FLD:FLD]	A range of fields in a named register
GPR(r)	General Purpose Register (GPR) r, where $0 \le r \le 31$.
(GPR(r))	The contents of GPR r, where $0 \le r \le 31$.
DCR(DCRN)	A Device Control Register (DCR) specified by the DCRF field in an **mfdcr** or **mtdcr** instruction
SPR(SPRN)	An SPR specified by the SPRF field in an **mfspr** or **mtspr** instruction
TBR(TBRN)	A Time Base Register (TBR) specified by the TBRF field in an **mftb** instruction
GPRs	RA, RB, $_{...}$
(Rx)	The contents of a GPR, where *x* is A, B, S, or T
(RA|0)	The contents of the register RA or 0, if the RA field is 0.
$c_{0:3}$	A four-bit object used to store condition results in compare instructions.
${}^{n}b$	The bit or bit value *b* is replicated *n* times.
xx	Bit positions which are don't-cares.
CEIL(x)	Least integer $\ge$ x.
EXTS(x)	The result of extending *x* on the left with sign bits.
PC	Program counter.
RESERVE	Reserve bit; indicates whether a process has reserved a block of storage.
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.
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.
MS(addr, n)	The number of bytes represented by *n* at the location in main storage represented by *addr*.
EA	Effective address; the 32-bit address, derived by applying indexing or indirect addressing rules to the specified operand, that specifies an location in main storage.
EA_b	A bit in an effective address.
$EA_{b:b}$	A range of bits in an effective address.
ROTL((RS),n)	Rotate left; the contents of RS are shifted left the number of bits specified by *n*.
MASK(MB,ME)	Mask having 1s in positions MB through ME (wrapping if MB > ME) and 0s elsewhere.
instruction(EA)	An instruction operating on a data or instruction cache block associated with an EA.

9.3.1 Operator Precedence

Table 9-2 lists the pseudocode operators and their associativity in descending order of precedence:

Table 9-2. Operator Precedence

Operators	Associativity
REG_b, REG[FLD], 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

9.4 Register Usage

Each instruction description lists the registers altered by the instruction. Some register changes are explicitly detailed in the instruction description (for example, the target register of a load instruction). Other registers are changed, with the details of the change not included in the instruction description. This category frequently includes the Condition Register (CR) and the Fixed-point Exception Register (XER). For discussion of the CR, see *Condition Register (CR)* on page 39. For discussion of XER, see *Fixed Point Exception Register (XER)* on page 37.

9.5 Alphabetical Instruction Listing

The following pages list the instructions available in the PPC405 in alphabetical order.

add	RT, RA, RB	OE = 0, Rc = 0
add.	RT, RA, RB	OE = 0, Rc = 1
addo	RT, RA, RB	OE = 1, Rc = 0
addo.	RT, RA, RB	OE = 1, Rc = 1

31	RT	RA	RB	OE	266	Rc
0	6	11	16	21	22	31

(RT) $\leftarrow$ (RA) + (RB)

The sum of the contents of register RA and the contents of register RB is placed into register RT.

Registers Altered

- RT
- CR[CR0]$_{LT,\ GT,\ EQ,\ SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

addc	RT, RA, RB	OE = 0, Rc = 0
addc.	RT, RA, RB	OE = 0, Rc = 1
addco	RT, RA, RB	OE = 1, Rc = 0
addco.	RT, RA, RB	OE = 1, Rc = 1

31	RT	RA	RB	OE	10	Rc
0	6	11	16	21	22	31

(RT) $\leftarrow$ (RA) + (RB)
if (RA) + (RB) $\overset{u}{>}$ $2^{32} - 1$ then
  XER[CA] $\leftarrow$ 1
else
  XER[CA] $\leftarrow$ 0

The sum of the contents of register RA and register RB is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

adde	RT, RA, RB	OE = 0, Rc = 0
adde.	RT, RA, RB	OE = 0, Rc = 1
addeo	RT, RA, RB	OE = 1, Rc = 0
addeo.	RT, RA, RB	OE = 1, Rc = 1

31	RT	RA	RB	OE	138	Rc
0	6	11	16	21	22	31

(RT) ← (RA) + (RB) + XER[CA]
if (RA) + (RB) + XER[CA] $\overset{u}{>}$ $2^{32} - 1$ then
  XER[CA] ← 1
else
  XER[CA] ← 0

The sum of the contents of register RA, register RB, and XER[CA] is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

addi RT, RA, IM

14	RT	RA	IM
0	6	11	16 ... 31

$(RT) \leftarrow (RA|0) + EXTS(IM)$

If the RA field is 0, the IM field, sign-extended to 32 bits, is placed into register RT.

If the RA field is nonzero, the sum of the contents of register RA and the contents of the IM field, sign-extended to 32 bits, is placed into register RT.

Registers Altered

- RT

Programming Note

To place an immediate, sign-extended value into the GPR specified by RT, set RA = 0.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-3. Extended Mnemonics for addi

Mnemonic	Operands	Function	Other Registers Altered
la	**RT, D(RA)**	**Load address (RA ≠ 0); D is an offset from a base address that is assumed to be (RA).** **(RT) ← (RA) + EXTS(D)** *Extended mnemonic for* **addi RT,RA,D**	
li	**RT, IM**	**Load immediate.** **(RT) ← EXTS(IM)** *Extended mnemonic for* **addi RT,0,IM**	
subi	**RT, RA, IM**	**Subtract EXTS(IM) from (RA|0).** **Place result in RT.** *Extended mnemonic for* **addi RT,RA,–IM**	

addic RT, RA, IM

12	RT	RA	IM
0	6	11	16 31

(RT) ← (RA) + EXTS(IM)
if (RA) + EXTS(IM) $\overset{u}{>}$ $2^{32} - 1$ then
  XER[CA] ← 1
else
  XER[CA] ← 0

The sum of the contents of register RA and the contents of the IM field, sign-extended to 32 bits, is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-4. Extended Mnemonics for addic

Mnemonic	Operands	Function	Other Registers Altered
subic	**RT, RA, IM**	**Subtract EXTS(IM) from (RA)** **Place result in RT; place carry-out in XER[CA].** *Extended mnemonic for* **addic RT,RA,–IM**	

addic. RT, RA, IM

13	RT	RA	IM
0	6	11	16 … 31

(RT) ← (RA) + EXTS(IM)
if (RA) + EXTS(IM) $\overset{u}{>}$ $2^{32} - 1$ then
  XER[CA] ← 1
else
  XER[CA] ← 0

The sum of the contents of register RA and the contents of the IM field, sign-extended to 32 bits, is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]
- $CR[CR0]_{LT, GT, EQ, SO}$

Programming Note

addic. is one of three instructions that implicitly update CR[CR0] without having an RC field. The other instructions are **andi.** and **andis.**.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-5. Extended Mnemonics for addic.

Mnemonic	Operands	Function	Other Registers Altered
subic.	**RT, RA, IM**	**Subtract EXTS(IM) from (RA).** **Place result in RT; place carry-out in XER[CA].** *Extended mnemonic for* **addic. RT,RA,−IM**	**CR[CR0]**

addis RT, RA, IM

15	RT	RA	IM
0	6	11	16 … 31

$(RT) \leftarrow (RA|0) + (IM \parallel {}^{16}0)$

If the RA field is 0, the IM field is concatenated on its right with sixteen 0-bits and placed into register RT.

If the RA field is nonzero, the contents of register RA are added to the contents of the extended IM field. The sum is stored into register RT.

Registers Altered

- RT

Programming Note

An **addi** instruction stores a sign-extended 16-bit value in a GPR. An **addis** instruction followed by an **ori** instruction stores an arbitrary 32-bit value in a GPR, as shown in the following example:

```
addis    RT, 0, high 16 bits of value
ori      RT, RT, low 16 bits of value
```

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-6. Extended Mnemonics for addis

Mnemonic	Operands	Function	Other Registers Altered
lis	**RT, IM**	**Load immediate shifted.** **$(RT) \leftarrow (IM \parallel {}^{16}0)$** *Extended mnemonic for* **addis RT,0,IM**	
subis	**RT, RA, IM**	**Subtract $(IM \parallel {}^{16}0)$ from (RA|0).** **Place result in RT.** *Extended mnemonic for* **addis RT,RA,–IM**	

addme	RT, RA	OE=0, Rc=0
addme.	RT, RA	OE=0, Rc=1
addmeo	RT, RA	OE=1, Rc=0
addmeo.	RT, RA	OE=1, Rc=1

31	RT	RA		OE	234	Rc
0	6	11	16	21	22	31

(RT) ← (RA) + XER[CA] + (–1)
if (RA) + XER[CA] + 0xFFFF FFFF $\overset{u}{>}$ 2^{32} – 1 then
XER[CA] ← 1
else
XER[CA] ← 0

The sum of the contents of register RA, XER[CA], and –1 is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

addze	RT, RA	OE=0, Rc=0
addze.	RT, RA	OE=0, Rc=1
addzeo	RT, RA	OE=1, Rc=0
addzeo.	RT, RA	OE=1, Rc=1

31	RT	RA		OE	202	Rc
0	6	11	16	21	22	31

(RT) $\leftarrow$ (RA) + XER[CA]
if (RA) + XER[CA] $\overset{u}{>}$ $2^{32} - 1$ then
  XER[CA] $\leftarrow$ 1
else
  XER[CA] $\leftarrow$ 0

The sum of the contents of register RA and XER[CA] is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

and	RA, RS, RB	Rc=0
and.	RA, RS, RB	Rc=1

31	RS	RA	RB	28	Rc
0	6	11	16	21	31

$(RA) \leftarrow (RS) \wedge (RB)$

The contents of register RS are ANDed with the contents of register RB; the result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

andc	RA,RS,RB	Rc=0
andc.	RA,RS,RB	Rc=1

31	RS	RA	RB	60	Rc
0	6	11	16	21 2	31

$(RA) \leftarrow (RS) \wedge \neg(RB)$

The contents of register RS are ANDed with the ones complement of the contents of register RB; the result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

andi. RA, RS, IM

28	RS	RA	IM
0	6	11	16 … 31

$(RA) \leftarrow (RS) \wedge (^{16}0 \parallel IM)$

The IM field is extended to 32 bits by concatenating 16 0-bits on its left. The contents of register RS is ANDed with the extended IM field; the result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$

Programming Note

The **andi.** instruction can test whether any of the 16 least-significant bits in a GPR are 1-bits.

andi. is one of three instructions that implicitly update CR[CR0] without having an Rc field. The other instructions are **addic.** and **andis.**.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

andis. RA, RS, IM

29	RS	RA	IM
0	6	11	16 ... 31

$(RA) \leftarrow (RS) \wedge (IM \parallel {}^{16}0)$

The IM field is extended to 32 bits by concatenating 16 0-bits on its right. The contents of register RS are ANDed with the extended IM field; the result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$

Programming Note

The **andis.** instruction can test whether any of the 16 most-significant bits in a GPR are 1-bits.

andis. is one of three instructions that implicitly update CR[CR0] without having an Rc field. The other instructions are **addic.** and **andi.**.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

b	target	AA=0, LK=0
ba	target	AA=1, LK=0
bl	target	AA=0, LK=1
bla	target	AA=1, LK=1

18	LI	AA	LK
0	6	30	31

If AA = 1 then
   LI $\leftarrow$ $target_{6:29}$
   NIA $\leftarrow$ EXTS(LI $\|$ $^{2}0$)
else
   LI $\leftarrow$ $(target – CIA)_{6:29}$
   NIA $\leftarrow$ CIA + EXTS(LI $\|$ $^{2}0$)
if LK = 1 then
   (LR) $\leftarrow$ CIA + 4
PC $\leftarrow$ NIA

The next instruction address (NIA) is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the LI field and sign-extending the result to 32 bits.

If the AA field contains 0, the base address is the address of the branch instruction, which is also the current instruction address (CIA). If the AA field contains 1, the base address is 0.

Program flow is transferred to the NIA.

If the LK field contains 1, then (CIA + 4) is placed into the LR.

Registers Altered

- LR if LK contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

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

16	BO	BI	BD	AA	LK
0	6	11	16	30	31

```
if BO2 = 0 then
   CTR ← CTR – 1
if (BO2 = 1 ∨ ((CTR = 0) = BO3)) ∧ (BO0 = 1 ∨ (CRBI = BO1))  then
   if AA = 1 then
      BD ← target16:29
      NIA ← EXTS(BD || 20)
   else
      BD ← (target – CIA)16:29
      NIA ← CIA + EXTS(BD || 20)
else
   NIA ← CIA + 4
if LK = 1 then
   (LR) ← CIA + 4
PC ← NIA
```

If bit 2 of the BO field contains 0, the CTR decrements.

The BI field specifies a bit in the CR to be used as the condition of the branch.

The next instruction address (NIA) is the effective address of the branch. The NIA is formed by adding a displacement to a base address. 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, the base address is the address of the branch instruction, which is also the current instruction address (CIA). If the AA field contains 1, the base address is 0.

The BO field controls options that determine when program flow is transferred to the NIA. The BO field also controls branch prediction, a performance-improvement feature. See *Branch Prediction* on page 52 for a complete discussion.

If the LK field contains 1, then (CIA + 4) is placed into the LR.

Registers Altered

- CTR if BO_2 contains 0
- LR if LK contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla

Mnemonic	Operands	Function	Other Registers Altered
bdnz	**target**	**Decrement CTR; branch if CTR ≠ 0.** *Extended mnemonic for* **bc 16,0,target**	
bdnza		*Extended mnemonic for* **bca 16,0,target**	
bdnzl		*Extended mnemonic for* **bcl 16,0,target**	**(LR) ← CIA + 4.**
bdnzla		*Extended mnemonic for* **bcla 16,0,target**	**(LR) ← CIA + 4.**
bdnzf	**cr_bit, target**	**Decrement CTR.** **Branch if CTR ≠ 0 AND CR_{cr_bit} = 0.** *Extended mnemonic for* **bc 0,cr_bit,target**	
bdnzfa		*Extended mnemonic for* **bca 0,cr_bit,target**	
bdnzfl		*Extended mnemonic for* **bcl 0,cr_bit,target**	**(LR) ← CIA + 4.**
bdnzfla		*Extended mnemonic for* **bcla 0,cr_bit,target**	**(LR) ← CIA + 4.**
bdnzt	**cr_bit, target**	**Decrement CTR.** **Branch if CTR ≠ 0 AND CR_{cr_bit} = 1.** *Extended mnemonic for* **bc 8,cr_bit,target**	
bdnzta		*Extended mnemonic for* **bca 8,cr_bit,target**	
bdnztl		*Extended mnemonic for* **bcl 8,cr_bit,target**	**(LR) ← CIA + 4.**
bdnztla		*Extended mnemonic for* **bcla 8,cr_bit,target**	**(LR) ← CIA + 4.**
bdz	**target**	**Decrement CTR; branch if CTR = 0.** *Extended mnemonic for* **bc 18,0,target**	
bdza		*Extended mnemonic for* **bca 18,0,target**	
bdzl		*Extended mnemonic for* **bcl 18,0,target**	**(LR) ← CIA + 4.**
bdzla		*Extended mnemonic for* **bcla 18,0,target**	**(LR) ← CIA + 4.**

Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla (Continued)

Mnemonic	Operands	Function	Other Registers Altered
bdzf	**cr_bit, target**	**Decrement CTR** **Branch if CTR = 0 AND CR_{cr_bit} = 0.** *Extended mnemonic for* **bc 2,cr_bit,target**	
bdzfa		*Extended mnemonic for* **bca 2,cr_bit,target**	
bdzfl		*Extended mnemonic for* **bcl 2,cr_bit,target**	**(LR) ← CIA + 4.**
bdzfla		*Extended mnemonic for* **bcla 2,cr_bit,target**	**(LR) ← CIA + 4.**
bdzt	**cr_bit, target**	**Decrement CTR** **Branch if CTR = 0 AND CR_{cr_bit} = 1.** *Extended mnemonic for* **bc 10,cr_bit,target**	
bdzta		*Extended mnemonic for* **bca 10,cr_bit,target**	
bdztl		*Extended mnemonic for* **bcl 10,cr_bit,target**	**(LR) ← CIA + 4.**
bdztla		*Extended mnemonic for* **bcla 10,cr_bit,target**	**(LR) ← CIA + 4.**
beq	**[cr_field,] target**	**Branch if equal** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bc 12,4∗cr_field+2,target**	
beqa		*Extended mnemonic for* **bca 12,4∗cr_field+2,target**	
beql		*Extended mnemonic for* **bcl 12,4∗cr_field+2,target**	**(LR) ← CIA + 4.**
beqla		*Extended mnemonic for* **bcla 12,4∗cr_field+2,target**	**(LR) ← CIA + 4.**
bf	**cr_bit, target**	**Branch if CR_{cr_bit} = 0.** *Extended mnemonic for* **bc 4,cr_bit,target**	
bfa		*Extended mnemonic for* **bca 4,cr_bit,target**	
bfl		*Extended mnemonic for* **bcl 4,cr_bit,target**	**LR**
bfla		*Extended mnemonic for* **bcla 4,cr_bit,target**	**LR**

Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla (Continued)

Mnemonic	Operands	Function	Other Registers Altered
bge	**[cr_field,] target**	**Branch if greater than or equal.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bc 4,4∗cr_field+0,target**	
bgea		*Extended mnemonic for* **bca 4,4∗cr_field+0,target**	
bgel		*Extended mnemonic for* **bcl 4,4∗cr_field+0,target**	**LR**
bgela		*Extended mnemonic for* **bcla 4,4∗cr_field+0,target**	**LR**
bgt	**[cr_field,] target**	**Branch if greater than.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bc 12,4∗cr_field+1,target**	
bgta		*Extended mnemonic for* **bca 12,4∗cr_field+1,target**	
bgtl		*Extended mnemonic for* **bcl 12,4∗cr_field+1,target**	**LR**
bgtla		*Extended mnemonic for* **bcla 12,4∗cr_field+1,target**	**LR**
ble	**[cr_field,] target**	**Branch if less than or equal.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bc 4,4∗cr_field+1,target**	
blea		*Extended mnemonic for* **bca 4,4∗cr_field+1,target**	
blel		*Extended mnemonic for* **bcl 4,4∗cr_field+1,target**	**LR**
blela		*Extended mnemonic for* **bcla 4,4∗cr_field+1,target**	**LR**
blt	**[cr_field,] target**	**Branch if less than** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bc 12,4∗cr_field+0,target**	
blta		*Extended mnemonic for* **bca 12,4∗cr_field+0,target**	
bltl		*Extended mnemonic for* **bcl 12,4∗cr_field+0,target**	**(LR) ← CIA + 4.**
bltla		*Extended mnemonic for* **bcla 12,4∗cr_field+0,target**	**(LR) ← CIA + 4.**

Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla (Continued)

Mnemonic	Operands	Function	Other Registers Altered
bne	**[cr_field,] target**	**Branch if not equal.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bc 4,4∗cr_field+2,target**	
bnea		*Extended mnemonic for* **bca 4,4∗cr_field+2,target**	
bnel		*Extended mnemonic for* **bcl 4,4*cr_field+2,target**	**(LR) ← CIA + 4.**
bnela		*Extended mnemonic for* **bcla 4,4∗cr_field+2,target**	**(LR) ← CIA + 4.**
bng	**[cr_field,] target**	**Branch if not greater than.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bc 4,4∗cr_field+1,target**	
bnga		*Extended mnemonic for* **bca 4,4∗cr_field+1,target**	
bngl		*Extended mnemonic for* **bcl 4,4∗cr_field+1,target**	**(LR) ← CIA + 4.**
bngla		*Extended mnemonic for* **bcla 4,4∗cr_field+1,target**	**(LR) ← CIA + 4.**
bnl	**[cr_field,] target**	**Branch if not less than; use CR0 if cr_field is omitted.** *Extended mnemonic for* **bc 4,4∗cr_field+0,target**	
bnla		*Extended mnemonic for* **bca 4,4∗cr_field+0,target**	
bnll		*Extended mnemonic for* **bcl 4,4∗cr_field+0,target**	**(LR) ← CIA + 4.**
bnlla		*Extended mnemonic for* **bcla 4,4∗cr_field+0,target**	**(LR) ← CIA + 4.**
bns	**[cr_field,] target**	**Branch if not summary overflow.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bc 4,4∗cr_field+3,target**	
bnsa		*Extended mnemonic for* **bca 4,4∗cr_field+3,target**	
bnsl		*Extended mnemonic for* **bcl 4,4∗cr_field+3,target**	**(LR) ← CIA + 4.**
bnsla		*Extended mnemonic for* **bcla 4,4∗cr_field+3,target**	**(LR) ← CIA + 4.**

Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla (Continued)

Mnemonic	Operands	Function	Other Registers Altered
bnu	**[cr_field,] target**	**Branch if not unordered.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bc 4,4∗cr_field+3,target**	
bnua		*Extended mnemonic for* **bca 4,4∗cr_field+3,target**	
bnul		*Extended mnemonic for* **bcl 4,4∗cr_field+3,target**	**(LR) ← CIA + 4.**
bnula		*Extended mnemonic for* **bcla 4,4∗cr_field+3,target**	**(LR) ← CIA + 4.**
bso	**[cr_field,] target**	**Branch if summary overflow.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bc 12,4∗cr_field+3,target**	
bsoa		*Extended mnemonic for* **bca 12,4∗cr_field+3,target**	
bsol		*Extended mnemonic for* **bcl 12,4∗cr_field+3,target**	**(LR) ← CIA + 4.**
bsola		*Extended mnemonic for* **bcla 12,4∗cr_field+3,target**	**(LR) ← CIA + 4.**
bt	**cr_bit, target**	**Branch if CR_{cr_bit} = 1.** *Extended mnemonic for* **bc 12,cr_bit,target**	
bta		*Extended mnemonic for* **bca 12,cr_bit,target**	
btl		*Extended mnemonic for* **bcl 12,cr_bit,target**	**(LR) ← CIA + 4.**
btla		*Extended mnemonic for* **bcla 12,cr_bit,target**	**(LR) ← CIA + 4.**
bun	**[cr_field], target**	**Branch if unordered.** **Use CR0 if *cr_field* is omitted.** *Extended mnemonic for* **bc 12,4∗cr_field+3,target**	
buna		*Extended mnemonic for* **bca 12,4∗cr_field+3,target**	
bunl		*Extended mnemonic for* **bcl 12,4∗cr_field+3,target**	**(LR) ← CIA + 4.**
bunla		*Extended mnemonic for* **bcla 12,4∗cr_field+3,target**	**(LR) ← CIA + 4.**

bcctr	BO, BI	LK = 0
bcctrl	BO, BI	LK = 1

19	BO	BI		528	LK
0	6	11	16	21	31

if $BO_2 = 0$ then
  $CTR \leftarrow CTR - 1$
if $(BO_2 = 1 \lor ((CTR = 0) = BO_3)) \land (BO_0 = 1 \lor (CR_{BI} = BO_1))$ then
  $NIA \leftarrow CTR_{0:29} \parallel {}^{2}0$
else
  $NIA \leftarrow CIA + 4$
if $LK = 1$ then
  $(LR) \leftarrow CIA + 4$
$PC \leftarrow NIA$

The BI field specifies a bit in the CR to be used as the condition of the branch.

The next instruction address (NIA) is the target address of the branch. The NIA is formed by concatenating the 30 most significant bits of the CTR with two 0-bits on the right.

The BO field controls options that determine when program flow is transferred to the NIA. The BO field also controls branch prediction, a performance-improvement feature. See *Branch Prediction* on page 52 for a complete discussion.

If the LK field contains 1, then (CIA + 4) is placed into the LR.

Registers Altered

- CTR if BO_2 contains 0
- LR if LK contains 1

Invalid Instruction Forms

- Reserved fields
- If bit 2 of the BO field contains 0, the instruction form is invalid, but the pseudocode applies. If the branch condition is true, the branch is taken; the NIA is the contents of the CTR after it is decremented.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-8. Extended Mnemonics for bcctr, bcctrl

Mnemonic	Operands	Function	Other Registers Altered
bctr		**Branch unconditionally to address in CTR.** *Extended mnemonic for* **bcctr 20,0**	
bctrl		*Extended mnemonic for* **bcctrl 20,0**	**(LR) ← CIA + 4.**

Table 9-8. Extended Mnemonics for bcctr, bcctrl (Continued)

Mnemonic	Operands	Function	Other Registers Altered
beqctr	**[cr_field]**	**Branch, if equal, to address in CTR** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 12,4∗cr_field+2**	
beqctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+2**	**(LR) ← CIA + 4.**
bfctr	**cr_bit**	**Branch, if CR_{cr_bit} = 0, to address in CTR.** *Extended mnemonic for* **bcctr 4,cr_bit**	
bfctrl		*Extended mnemonic for* **bcctrl 4,cr_bit**	**(LR) ← CIA + 4.**
bgectr	**[cr_field]**	**Branch, if greater than or equal, to address in CTR. Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 4,4∗cr_field+0**	
bgectrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+0**	**(LR) ← CIA + 4.**
bgtctr	**[cr_field]**	**Branch, if greater than, to address in CTR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 12,4∗cr_field+1**	
bgtctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+1**	**(LR) ← CIA + 4.**
blectr	**[cr_field]**	**Branch, if less than or equal, to address in CTR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 4,4∗cr_field+1**	
blectrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+1**	**(LR) ← CIA + 4.**
bltctr	**[cr_field]**	**Branch, if less than, to address in CTR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 12,4∗cr_field+0**	
bltctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+0**	**(LR) ← CIA + 4.**
bnectr	**[cr_field]**	**Branch, if not equal, to address in CTR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 4,4∗cr_field+2**	
bnectrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+2**	**(LR) ← CIA + 4.**
bngctr	**[cr_field]**	**Branch, if not greater than, to address in CTR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 4,4∗cr_field+1**	
bngctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+1**	**(LR) ← CIA + 4.**

Table 9-8. Extended Mnemonics for bcctr, bcctrl (Continued)

Mnemonic	Operands	Function	Other Registers Altered
bnlctr	**[cr_field]**	**Branch, if not less than, to address in CTR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 4,4∗cr_field+0**	
bnlctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+0**	**(LR) ← CIA + 4.**
bnsctr	**[cr_field]**	**Branch, if not summary overflow, to address in CTR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 4,4∗cr_field+3**	
bnsctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+3**	**(LR) ← CIA + 4.**
bnuctr	**[cr_field]**	**Branch, if not unordered, to address in CTR; use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 4,4∗cr_field+3**	
bnuctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+3**	**(LR) ← CIA + 4.**
bsoctr	**[cr_field]**	**Branch, if summary overflow, to address in CTR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 12,4∗cr_field+3**	
bsoctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+3**	**(LR) ← CIA + 4.**
btctr	**cr_bit**	**Branch if CR_{cr_bit} = 1 to address in CTR.** *Extended mnemonic for* **bcctr 12,cr_bit**	
btctrl		*Extended mnemonic for* **bcctrl 12,cr_bit**	**(LR) ← CIA + 4.**
bunctr	**[cr_field]**	**Branch if unordered to address in CTR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bcctr 12,4∗cr_field+3**	
bunctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+3**	**(LR) ← CIA + 4.**

bclr	BO, BI	LK = 0
bclrl	BO, BI	LK = 1

19	BO	BI		16	LK
0	6	11	16	21	31

if $BO_2 = 0$ then
  $CTR \leftarrow CTR - 1$
if $(BO_2 = 1 \lor ((CTR = 0) = BO_3)) \land (BO_0 = 1 \lor (CR_{BI} = BO_1))$ then
  $NIA \leftarrow LR_{0:29} \parallel {}^{2}0$
else
  $NIA \leftarrow CIA + 4$
if $LK = 1$ then
  $(LR) \leftarrow CIA + 4$
$PC \leftarrow NIA$

If bit 2 of the BO field contains 0, the CTR is decremented.

The BI field specifies a bit in the CR to be used as the condition of the branch.

The next instruction address (NIA) is the target address of the branch. The NIA is formed by concatenating the 30 most significant bits of the LR with two 0-bits on the right.

The BO field controls options that determine when program flow is transferred to the NIA. The BO field also controls branch prediction, a performance-improvement feature. See *Branch Prediction* on page 52 for a complete discussion.

If the LK field contains 1, then (CIA + 4) is placed into the LR.

Registers Altered

- CTR if BO_2 contains 0
- LR if LK contains 1

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-9. Extended Mnemonics for bclr, bclrl

Mnemonic	Operands	Function	Other Registers Altered
blr		**Branch unconditionally to address in LR.** *Extended mnemonic for* **bclr 20,0**	
blrl		*Extended mnemonic for* **bclrl 20,0**	**(LR) ← CIA + 4.**

Table 9-9. Extended Mnemonics for bclr, bclrl (Continued)

Mnemonic	Operands	Function	Other Registers Altered
bdnzlr		**Decrement CTR.** **Branch if CTR ≠ 0 to address in LR.** *Extended mnemonic for* **bclr 16,0**	
bdnzlrl		*Extended mnemonic for* **bclrl 16,0**	**(LR) ← CIA + 4.**
bdnzflr	**cr_bit**	**Decrement CTR.** **Branch if CTR ≠ 0 AND CR_{cr_bit} = 0 to address in LR.** *Extended mnemonic for* **bclr 0,cr_bit**	
bdnzflrl		*Extended mnemonic for* **bclrl 0,cr_bit**	**(LR) ← CIA + 4.**
bdnztlr	**cr_bit**	**Decrement CTR.** **Branch if CTR ≠ 0 AND CR_{cr_bit} = 1 to address in LR.** *Extended mnemonic for* **bclr 8,cr_bit**	
bdnztlrl		*Extended mnemonic for* **bclrl 8,cr_bit**	**(LR) ← CIA + 4.**
bdzlr		**Decrement CTR.** **Branch if CTR = 0 to address in LR.** *Extended mnemonic for* **bclr 18,0**	
bdzlrl		*Extended mnemonic for* **bclrl 18,0**	**(LR) ← CIA + 4.**
bdzflr	**cr_bit**	**Decrement CTR.** **Branch if CTR = 0 AND CR_{cr_bit} = 0 to address in LR.** *Extended mnemonic for* **bclr 2,cr_bit**	
bdzflrl		*Extended mnemonic for* **bclrl 2,cr_bit**	**(LR) ← CIA + 4.**
bdztlr	**cr_bit**	**Decrement CTR.** **Branch if CTR = 0 AND CR_{cr_bit} = 1 to address in LR.** *Extended mnemonic for* **bclr 10,cr_bit**	
bdztlrl		*Extended mnemonic for* **bclrl 10,cr_bit**	**(LR) ← CIA + 4.**
beqlr	**[cr_field]**	**Branch if equal to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 12,4∗cr_field+2**	
beqlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+2**	**(LR) ← CIA + 4.**
bflr	**cr_bit**	**Branch if CR_{cr_bit} = 0 to address in LR.** *Extended mnemonic for* **bclr 4,cr_bit**	
bflrl		*Extended mnemonic for* **bclrl 4,cr_bit**	**(LR) ← CIA + 4.**

Table 9-9. Extended Mnemonics for bclr, bclrl (Continued)

Mnemonic	Operands	Function	Other Registers Altered
bgelr	**[cr_field]**	**Branch, if greater than or equal, to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 4,4∗cr_field+0**	
bgelrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+0**	**(LR) ← CIA + 4.**
bgtlr	**[cr_field]**	**Branch, if greater than, to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 12,4∗cr_field+1**	
bgtlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+1**	**(LR) ← CIA + 4.**
blelr	**[cr_field]**	**Branch, if less than or equal, to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 4,4∗cr_field+1**	
blelrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+1**	**(LR) ← CIA + 4.**
bltlr	**[cr_field]**	**Branch, if less than, to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 12,4∗cr_field+0**	
bltlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+0**	**(LR) ← CIA + 4.**
bnelr	**[cr_field]**	**Branch, if not equal, to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 4,4∗cr_field+2**	
bnelrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+2**	**(LR) ← CIA + 4.**
bnglr	**[cr_field]**	**Branch, if not greater than, to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 4,4∗cr_field+1**	
bnglrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+1**	**(LR) ← CIA + 4.**
bnllr	**[cr_field]**	**Branch, if not less than, to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 4,4∗cr_field+0**	
bnllrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+0**	**(LR) ← CIA + 4.**
bnslr	**[cr_field]**	**Branch if not summary overflow to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 4,4∗cr_field+3**	
bnslrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+3**	**(LR) ← CIA + 4.**

Table 9-9. Extended Mnemonics for bclr, bclrl (Continued)

Mnemonic	Operands	Function	Other Registers Altered
bnulr	**[cr_field]**	**Branch if not unordered to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 4,4∗cr_field+3**	
bnulrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+3**	**(LR) ← CIA + 4.**
bsolr	**[cr_field]**	**Branch if summary overflow to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 12,4∗cr_field+3**	
bsolrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+3**	**(LR) ← CIA + 4.**
btlr	**cr_bit**	**Branch if CR_{cr_bit} = 1 to address in LR.** *Extended mnemonic for* **bclr 12,cr_bit**	
btlrl		*Extended mnemonic for* **bclrl 12,cr_bit**	**(LR) ← CIA + 4.**
bunlr	**[cr_field]**	**Branch if unordered to address in LR.** **Use CR0 if cr_field is omitted.** *Extended mnemonic for* **bclr 12,4∗cr_field+3**	
bunlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+3**	**(LR) ← CIA + 4.**

cmp BF, 0, RA, RB

31	BF		RA	RB	0	
0	6	9	11	16	21	31

$c_{0:3} \leftarrow {}^{4}0$
if (RA) < (RB) then $c_0 \leftarrow 1$
if (RA) > (RB) then $c_1 \leftarrow 1$
if (RA) = (RB) then $c_2 \leftarrow 1$
$c_3 \leftarrow$ XER[SO]
n ← BF
CR[CRn] ← $c_{0:3}$

The contents of register RA are compared with the contents of register RB using a 32-bit signed compare.

The CR field specified by the BF field is updated to reflect the results of the compare and the value of XER[SO] is placed into the same CR field.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR[CR*n*] where *n* is specified by the BF field

Invalid Instruction Forms

- Reserved fields

Programming Note

The PowerPC Architecture defines this instruction as **cmp BF,L,RA,RB**, where L selects operand size for 64-bit PowerPC implementations. For all 32-bit PowerPC implementations, L = 0 is required (L = 1 is an invalid form); hence for PPC405, use of the extended mnemonic **cmpw BF,RA,RB** is recommended.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-10. Extended Mnemonics for cmp

Mnemonic	Operands	Function	Other Registers Altered
cmpw	**[BF,] RA, RB**	**Compare Word; use CR0 if BF is omitted.** *Extended mnemonic for* **cmp BF,0,RA,RB**	

cmpi BF, 0, RA, IM

11	BF		RA	IM
0	6	9	11	16 … 31

$c_{0:3} \leftarrow {}^{4}0$
if (RA) < EXTS(IM) then $c_0 \leftarrow 1$
if (RA) > EXTS(IM) then $c_1 \leftarrow 1$
if (RA) = EXTS(IM) then $c_2 \leftarrow 1$
$c_3 \leftarrow$ XER[SO]
n ← BF
CR[CRn] ← $c_{0:3}$

The IM field is sign-extended to 32 bits. The contents of register RA are compared with the extended IM field, using a 32-bit signed compare.

The CR field specified by the BF field is updated to reflect the results of the compare and the value of XER[SO] is placed into the same CR field.

Registers Altered

- CR[CR*n*] where *n* is specified by the BF field

Invalid Instruction Forms

- Reserved fields

Programming Note

The PowerPC Architecture defines this instruction as **cmpi BF,L,RA,IM**, where L selects operand size for 64-bit PowerPC implementations. For all 32-bit PowerPC implementations, L = 0 is required (L = 1 is an invalid form); hence for the PPC405, use of the extended mnemonic **cmpwi BF,RA,IM** is recommended.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-11. Extended Mnemonics for cmpi

Mnemonic	Operands	Function	Other Registers Altered
cmpwi	**[BF,] RA, IM**	**Compare Word Immediate.** **Use CR0 if BF is omitted.** *Extended mnemonic for* **cmpi BF,0,RA,IM**	

cmpl BF, 0, RA, RB

31	BF		RA	RB	32	
0	6	9	11	16	21	31

$c_{0:3} \leftarrow {}^4 0$
if (RA) $\overset{u}{<}$ (RB) then $c_0 \leftarrow 1$
if (RA) $\overset{u}{>}$ (RB) then $c_1 \leftarrow 1$
if (RA) = (RB) then $c_2 \leftarrow 1$
$c_3 \leftarrow$ XER[SO]
n ← BF
CR[CRn] ← $c_{0:3}$

The contents of register RA are compared with the contents of register RB, using a 32-bit unsigned compare.

The CR field specified by the BF field is updated to reflect the results of the compare and the value of XER[SO] is placed into the same CR field.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR[CR*n*] where *n* is specified by the BF field

Invalid Instruction Forms

- Reserved fields

Programming Notes

The PowerPC Architecture defines this instruction as **cmpl BF,L,RA,RB**, where L selects operand size for 64-bit PowerPC implementations. For all 32-bit PowerPC implementations, L = 0 is required (L = 1 is an invalid form); hence for PPC405, use of the extended mnemonic **cmplw BF,RA,RB** is recommended.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-12. Extended Mnemonics for cmpl

Mnemonic	Operands	Function	Other Registers Altered
cmplw	**[BF,] RA, RB**	**Compare Logical Word.** **Use CR0 if BF is omitted.** *Extended mnemonic for* **cmpl BF,0,RA,RB**	

cmpli BF, 0, RA, IM

10	BF		RA	IM
0	6	9	11	16 … 31

$c_{0:3} \leftarrow {}^{4}0$
if (RA) $\overset{u}{<}$ $({}^{16}0 \parallel \text{IM})$ then $c_0 \leftarrow 1$
if (RA) $\overset{u}{>}$ $({}^{16}0 \parallel \text{IM})$ then $c_1 \leftarrow 1$
if (RA) $=$ $({}^{16}0 \parallel \text{IM})$ then $c_2 \leftarrow 1$
$c_3 \leftarrow$ XER[SO]
$n \leftarrow$ BF
CR[CRn] $\leftarrow c_{0:3}$

The IM field is extended to 32 bits by concatenating 16 0-bits to its left. The contents of register RA are compared with IM using a 32-bit unsigned compare.

The CR field specified by the BF field is updated to reflect the results of the compare and the value of XER[SO] is placed into the same CR field.

Registers Altered

- CR[CR*n*] where *n* is specified by the BF field

Invalid Instruction Forms

- Reserved fields

Programming Note

The PowerPC Architecture defines this instruction as **cmpli BF,L,RA,IM**, where L selects operand size for 64-bit PowerPC implementations. For all 32-bit PowerPC implementations, L = 0 is required (L = 1 is an invalid form); hence for the PPC405, use of the extended mnemonic **cmplwi BF,RA,IM** is recommended.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-13. Extended Mnemonics for cmpli

Mnemonic	Operands	Function	Other Registers Changed
cmplwi	**[BF,] RA, IM**	**Compare Logical Word Immediate.** **Use CR0 if BF is omitted.** *Extended mnemonic for* **cmpli BF,0,RA,IM**	

cntlzw	RA, RS	Rc=0
cntlzw.	RA, RS	Rc=1

31	RS	RA		26	Rc
0	6	11	16	21	31

```
n ← 0
do while n < 32
   if (RS)n = 1 then leave
   n ← n + 1
(RA) ← n
```

The consecutive leading 0 bits in register RS are counted; the count is placed into register RA.

The count ranges from 0 through 32, inclusive.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

crand BT, BA, BB

19	BT	BA	BB	257	
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \wedge CR_{BB}$

The CR bit specified by the BA field is ANDed with the CR bit specified by the BB field; the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

crandc BT, BA, BB

19	BT	BA	BB	129	
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \land \neg CR_{BB}$

The CR bit specified by the BA field is ANDed with the ones complement of the CR bit specified by the BB field; the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

creqv BT, BA, BB

19	BT	BA	BB	289	
0	6	11	16	21	31

$$CR_{BT} \leftarrow \neg(CR_{BA} \oplus CR_{BB})$$

The CR bit specified by the BA field is XORed with the CR bit specified by the BB field; the ones complement of the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-14. Extended Mnemonics for creqv

Mnemonic	Operands	Function	Other Registers Altered
crset	**bx**	**CR set.** *Extended mnemonic for* **creqv bx,bx,bx**	

crnand BT, BA, BB

19	BT	BA	BB	225	
0	6	11	16	21	31

$CR_{BT} \leftarrow \neg(CR_{BA} \wedge CR_{BB})$

The CR bit specified by the BA field is ANDed with the CR bit specified by the BB field; the ones complement of the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

crnor BT, BA, BB

19	BT	BA	BB	33	
0	6	11	16	21	31

$CR_{BT} \leftarrow \neg(CR_{BA} \vee CR_{BB})$

The CR bit specified by the BA field is ORed with the CR bit specified by the BB field; the ones complement of the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-15. Extended Mnemonics for crnor

Mnemonic	Operands	Function	Other Registers Altered
crnot	**bx, by**	**CR not.** *Extended mnemonic for* **crnor bx,by,by**	

cror BT, BA, BB

19	BT	BA	BB	449	
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \vee CR_{BB}$

The CR bit specified by the BA field is ORed with the CR bit specified by the BB field; the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-16. Extended Mnemonics for cror

Mnemonic	Operands	Function	Other Registers Altered
crmove	**bx, by**	**CR move.** *Extended mnemonic for* **cror bx,by,by**	

crorc BT, BA, BB

19	BT	BA	BB	417	
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \vee \neg CR_{BB}$

The condition register (CR) bit specified by the BA field is ORed with the ones complement of the CR bit specified by the BB field; the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

crxor BT, BA, BB

19	BT	BA	BB	193	
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \oplus CR_{BB}$

The CR bit specified by the BA field is XORed with the CR bit specified by the BB field; the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-17. Extended Mnemonics for crxor

Mnemonic	Operands	Function	Other Registers Altered
crclr	**bx**	**Condition register clear.** *Extended mnemonic for* **crxor bx,bx,bx**	

dcba RA, RB

31		RA	RB	758	
0	6	11	16	21	31

$EA \leftarrow (RA|0) + (RB)$
DCBA(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the data block at the EA is in the data cache and the EA is marked as cacheable and non-write-through, the data in the cache block is architecturally undefined. For the PPC405, the cache data block is set to 0.

If the data block at the EA is not in the data cache and the EA is marked as cacheable and not marked as write-through, a cache block is established and set to an architecturally-undefined value. Note that no data is read from main storage, as described in the programming note.

If the data block at the EA is marked as non cacheable, a no-op occurs.

If the data block at the EA is in the data cache and marked as write-through, architecturally the data in the cache block can be left unmodified. Alternatively, the data block at the EA can be undefined in the data cache and in main storage. For the PPC405, a no-op occurs.

If the data block at the EA is not in the data cache and marked as write-through, architecturally the instruction can establish a cache block and set the block to 0, or a no-op can occur. For the PPC405, a no-op occurs.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

Because **dcba** can establish an address in the data cache without copying the contents of that address from main storage, the address established can be invalid with respect to main storage. A subsequent operation may cause the address to be copied back to main storage, for example, to make room for a new cache block; a machine check exception could occur under these circumstances.

dcba provides a hint that a block of storage will soon be stored to or no longer needed; there is no need to retain the data in the block. Establishing the line in the cache, without reading from main storage, improves performance.

Exceptions

This instruction is considered a "store" 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 such exceptions, **dcba** is treated as a no-op.

This instruction is considered a “store” with respect to data address compare (DAC) debug exceptions. See *Data Storage Interrupt* on page 120.

Architecture Note

This instruction is part of the PowerPC Embedded Virtual Environment.

dcbf RA, RB

31		RA	RB	86	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
DCBF(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the data block corresponding to the EA is in the data cache and marked as modified (stored into), the data block is copied back to main storage and then marked invalid in the data cache. If the data block is not marked as modified, it is simply marked invalid in the data cache. The operation is performed whether or not the EA is marked as cacheable.

If the data block at the EA is not in the data cache, no operation is performed.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Exceptions

This instruction is considered a “load” with respect to data storage exceptions. See *Data Storage Interrupt* on page 120.

This instruction is considered a “store” with respect to data address compare (DAC) debug exceptions. See *Debug Interrupt* on page 128.

Architecture Note

This instruction is part of the PowerPC Embedded Virtual Environment.

dcbi RA, RB

31		RA	RB	470	
0	6	11	16	21	31

$$EA \leftarrow (RA|0) + (RB)$$
DCBI(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the data block at the EA is in the data cache, the data block is marked invalid, regardless of whether or not the EA is marked as cacheable. If modified data existed in the data block prior to the operation of this instruction, that data is lost.

If the data block at the EA is not in the data cache, no operation is performed.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

Execution of this instruction is privileged.

Exceptions

This instruction is considered a “store” with respect to data storage exceptions. See *Data Storage Interrupt* on page 120.

This instruction is considered a “store” with respect to data address compare (DAC) debug exceptions. See *Debug Interrupt* on page 128.

Architecture Note

This instruction is part of the PowerPC Embedded Operating Environment.

dcbst RA, RB

31		RA	RB	54	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
DCBST(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0, and is the contents of register RA otherwise.

If the data block at the EA is in the data cache and marked as modified, the data block is copied back to main storage and marked as unmodified in the data cache.

If the data block at the EA is in the data cache, and is not marked as modified, or if the data block at the EA is not in the data cache, no operation is performed.

The operation specified by this instruction is performed whether or not the EA is marked as cacheable.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Exceptions

This instruction is considered a "load" with respect to data storage exceptions. See *Data Storage Interrupt* on page 120.

This instruction is considered a "store" with respect to data address compare (DAC) debug exceptions. See *Debug Interrupt* on page 128.

Architecture Note

This instruction is part of the PowerPC Embedded Virtual Environment.

dcbt RA, RB

31		RA	RB	278	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
DCBT(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

If the data block at the EA is not in the data cache and the EA is marked as cacheable, the block is read from main storage into the data cache.

If the data block at the EA is in the data cache, or if the EA is marked as non cacheable, no operation is performed.

This instruction is not allowed to cause data storage exceptions or data TLB miss exceptions. If execution of the instruction would cause such an exception, then no operation is performed, and no exception occurs.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

The **dcbt** instruction allows a program to begin a cache block fetch from main storage before the program needs the data. The program can later load data from the cache into registers without incurring the latency of a cache miss.

Exceptions

This instruction is considered a “load” with respect to data storage exceptions. See *Data Storage Interrupt* on page 120.

This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions. See *Debug Interrupt* on page 128.

Architecture Note

This instruction is part of the PowerPC Embedded Virtual Environment.

dcbtst RA, RB

31		RA	RB	246	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
DCBTST(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the data block at the EA is not in the data cache and the EA address is marked as cacheable, the data block is loaded into the data cache.

If the EA is marked as non cacheable, or if the data block at the EA is in the data cache, no operation is performed.

This instruction is not allowed to cause data storage exceptions or data TLB miss exceptions. If execution of the instruction would cause such an exception, then no operation is performed, and no exception occurs.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

The **dcbtst** instruction allows a program to begin a cache block fetch from main storage before the program needs the data. The program can later store data from GPRs into the cache block, without incurring the latency of a cache miss.

Architecturally, **dcbtst** brings data into the cache in “Exclusive” mode, which allows the program to alter the cached data. “Exclusive” mode is part of the MESI protocol for multi-processor systems, and is not implemented. The implementation of the **dcbtst** instruction is identical to the implementation of the **dcbt** instruction.

Exceptions

This instruction is considered a “load” with respect to data storage exceptions. See *Data Storage Interrupt* on page 120.

This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions. See *Debug Interrupt* on page 128.

Architecture Note

This instruction is part of the PowerPC Embedded Virtual Environment.

dcbz RA, RB

31		RA	RB	1014	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
DCBZ(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the data block at the EA is in the data cache and the EA is marked as cacheable and non-write-through, the data in the cache block is set to 0.

If the data block at the EA is not in the data cache and the EA is marked as cacheable and non-write-through, a cache block is established and set to 0. Note that nothing is read from main storage, as described in the programming note.

If the data block at the EA is marked as either write-through or as non cacheable, an alignment exception occurs.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

Because **dcbz** can establish an address in the data cache without copying the contents of that address from main storage, the address established may be invalid with respect to the storage subsystem. A subsequent operation may cause the address to be copied back to main storage, for example, to make room for a new cache block; a machine check exception could occur under these circumstances.

If **dcbz** is attempted to an EA which is marked as non cacheable, the software alignment exception handler should emulate the instruction by storing zeros to the block in main storage. If a data block corresponding to the EA exists in the cache, but the EA is non cacheable, stores (including **dcbz**) to that address are considered programming errors (the cache block should previously have been flushed).

If the EA is marked as write-through, the software alignment exception handler should emulate the instruction by storing zeros to the block in main storage. An EA that is marked as write-through required should also be marked as cacheable; when **dcbz** is attempted to such an address, the alignment exception handler should maintain coherency of cache and memory.

Exceptions

An alignment exception occurs if the EA is marked as non cacheable or as write-through.

This instruction is considered a “store” with respect to data storage exceptions. See *Data Storage Interrupt* on page 120.

This instruction is considered a “store” with respect to data address compare (DAC) debug exceptions. See *Debug Interrupt* on page 128.

Architecture Note

This instruction is part of the PowerPC Embedded Virtual Environment.

dccci RA, RB

31		RA	RB	454	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
DCCCI(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

Both cache lines in the congruence class specified by $EA_{18:26}$ are invalidated, whether or not they match the EA. If modified data existed in the cache congruence class before the operation of this instruction, that data is lost.

The operation specified by this instruction is performed whether or not the EA is marked as cacheable.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged.

This instruction is intended for use in the power-on reset routine to invalidate the entire data cache tag array before enabling the data cache. A series of **dccci** instruction should be executed, one for each congruence class. Cachability can then be enabled.

Exceptions

See *Access Protection for Cache Control Instructions* on page 104.

The execution of an **dccci** instruction can cause a data TLB miss exception, at the specified EA, regardless of the non-specific intent of that EA.

This instruction does not cause data address compare (DAC) debug exceptions. See *Debug Interrupt* on page 128.

Architecture Note

This instruction is implementation-specific and may not be portable to other implementations.

dcread RT, RA, RB

31	RT	RA	RB	486	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
if ((CCR0[CIS] = 0) $\wedge$ (CCR0[CWS] = 0)) then (RT) $\leftarrow$ (d-cache data, way A)
if ((CCR0[CIS] = 0) $\wedge$ (CCR0[CWS] = 1)) then (RT) $\leftarrow$ (d-cache data, way B)
if ((CCR0[CIS] = 1) $\wedge$ (CCR0[CWS] = 0)) then (RT) $\leftarrow$ (d-cache tag, way A)
if ((CCR0[CIS] = 1) $\wedge$ (CCR0[CWS] = 1)) then (RT) $\leftarrow$ (d-cache tag, way B)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

This instruction is a debugging tool for reading the data cache entries for the congruence class specified by $EA_{18:26}$. The cache information is read into register RT.

If CCR0[CIS] = 0, the information is a word of data cache array data from the addressed congruence class. The word is specified by $EA_{27:29}$. If $EA_{30:31}$ are not 00, an alignment exception occurs. If CCR0[CWS] = 0, the data is from the A-way; otherwise; the data is from the B-way.

If CCR0[CIS] = 1, the information is a cache tag from the addressed congruence class. If CCR0[CWS] = 0, the tag is from the A-way; otherwise the tag is from the B-way.

Data cache tag information is placed into register RT as shown:

0:19	TAG	Cache Tag
20:25		Reserved
26	D	Cache Line Dirty 0 Not dirty 1 Dirty
27	V	Cache Line Valid 0 Not valid 1 Valid
28:30		Reserved
31	LRU	Least Recently Used (LRU) 0 A-way LRU 1 B-way LRU

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged.

Exceptions

If EA is not word-aligned, an alignment exception occurs.

This instruction is considered a “load” with respect to data storage exceptions, but cannot cause a data storage exception. See *Access Protection for Cache Control Instructions* on page 104.

The execution of an **dcread** instruction can cause a data TLB miss exception, at the specified EA, regardless of the non-specific intent of that effective address.

This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions. See *Debug Interrupt* on page 128.

Architecture Note

This instruction is implementation-specific and may not be portable to other implementations.

divw	RT, RA, RB	OE=0, Rc=0
divw.	RT, RA, RB	OE=0, Rc=1
divwo	RT, RA, RB	OE=1, Rc=0
divwo.	RT, RA, RB	OE=1, Rc=1

31	RT	RA	RB	OE	491	Rc
0	6	11	16	21	22	31

(RT) $\leftarrow$ (RA) $\div$ (RB)

The contents of register RA are divided by the contents of register RB. The quotient is placed into register RT.

Both the dividend and the divisor are interpreted as signed integers. The quotient is the unique signed integer that satisfies:

dividend = (quotient $\times$ divisor) + remainder

where the remainder has the same sign as the dividend and its magnitude is less than that of the divisor.

If an attempt is made to perform (0x8000 0000 $\div$ –1) or ($n \div 0$), the contents of register RT are undefined; if the Rc field also contains 1, the contents of $CR[CR0]_{LT, GT, EQ}$ are undefined. Either invalid division operation sets XER[OV, SO] to 1 if the OE field contains 1.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[OV, SO] if OE contains 1

Programming Note

The 32-bit remainder can be calculated using the following sequence of instructions:

```
divw    RT,RA,RB    # RT = quotient
mullw   RT,RT,RB    # RT = quotient × divisor
subf    RT,RT,RA    # RT = remainder
```

The sequence does not calculate correct results for the invalid divide operations.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

divwu	RT, RA, RB	OE=0, Rc=0
divwu.	RT, RA, RB	OE=0, Rc=1
divwuo	RT, RA, RB	OE=1, Rc=0
divwuo.	RT, RA, RB	OE=1, Rc=1

31	RT	RA	RB	OE	459	Rc
0	6	11	16	21	22	31

$(RT) \leftarrow (RA) \div (RB)$

The contents of register RA are divided by the contents of register RB. The quotient is placed into register RT.

The dividend and the divisor are interpreted as unsigned integers. The quotient is the unique unsigned integer that satisfies:

dividend = (quotient $\times$ divisor) + remainder

If an attempt is made to perform ($n \div 0$), the contents of register RT are undefined; if the Rc also contains 1, the contents of $CR[CR0]_{LT, GT, EQ}$ are also undefined. The invalid division operation also sets XER[OV, SO] to 1 if the OE field contains 1.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[OV, SO] if OE contains 1

Programming Note

The 32-bit remainder can be calculated using the following sequence of instructions

```
divwu   RT,RA,RB     # RT = quotient
mullw   RT,RT,RB     # RT = quotient × divisor
subf    RT,RT,RA     # RT = remainder
```

This sequence does not calculate the correct result if the divisor is zero.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

eieio

31		854	
0	6	21	31

The **eieio** instruction ensures that all loads and stores preceding **eieio** complete with respect to main storage before any loads and stores following **eieio** access main storage.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

Architecturally, **eieio** orders storage access, not instruction completion. Therefore, non-storage operations after **eieio** could complete before storage operations that were before **eieio**. The **sync** instruction guarantees ordering of both instruction completion and storage access. For the PPC405, the **eieio** instruction is implemented to behave as a **sync** instruction.

To write code that is portable between various PowerPC implementations, programmers should use the mnemonic that corresponds to the desired behavior.

Architecture Note

This instruction is part of the PowerPC Embedded Virtual Environment.

eqv	RA, RS, RB	Rc=0
eqv.	RA, RS, RB	Rc=1

31	RS	RA	RB	284	Rc
0	6	11	16	21	31

$(RA) \leftarrow \neg((RS) \oplus (RB))$

The contents of register RS are XORed with the contents of register RB; the ones complement of the result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

extsb	RA, RS	Rc=0
extsb.	RA, RS	Rc=1

31	RS	RA		954	Rc
0	6	11	16	21	31

$(RA) \leftarrow EXTS(RS)_{24:31}$

The least significant byte of register RS is sign-extended to 32 bits by replicating bit 24 of the register into bits 0 through 23 of the result. The result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

extsh	RA, RS	Rc=0
extsh.	RA, RS	Rc=1

31	RS	RA		922	Rc
0	6	11	16	21	31

$(RA) \leftarrow EXTS(RS)_{16:31}$

The least significant halfword of register RS is sign-extended to 32 bits by replicating bit 16 of the register into bits 0 through 15 of the result. The result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

icbi

Instruction Cache Block Invalidate

icbi RA, RB

31		RA	RB	982	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
ICBI(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the instruction block at the EA is in the instruction cache, the cache block is marked invalid.

If the instruction block at the EA is not in the instruction cache, no additional operation is performed.

The operation specified by this instruction is performed whether or not the EA is marked as cacheable in the ICCR.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

Instruction cache operations use MSR[DR], not MSR[IR], to determine translation of their operands.

When data translation is disabled, cachability for the EA of the operand of instruction cache operations is determined by the ICCR, not the DCCR.

Exceptions

Instruction storage exceptions and instruction-side TLB miss exceptions are associated with instruction *fetching*, not with instruction execution. Exceptions that occur during the *execution* of instruction cache operations cause data-side exceptions (data storage exceptions and data TLB miss exceptions).

This instruction is considered a "load" with respect to data storage exceptions. See *Debug Interrupt* on page 128.

This instruction is considered a "load" with respect to data address compare (DAC) debug exceptions.

Architecture Note

This instruction is part of the PowerPC Embedded Virtual Environment.

icbt RA, RB

31		RA	RB	262	
0	6	11	16	21	31

EA← (RA|0) + (RB)
ICBT(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the instruction block at the EA is not in the instruction cache, and is marked as cacheable, the instruction block is loaded into the instruction cache.

If the instruction block at the EA is in the instruction cache, or if the EA is marked as non cacheable, no operation is performed.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

This instruction allows a program to begin a cache block fetch from main storage before the program needs the instruction. The program can later branch to the instruction address and fetch the instruction from the cache without incurring the latency of a cache miss.

Instruction cache operations use MSR[DR], not MSR[IR], to determine translation of their operands. When data translation is disabled, cachability for the effective address of the operand of instruction cache operations is determined by the ICCR, not the DCCR.

Exceptions

Instruction storage exceptions and instruction-side TLB miss exceptions are associated with instruction *fetching*, not with instruction execution. Exceptions occurring during *execution* of instruction cache operations cause data storage and data TLB miss exceptions.

If the execution of an **icbt** instruction would cause a data TLB miss exception, no operation is performed and no exception occurs.

This instruction is considered a "load" with respect to protection exceptions, but cannot cause data storage exceptions. This instruction is also considered a "load" with respect to data address compare (DAC) debug exceptions.

Architecture Note

This instruction is part of the PowerPC Embedded Operating Environment.

iccci RA, RB

31		RA	RB	966	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
ICCCI(ICU cache array)

This instruction invalidates the entire ICU cache array. The EA is not used; previous implementations have used the EA for protection checks. The instruction form is maintained for software and tool compatibility.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

Execution of this instruction is privileged.

This instruction is intended for use in the power-on reset routine to invalidate the entire cache tag array before enabling the cache. Cachability can then be enabled.

Architecture Note

This instruction is implementation-specific and may not be portable to other implementations.

icread RA, RB

31		RA	RB	998	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
if ((CCR0[CIS] = 0) $\wedge$ (CCR0[CWS] = 0)) then (ICDBDR) $\leftarrow$ (i-cache data, way A)
if ((CCR0[CIS] = 0) $\wedge$ (CCR0[CWS] = 1)) then (ICDBDR) $\leftarrow$ (i-cache data, way B)
if ((CCR0[CIS] = 1) $\wedge$ (CCR0[CWS] = 0)) then (ICDBDR) $\leftarrow$ (i-cache tag, way A)
if ((CCR0[CIS] = 1) $\wedge$ (CCR0[CWS] = 1)) then (ICDBDR) $\leftarrow$ (i-cache tag, way B)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

This instruction is a debugging tool for reading the instruction cache entries for the congruence class specified by $EA_{18:26}$. The cache information is read into the Instruction Cache Debug Data Register (ICDBDR), from where it can be read into a GPR using the extended mnemonic **mficdbdr**.

If CCR0[CIS] = 0, the information is a word of instruction cache data from the addressed line. The word is specified by $EA_{27:29}$. If CCR0[CWS] = 0, the data is from the A-way, otherwise from the B-way.

If (CCR0[CIS] = 1), the information is a cache tag from the addressed congruence class. If (CCR0[CWS] = 0), the tag is from the A-way, otherwise from the B-way.

Instruction cache tag information is placed in the ICDBDR as shown:

0:21	TAG	Cache Tag
22:26		Reserved
27	V	Cache Line Valid 0 Not valid 1 Valid
28:30		Reserved
31	LRU	Least Recently Used (LRU) 0 A-way LRU 1 B-way LRU

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- ICDBDR

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged.

The instruction pipeline does not automatically wait for data from **icread** to arrive at the ICDBDR before attempting to use the contents of the ICDBDR. Therefore, insert an **isync** instruction between **icread** and **mficdbdr**.

```
icread r5,r6   # read cache information
isync          # ensure completion of icread
mficdbdr r7    # move information to GPR
```

Instruction cache operations use MSR[DR], not MSR[IR], to determine translation of their operands. When data translation is disabled, cachability for the EA of the operand of instruction cache operations is determined by the ICCR, not the DCCR.

Exceptions

Instruction storage exceptions and instruction-side TLB miss exceptions are associated with instruction *fetching*, not with instruction execution. Exceptions that occur during the *execution* of instruction cache operations cause data-side exceptions (data storage exceptions and data TLB miss exceptions).

The execution of **icread** can cause a data TLB miss exception, at the specified EA, regardless of the non-specific intent of that EA.

This instruction is considered a “load” and cannot cause a data storage exception.

This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions, but will not cause DAC debug events.

Architecture Note

This instruction is implementation-specific and may not be portable to other implementations.

isync

19		150	
0	6	21	31

The **isync** instruction is a context synchronizing instruction.

isync provides an ordering function for the effects of all instructions executed by the processor. Executing **isync** insures that all instructions preceding the **isync** instruction execute before **isync** completes, except that storage accesses caused by those instructions need not have completed.

No subsequent instructions are initiated by the processor until **isync** completes. Finally, execution of **isync** causes the processor to discard any prefetched instructions, with the effect that subsequent instructions are fetched and executed in the context established by the instructions preceding **isync**.

isync has no effect on caches.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

See the discussion of context synchronizing instructions in *Synchronization* on page 58.

The following code example illustrates the necessary steps for self-modifying code. This example assumes that addr1 is both data and instruction cacheable.

```
stw      regN, addr1     # data in regN is to become an instruction at addr1
dcbst    addr1           # forces data from the data cache to memory
sync                     # wait until the data actually reaches the memory
icbi     addr1           # the previous value at addr1 might already be in
                           the instruction cache; invalidate in the cache
isync                    # the previous value at addr1 might already have been
                           pre-fetched into the queue; invalidate the queue
                           so that the instruction must be re-fetched
```

Architecture Note

This instruction is part of the PowerPC Embedded Virtual Environment.

lbz RT, D(RA)

34	RT	RA	D
0	6	11	16 … 31

$$EA \leftarrow (RA|0) + EXTS(D)$$
$$(RT) \leftarrow {}^{24}0 \;||\; MS(EA,1)$$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The byte at the EA is extended to 32 bits by concatenating 24 0-bits to its left. The result is placed into register RT.

Registers Altered

- RT

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lbzu RT, D(RA)

35	RT	RA	D
0	6	11	16 ... 31

EA ← (RA|0) + EXTS(D)
(RA) ← EA
(RT) ← $^{24}0$ || MS(EA,1)

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise. The EA is placed into register RA.

The byte at the EA is extended to 32 bits by concatenating 24 0-bits to its left. The result is placed into register RT.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- RA=RT
- RA=0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lbzux RT, RA, RB

31	RT	RA	RB	119	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
(RA) $\leftarrow$ EA
(RT) $\leftarrow$ ${}^{24}0$ || MS(EA,1)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise. The EA is placed into register RA.

The byte at the EA is extended to 32 bits by concatenating 24 0-bits to its left. The result is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- Reserved fields
- RA=RT
- RA=0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lbzx RT,RA, RB

31	RT	RA	RB	87	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
(RT) $\leftarrow$ $^{24}0$ || MS(EA,1)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The byte at the EA is extended to 32 bits by concatenating 24 0-bits to its left. The result is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lha RT, D(RA)

42	RT	RA	D
0	6	11	16 ... 31

EA $\leftarrow$ (RA|0) + EXTS(D)
(RT) $\leftarrow$ EXTS(MS(EA,2))

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The halfword at the EA is sign-extended to 32 bits and placed into register RT.

Registers Altered

- RT

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhau RT, D(RA)

43	RT	RA	D
0	6	11	16 31

EA $\leftarrow$ (RA) + EXTS(D)
(RA) $\leftarrow$ EA
(RT) $\leftarrow$ EXTS(MS(EA,2))

An effective address (EA) is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The EA is placed into register RA.

The halfword at the EA is sign-extended to 32 bits and placed into register RT.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- RA = RT
- RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhaux RT, RA, RB

31	RT	RA	RB	375	
0	6	11	16	21	31

EA $\leftarrow$ (RA) + (RB)
(RA) $\leftarrow$ EA
(RT) $\leftarrow$ EXTS(MS(EA,2))

An effective address (EA) is formed by adding an index to the base address in register RA. The index is the contents of register RB. The EA is placed into register RA.

The halfword at the EA is sign-extended to 32 bits and placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- Reserved fields
- RA = RT
- RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhax RT, RA, RB

31	RT	RA	RB	343	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
(RT) $\leftarrow$ EXTS(MS(EA,2))

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The halfword at the EA is sign-extended to 32 bits and placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhbrx RT, RA, RB

31	RT	RA	RB	790	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
(RT) $\leftarrow$ $^{16}0$ || MS(EA +1,1) || MS(EA,1)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The halfword at the EA is byte-reversed. The resulting halfword is extended to 32 bits by concatenating 16 0-bits to its left. The result is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhz RT, D(RA)

40	RT	RA	D
0	6	11	16 … 31

EA $\leftarrow$ (RA|0) + EXTS(D)
(RT) $\leftarrow$ ${}^{16}0$ || MS(EA,2)

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The halfword at the EA is extended to 32 bits by concatenating 16 0-bits to its left. The result is placed into register RT.

Registers Altered

- RT

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhzu RT, D(RA)

41	RT	RA	D
0	6	11	16 … 31

EA $\leftarrow$ (RA) + EXTS(D)
(RA) $\leftarrow$ EA
(RT) $\leftarrow$ ${}^{16}0 \parallel$ MS(EA,2)

An effective address (EA) is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The EA is placed into register RA.

The halfword at the EA is extended to 32 bits by concatenating 16 0-bits to its left. The result is placed into register RT.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- RA = RT
- RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhzux **RT, RA, RB**

31	RT	RA	RB	311	
0	6	11	16	21	31

EA $\leftarrow$ (RA) + (RB)
(RA) $\leftarrow$ EA
(RT) $\leftarrow$ $^{16}0$ || MS(EA,2)

An effective address (EA) is formed by adding an index to the base address in register RA. The index is the contents of register RB. The EA is placed into register RA.

The halfword at the EA is extended to 32 bits by concatenating 16 0-bits to its left. The result is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- Reserved fields
- RA = RT
- RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhzx RT, RA, RB

31	RT	RA	RB	279	
0	6	11	16	21	31

$EA \leftarrow (RA|0) + (RB)$
$(RT) \leftarrow {}^{16}0 \; || \; MS(EA,2)$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The halfword at the EA is extended to 32 bits by concatenating 16 0-bits to its left. The result is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lmw RT, D(RA)

46	RT	RA	D
0	6	11	16 … 31

```
EA ← (RA|0) + EXTS(D)
r ← RT
do while r ≤ 31
   if ((r ≠ RA) ∨ (r = 31)) then
      (GPR(r)) ← MS(EA,4)
   r ← r + 1
   EA ← EA + 4
```

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field in the instruction to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

A series of consecutive words starting at the EA are loaded into a set of consecutive GPRs, starting with register RT and continuing to and including GPR(31). Register RA is not altered by this instruction (unless RA is GPR(31), which is an invalid form of this instruction). The word which would have been placed into register RA is discarded.

Registers Altered

- RT through GPR(31).

Invalid Instruction Forms

- RA is in the range of registers to be loaded, including the case RA = RT = 0.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lswi RT, RA, NB

31	RT	RA	NB	597	
0	6	11	16	21	31

```
EA ← (RA|0)
if NB = 0 then
   CNT ← 32
else
   CNT ← NB
n ← CNT
R_FINAL ← ((RT + CEIL(CNT/4) – 1) % 32)
r ← RT – 1
i ← 0
do while n > 0
   if i = 0 then
      r ← r + 1
      if r = 32 then
         r ← 0
      if ((r ≠ RA) ∨ (r = R_FINAL)) then
         (GPR(r)) ← 0
   if ((r ≠ RA) ∨ (r = R_FINAL)) then
      (GPR(r)_i:i+7) ← MS(EA,1)
   i ← i + 8
   if i = 32 then
      i ← 0
   EA ← EA + 1
   n ← n – 1
```

An effective address (EA) is determined by the RA field. If the RA field contains 0, the EA is 0. Otherwise, the EA is the contents of register RA.

The NB field specifies the byte count CNT. If the NB field contains 0, the byte count is CNT = 32. Otherwise, the byte count is CNT = NB.

A series of CNT consecutive bytes in main storage, starting at the EA, are loaded into CEIL(CNT/4) consecutive GPRs, four bytes per GPR, until the byte count is exhausted. Bytes are loaded into GPRs; the byte at the lowest address is loaded into the most significant byte. Bits to the right of the last byte loaded into the last GPR are set to 0.

The set of loaded GPRs starts at register RT, continues consecutively through GPR(31), and wraps to register 0, loading until the byte count is exhausted, which occurs in register R_{FINAL}. Register RA is not altered (unless RA = R_{FINAL}, an invalid form of this instruction). Bytes which would have been loaded into register RA are discarded.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT and subsequent GPRs as described above.

Invalid Instruction Forms

- Reserved fields
- RA is in the range of registers to be loaded
- RA = RT = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lswx RT, RA, RB

31	RT	RA	RB	533	
0	6	11	16	21	31

$EA \leftarrow (RA|0) + (RB)$
$CNT \leftarrow XER[TBC]$
$n \leftarrow CNT$
$R_{FINAL} \leftarrow ((RT + CEIL(CNT/4) – 1) \% 32)$
$r \leftarrow RT – 1$
$i \leftarrow 0$
do while $n > 0$
  if $i = 0$ then
    $r \leftarrow r + 1$
    if $r = 32$ then
      $r \leftarrow 0$
    if $(((r \neq RA) \wedge (r \neq RB)) \vee (r = R_{FINAL}))$ then
      $(GPR(r)) \leftarrow 0$
  if $(((r \neq RA) \wedge (r \neq RB)) \vee (r = R_{FINAL}))$ then
    $(GPR(r)_{i:i+7}) \leftarrow MS(EA,1)$
  $i \leftarrow i + 8$
  if $i = 32$ then
    $i \leftarrow 0$
  $EA \leftarrow EA + 1$
  $n \leftarrow n – 1$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

A byte count CNT is obtained from XER[TBC].

A series of CNT consecutive bytes in main storage, starting at the EA, are loaded into CEIL(CNT/4) consecutive GPRs, four bytes per GPR, until the byte count is exhausted. Bytes are loaded into GPRs; the byte having the lowest address is loaded into the most significant byte. Bits to the right of the last byte loaded in the last GPR used are set to 0.

The set of consecutive GPRs loaded starts at register RT, continues through GPR(31), and wraps to register 0, loading until the byte count is exhausted, which occurs in register R_{FINAL}. Register RA is not altered (unless RA = R_{FINAL}, which is an invalid form of this instruction). Register RB is not altered (unless RB = R_{FINAL}, which is an invalid form of this instruction). Bytes which would have been loaded into registers RA or RB are discarded.

If XER[TBC] is 0, the byte count is 0 and the contents of register RT are undefined.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT and subsequent GPRs as described above.

Invalid Instruction Forms

- Reserved fields
- RA or RB is in the range of registers to be loaded.

- RA = RT = 0

Programming Note

If XER[TBC] = 0, the contents of register RT are unchanged and **lswx** is treated as a no-op.

The PowerPC Architecture states that, if XER[TBC] = 0 and if the EA is such that a precise data exception would normally occur (if not for the zero length), **lswx** is treated as a no-op and the precise exception will not occur. Data storage exceptions and alignment exceptions are examples of precise data exceptions.

However, the PowerPC Architecture makes no statement regarding imprecise exceptions related to **lswx** with XER[TBC] = 0. The PPC405 generates an imprecise exception (machine check) on this instruction when all of the following conditions are true:

- The instruction passes all protection bounds checking
- The address is cacheable
- The address is passed to the data cache
- The address misses in the data cache (resulting in a line fill request)
- The address encounters some form of bus error

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwarx RT, RA, RB

31	RT	RA	RB	20	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
RESERVE $\leftarrow$ 1
(RT) $\leftarrow$ MS(EA,4)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The word at the EA is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Execution of the **lwarx** instruction sets the reservation bit.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Programming Note

lwarx and the **stwcx.** instruction should paired in a loop, as shown in the following example, to create the effect of an atomic operation to a memory area used as a semaphore between asynchronous processes. Only **lwarx** can set the reservation bit to 1. **stwcx.** sets the reservation bit to 0 upon its completion, whether or not **stwcx.** sent (RS) to memory. CR[CR0]$_{EQ}$ must be examined to determine whether (RS) was sent to memory.

```
loop: lwarx   # read the semaphore from memory; set reservation
"alter"       # change the semaphore bits in register as required
stwcx.        # attempt to store semaphore; reset reservation
bne loop      # an asynchronous process has intervened; try again
```

If the asynchronous process in the code example had paired **lwarx** with a store other than **stwcx.**, the reservation bit would not have been cleared in the asynchronous process, and the code example would have overwritten the semaphore.

Exceptions

An alignment exception occurs if the EA is not word-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwbrx RT, RA, RB

31	RT	RA	RB	534	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
(RT) $\leftarrow$ MS(EA+3,1) $\|$ MS(EA+2,1) $\|$ MS(EA+1,1) $\|$ MS(EA,1)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The word at the EA is byte-reversed: the least significant byte becomes the most significant byte, the next least significant byte becomes the next most significant byte, and so on. The resulting word is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwz RT, D(RA)

32	RT	RA	D
0	6	11	16 … 31

EA $\leftarrow$ (RA|0) + EXTS(D)
(RT) $\leftarrow$ MS(EA,4)

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The word at the EA is placed into register RT.

Registers Altered

- RT

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwzu RT, D(RA)

33	RT	RA	D
0	6	11	16 … 31

EA $\leftarrow$ (RA) + EXTS(D)
(RA) $\leftarrow$ EA
(RT) $\leftarrow$ MS(EA,4)

An effective address (EA) is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The EA is placed into register RA.

The word at the EA is placed into register RT.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- RA = RT
- RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwzux RT, RA, RB

31	RT	RA	RB	55	
0	6	11	16	21	31

EA $\leftarrow$ (RA) + (RB)
(RA) $\leftarrow$ EA
(RT) $\leftarrow$ MS(EA,4)

An effective address (EA) is formed by adding an index to the base address in register RA. The index is the contents of register RB. The EA is placed into register RA.

The word at the EA is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- Reserved fields
- RA = RT
- RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwzx RT, RA, RB

31	RT	RA	RB	23	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
(RT) $\leftarrow$ MS(EA,4)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The word at the EA is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

macchw	RT, RA, RB	OE=0, Rc=0
macchw.	RT, RA, RB	OE=0, Rc=1
macchwo	RT, RA, RB	OE=1, Rc=0
macchwo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	172	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ signed

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

$(RT) \leftarrow temp_{1:32}$

The low-order halfword of RA is multiplied by the high-order halfword of RB. The signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register. The contents of RT are replaced by the low-order 32 bits of the temporary register.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

macchws	RT, RA, RB	OE=0, Rc=0
macchws.	RT, RA, RB	OE=0, Rc=1
macchwso	RT, RA, RB	OE=1, Rc=0
macchwso.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	236	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ signed

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \,\|\, ^{31}(\neg RT_0))$

else $(RT) \leftarrow temp_{1:32}$

The low-order halfword of RA is multiplied by the high-order halfword of RB. The signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register.

If a result does not overflow, the low-order 32 bits of the temporary register are stored in RT.

If a result overflows, the returned result is the nearest representable value. Thus, if a result is less than -2^{31}, the value stored in RT is -2^{31}. Likewise, if a result is greater than $2^{31} - 1$, the value stored in RT is $2^{31} - 1$.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

macchwsu	RT, RA, RB	OE=0, Rc=0
macchwsu.	RT, RA, RB	OE=0, Rc=1
macchwsuo	RT, RA, RB	OE=1, Rc=0
macchwsuo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	204	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ unsigned

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

$(RT) \leftarrow (temp_{1:32} \vee {}^{32}temp_0)$

The low-order halfword of RA is multiplied by the high-order halfword of RB. The unsigned product is summed with the contents of RT and the sum is stored in a 33-bit temporary register.

If a result does not overflow, the low-order 32 bits of the temporary register are stored in RT.

If a result overflows, the returned result is the nearest representable value. Thus, if a result is greater than $2^{32} – 1$, the value stored in RT is $2^{32} – 1$.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

macchwu	RT, RA, RB	OE=0, Rc=0
macchwu.	RT, RA, RB	OE=0, Rc=1
macchwuo	RT, RA, RB	OE=1, Rc=0
macchwuo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	140	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ unsigned

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

$(RT) \leftarrow temp_{1:32}$

The low-order halfword of RA is multiplied by the high-order halfword of RB. The unsigned product is summed with the contents of RT and the sum is stored in a 33-bit temporary register. The contents of RT are replaced by the low-order 32 bits of the temporary register.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

machhw	RT, RA, RB	OE=0, Rc=0
machhw.	RT, RA, RB	OE=0, Rc=1
machhwo	RT, RA, RB	OE=1, Rc=0
machhwo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	44	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{0:15}$ x $(RB)_{0:15}$ signed

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

$(RT) \leftarrow temp_{1:32}$

The high-order halfword of RA is multiplied by the high-order halfword of RB. The signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register. The contents of RT are replaced by the low-order 32 bits of the temporary register.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

machhws	RT, RA, RB	OE=0, Rc=0
machhws.	RT, RA, RB	OE=0, Rc=1
machhwso	RT, RA, RB	OE=1, Rc=0
machhwso.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	108	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ signed

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$

else $(RT) \leftarrow temp_{1:32}$

The high-order halfword of RA is multiplied by the high-order halfword of RB. The signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register.

If a result does not overflow, the low-order 32 bits of the temporary register are stored in RT.

If a result overflows, the returned result is the nearest representable value. Thus, if a result is less than -2^{31}, the value stored in RT is -2^{31}. Likewise, if a result is greater than $2^{31} - 1$, the value stored in RT is $2^{31} - 1$.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

machhwsu	RT, RA, RB	OE=0, Rc=0
machhwsu.	RT, RA, RB	OE=0, Rc=1
machhwsuo	RT, RA, RB	OE=1, Rc=0
machhwsuo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	76	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ unsigned

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

$(RT) \leftarrow (temp_{1:32} \vee {}^{32}temp_0)$

The high-order halfword of RA is multiplied by the high-order halfword of RB. The unsigned product is summed with the contents of RT and the sum is stored in a 33-bit temporary register.

If a result does not overflow, the low-order 32 bits of the temporary register are stored in RT.

If a result overflows, the returned result is the nearest representable value. Thus, if a result is greater than $2^{32} – 1$, the value stored in RT is $2^{32} – 1$.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

machhwu	RT, RA, RB	OE=0, Rc=0
machhwu.	RT, RA, RB	OE=0, Rc=1
machhwuo	RT, RA, RB	OE=1, Rc=0
machhwuo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	12	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ unsigned

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

$(RT) \leftarrow temp_{1:32}$

The high-order halfword of RA is multiplied by the high-order halfword of RB. The unsigned product is summed with the contents of RT and the sum is stored in a 33-bit temporary register. The contents of RT are replaced by the low-order 32 bits of the temporary register.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

maclhw	RT, RA, RB	OE=0, Rc=0
maclhw.	RT, RA, RB	OE=0, Rc=1
maclhwo	RT, RA, RB	OE=1, Rc=0
maclhwo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	428	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ signed

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

$(RT) \leftarrow temp_{1:32}$

The low-order halfword of RA is multiplied by the low-order halfword of RB. The signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register. The contents of RT are replaced by the low-order 32 bits of the temporary register.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

maclhws	RT, RA, RB	OE=0, Rc=0
maclhws.	RT, RA, RB	OE=0, Rc=1
maclhwso	RT, RA, RB	OE=1, Rc=0
maclhwso.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	492	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ signed

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$

else $(RT) \leftarrow temp_{1:32}$

The low-order halfword of RA is multiplied by the low-order halfword of RB. The signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register.

If a result does not overflow, the low-order 32 bits of the temporary register are stored in RT.

If a result overflows, the returned result is the nearest representable value. Thus, if a result is less than -2^{31}, the value stored in RT is -2^{31}. Likewise, if a result is greater than $2^{31} - 1$, the value stored in RT is $2^{31} - 1$.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

maclhwsu	RT, RA, RB	OE=0, Rc=0
maclhwsu.	RT, RA, RB	OE=0, Rc=1
maclhwsuo	RT, RA, RB	OE=1, Rc=0
maclhwsuo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	460	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ unsigned

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

$(RT) \leftarrow (temp_{1:32} \vee {}^{32}temp_0)$

The low-order halfword of RA is multiplied by the low-order halfword of RB. The unsigned product is summed with the contents of RT and the sum is stored in a 33-bit temporary register.

If a result does not overflow, the low-order 32 bits of the temporary register are stored in RT.

If a result overflows, the returned result is the nearest representable value. Thus, if a result is greater than $2^{32} - 1$, the value stored in RT is $2^{32} - 1$.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

maclhwu	RT, RA, RB	OE=0, Rc=0
maclhwu.	RT, RA, RB	OE=0, Rc=1
maclhwuo	RT, RA, RB	OE=1, Rc=0
maclhwuo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	396	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ unsigned

$temp_{0:32} \leftarrow prod_{0:31} + (RT)$

$(RT) \leftarrow temp_{1:32}$

The low-order halfword of RA is multiplied by the low-order halfword of RB. The unsigned product is summed with the contents of RT and the sum is stored in a 33-bit temporary register. The contents of RT are replaced by the low-order 32 bits of the temporary register.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

mcrf BF, BFA

19	BF		BFA		0	
0	6	9	11	14	21	31

m ← BFA
n ← BF
(CR[CRn]) ← (CR[CRm])

The contents of the CR field specified by the BFA field are placed into the CR field specified by the BF field.

Registers Altered

- CR[CR*n*] where *n* is specified by the BF field.

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

mcrxr BF

31	BF		512	
0	6	9	21	31

n ← BF
(CR[CRn]) ← $XER_{0:3}$
$XER_{0:3}$ ← $^{4}0$

The contents of $XER_{0:3}$ are placed into the CR field specified by the BF field. $XER_{0:3}$ are then set to 0.

This transfer is positional, by bit number, so the mnemonics associated with each bit are changed. See Table 9-18 for clarification.

Table 9-18. Transfer Bit Mnemonic Assignment

Bit	XER Usage	CR Usage
0	SO	LT
1	OV	GT
2	CA	EQ
3	Reserved	SO

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR[CR*n*] where *n* is specified by the BF field.
- XER[SO, OV, CA]

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

mfcr RT

31	RT		19	
0	6	11	21	31

(RT) ← (CR)

The contents of the CR are placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

mfdcr RT, DCRN

31	RT	DCRF	323	
0	6	11	21	31

$DCRN \leftarrow DCRF_{5:9} \parallel DCRF_{0:4}$
$(RT) \leftarrow (DCR(DCRN))$

The contents of the DCR specified by the DCRF field are placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields
- Invalid DCRF values

Programming Note

Execution of this instruction is privileged.

The DCR number (DCRN) specified in the assembler language coding of **mfdcr** refers to a DCR number. The assembler handles the unusual register number encoding to generate the DCRF field.

Architecture Note

This instruction is implementation-specific and may not be portable to other implementations.

mfmsr RT

31	RT		83	
0	6	11	21	31

(RT) ← (MSR)

The contents of the MSR are placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged.

Architecture Note

This instruction is part of the PowerPC Embedded Operating Environment.

mfspr RT, SPRN

31	RT	SPRF	339	
0	6	11	21	31

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

The contents of the SPR specified by the SPRF field are placed into register RT. See *Special Purpose Registers* on page 354 for a listing of SPR mnemonics and corresponding SPRN and SPRF values.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields
- Invalid SPRF values

Programming Note

Execution of this instruction is privileged if instruction bit 11 contains 1. See *User and Supervisor Modes* on page 56.

The SPR number (SPRN) specified in the assembler language coding of **mfspr** refers to an SPR number (see *Special Purpose Registers* on page 354 for a list of SPRN values). The assembler handles the unusual register number encoding to generate the SPRF field. Also, see *Privileged SPRs* on page 57 for information about privileged SPRs.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-19. Extended Mnemonics for mfspr

Mnemonic	Operands	Function	Other Registers Changed
mfccr0 **mfctr** **mfdac1** **mfdac2** **mfdear** **mfdbcr0** **mfdbcr1** **mfdbsr** **mfdccr** **mfdcwr** **mfdvc1** **mfdvc2** **mfesr** **mfevpr** **mfiac1** **mfiac2** **mfiac3** **mfiac4** **mficcr** **mficdbdr** **mflr** **mfpid** **mfpit** **mfpvr** **mfsgr** **mfsler** **mfsprg0** **mfsprg1** **mfsprg2** **mfsprg3** **mfsprg4** **mfsprg5** **mfsprg6** **mfsprg7** **mfsrr0** **mfsrr1** **mfsrr2** **mfsrr3** **mfsu0r** **mftcr** **mftsr** **mfxer** **mfzpr**	**RT**	**Move from special purpose register SPRN.** *Extended mnemonic for* **mfspr RT,SPRN** **See *Special Purpose Registers* on page 354 for a list of valid SPRN values.**	

mftb RT, TBRN

31	RT	TBRF	371	
0	6	11	21	31

$$\text{TBRN} \leftarrow \text{TBRF}_{5:9} \parallel \text{TBRF}_{0:4}$$
$$(\text{RT}) \leftarrow (\text{TBR(TBRN)})$$

The contents of the time base register (TBR) specified by the TBRF field are placed into register RT. The following table lists the TBRN and TBRF values.

Table 9-20. Extended Mnemonics for mftb

Register Mnemonic	Register Name	TBRN		TBRF	Access
		Decimal	Hex		
TBL	Time Base Lower	268	0x10C	0x188	Read-only
TBU	Time Base Upper	269	0x10D	0x1A8	Read-only

If TBRN is a value other than those listed in the table, the results are **boundedly** undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields
- Invalid TBRF values

Programming Notes

The mnemonic **mftb** serves as both a hardware mnemonic and an extended mnemonic. The assembler recognizes an **mftb** mnemonic having two operands as the hardware form; an **mftb** mnemonic having one operand is recognized as the extended form.

The TBR number (TBRN) specified in the assembler language coding of the **mftb** instruction refers to a TBR number listed in the preceding table. The assembler handles the unusual register number encoding to generate the TBRF field.

Architecture Note

This instruction is part of the PowerPC Embedded Virtual Environment.

Table 9-21. Extended Mnemonics for mftb

Mnemonic	Operands	Function	Other Registers Altered
mftb	**RT**	**Move the contents of TBL into RT.** *Extended mnemonic for* **mftb RT,TBL**	
mftbu	**RT**	**Move the contents of TBU into RT.** *Extended mnemonic for* **mftb RT,TBU**	

mtcrf FXM, RS

31	RS		FXM		144	
0	6	11	12	20	21	31

$\text{mask} \leftarrow {}^4(\text{FXM}_0) \parallel {}^4(\text{FXM}_1) \parallel \ldots \parallel {}^4(\text{FXM}_6) \parallel {}^4(\text{FXM}_7)$
$(\text{CR}) \leftarrow ((\text{RS}) \wedge \text{mask}) \vee ((\text{CR}) \wedge \neg\text{mask})$

Some or all of the contents of register RS are placed into the CR as specified by the FXM field.

Each bit in the FXM field controls the copying of 4 bits in register RS into the corresponding bits in the CR. The correspondence between the bits in the FXM field and the bit copying operation is shown in the following table:

FXM Bit Number	Bits Controlled
0	0:3
1	4:7
2	8:11
3	12:15
4	16:19
5	20:23
6	24:27
7	28:31

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-22. Extended Mnemonics for mtcrf

Mnemonic	Operands	Function	Other Registers Altered
mtcr	**RS**	**Move to CR.** *Extended mnemonic for* **mtcrf 0xFF,RS**	

mtdcr DCRN, RS

31	RS	DCRF	451	
0	6	11	21	31

$DCRN \leftarrow DCRF_{5:9} \parallel DCRF_{0:4}$
$(DCR(DCRN)) \leftarrow (RS)$

The contents of register RS are placed into the DCR specified by the DCRF field.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- DCR(DCRN)

Invalid Instruction Forms

- Reserved fields
- Invalid DCRF values

Programming Note

Execution of this instruction is privileged.

The DCR number (DCRN) specified in the assembler language coding of **mtdcr** refers to a DCR number. The assembler handles the unusual register number encoding to generate the DCRF field.

Architecture Note

This instruction is implementation-specific and may not be portable to other implementations.

mtmsr RS

31	RS		146	
0	6	11	21	31

(MSR) ← (RS)

The contents of register RS are placed into the MSR.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- MSR

Invalid Instruction Forms

- Reserved fields

Programming Note

The **mtmsr** instruction is privileged and execution synchronizing.

Architecture Note

This instruction is part of the PowerPC Embedded Operating Environment.

mtspr SPRN, RS

31	RS	SPRF	467	
0	6	11	21	31

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

The contents of register RS are placed into register RT. See *Special Purpose Registers* on page 354 for a listing of SPR mnemonics and corresponding SPRN and SPRF values.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- SPR(SPRN)

Invalid Instruction Forms

- Reserved fields
- Invalid SPRF values

Programming Note

Execution of this instruction is privileged if instruction bit 11 is a 1. See *Privileged SPRs* on page 57 for more information.

The SPR number (SPRN) specified in the assembler language coding of the **mtspr** instruction refers to an SPR number (see *Special Purpose Registers* on page 354 for a list of SPRN values). The assembler handles the unusual register number encoding to generate the SPRF field.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-23. Extended Mnemonics for mtspr

Mnemonic	Operands	Function	Other Registers Altered
mtccr0 **mtctr** **mtdac1** **mtdac2** **mtdbcr0** **mtdbcr1** **mtdbsr** **mtdccr** **mtdcwr** **mtdear** **mtdvc1** **mtdvc2** **mtesr** **mtevpr** **mtiac1** **mtiac2** **mtiac3** **mtiac4** **mticcr** **mticdbdr** **mtlr** **mtpid** **mtpit** **mtpvr** **mtsgr** **mtsler** **mtsprg0** **mtsprg1** **mtsprg2** **mtsprg3** **mtsprg4** **mtsprg5** **mtsprg6** **mtsprg7** **mtsrr0** **mtsrr1** **mtsrr2** **mtsrr3** **mtsu0r** **mttcr** **mttsr** **mtxer** **mtzpr**	**RS**	**Move to special purpose register SPRN.** *Extended mnemonic for* **mtspr SPRN,RS** **See *Special Purpose Registers* on page 354 for a list of valid SPRN values.**	

mulchw	RT, RA, RB	Rc=0
mulchw.	RT, RA, RB	Rc=1

4	RT	RA	RB	168	Rc
0	6	11	16	21	31

$(RT)_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ signed

The low-order halfword of RA is multiplied by the high-order halfword of RB. The resulting signed product replaces the contents of RT.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

mulchwu	RT, RA, RB	Rc=0
mulchwu.	RT, RA, RB	Rc=1

4	RT	RA	RB	136	Rc
0	6	11	16	21	31

$(RT)_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ unsigned

The low-order halfword of RA is multiplied by the high-order halfword of RB. The resulting unsigned product replaces the contents of RT.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

mulhhw	RT, RA, RB	Rc=0
mulhhw.	RT, RA, RB	Rc=1

4	RT	RA	RB	40	Rc
0	6	11	16	21	31

$(RT)_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ signed

The high-order halfword of RA is multiplied by the high-order halfword of RB. The resulting signed product replaces the contents of RT.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

mulhhwu	RT, RA, RB	Rc=0
mulhhwu.	RT, RA, RB	Rc=1

4	RT	RA	RB	8	Rc
0	6	11	16	21	31

$(RT)_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ unsigned

The high-order halfword of RA is multiplied by the high-order halfword of RB. The resulting unsigned product replaces the contents of RT.

Registers Altered

- RT
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

mulhw	RT, RA, RB	Rc=0
mulhw.	RT, RA, RB	Rc=1

31	RT	RA	RB		75	Rc
0	6	11	16	21	22	31

$prod_{0:63} \leftarrow$ (RA) $\times$ (RB) signed
(RT) $\leftarrow prod_{0:31}$

The 64-bit signed product of registers RA and RB is formed. The most significant 32 bits of the result is placed into register RT.

Registers Altered

- RT
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc contains 1

Programming Note

The most significant 32 bits of the product, unlike the least significant 32 bits, may differ depending on whether the registers RA and RB are interpreted as signed or unsigned quantities. **mulhw** generates the correct result when these operands are interpreted as signed quantities. **mulhwu** generates the correct result when these operands are interpreted as unsigned quantities.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

mulhwu	RT, RA, RB	Rc=0
mulhwu.	RT, RA, RB	Rc=1

31	RT	RA	RB		11	Rc
0	6	11	16	21		31

$prod_{0:63} \leftarrow$ (RA) $\times$ (RB) unsigned
(RT) $\leftarrow prod_{0:31}$

The 64-bit unsigned product of registers RA and RB is formed. The most significant 32 bits of the result are placed into register RT.

Registers Altered

- RT
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc contains 1

Programming Note

The most significant 32 bits of the product, unlike the least significant 32 bits, may differ depending on whether the registers RA and RB are interpreted as signed or unsigned quantities. The **mulhw** instruction generates the correct result when these operands are interpreted as signed quantities. The **mulhwu** instruction generates the correct result when these operands are interpreted as unsigned quantities.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

mullhw	RT, RA, RB	Rc=0
mullhw.	RT, RA, RB	Rc=1

4	RT	RA	RB	424	Rc
0	6	11	16	21	31

$(RT)_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ signed

The low-order halfword of RA is multiplied by the low-order halfword of RB. The resulting signed product replaces the contents of RT.

Registers Altered

- RT
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

mullhwu	RT, RA, RB	OE=0, Rc=0
mullhwu.	RT, RA, RB	OE=0, Rc=1

4	RT	RA	RB	392	Rc
0	6	11	16	21	31

$(RT)_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ unsigned

The low-order halfword of RA is multiplied by the low-order halfword of RB. The resulting unsigned product replaces the contents of RT.

Registers Altered

- RT
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

mulli RT, RA, IM

7	RT	RA	IM
0	6	11	16 31

$prod_{0:47} \leftarrow (RA) \times EXTS(IM)$ signed
$(RT) \leftarrow prod_{16:47}$

The 48-bit product of register RA and the sign-extended IM field is formed. Both register RA and the IM field are interpreted as signed quantities. The least significant 32 bits of the product are placed into register RT.

Registers Altered

- RT

Programming Note

The least significant 32 bits of the product are correct, regardless of whether register RA and field IM are interpreted as signed or unsigned numbers.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

mullw	RT, RA, RB	OE=0, Rc=0
mullw.	RT, RA, RB	OE=0, Rc=1
mullwo	RT, RA, RB	OE=1, Rc=0
mullwo.	RT, RA, RB	OE=1, Rc=1

31	RT	RA	RB	OE	235	Rc
0	6	11	16	21	22	31

$prod_{0:63} \leftarrow$ (RA) $\times$ (RB) signed
(RT) $\leftarrow prod_{32:63}$

The 64-bit signed product of register RA and register RB is formed. The least significant 32 bits of the result is placed into register RT.

If the signed product cannot be represented in 32 bits and OE=1, XER[SO, OV] are set to 1.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE=1

Programming Note

The least significant 32 bits of the product are correct, regardless of whether register RA and register RB are interpreted as signed or unsigned numbers. The overflow indication is correct only if the operands are regarded as signed numbers.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

nand	RA, RS, RB	Rc=0
nand.	RA, RS, RB	Rc=1

31	RT	RA	RB	476	Rc
0	6	11	16	21	31

$(RA) \leftarrow \neg((RS) \land (RB))$

The contents of register RS is ANDed with the contents of register RB; the ones complement of the result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

neg	RT, RA	OE=0, Rc=0
neg.	RT, RA	OE=0, Rc=1
nego	RT, RA	OE=1, Rc=0
nego.	RT, RA	OE=1, Rc=1

31	RT	RA		OE	104	Rc
0	6	11	16	21	22	31

$(RT) \leftarrow \neg(RA) + 1$

The twos complement of the contents of register RA are placed into register RT.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE=1

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

nmacchw	RT, RA, RB	OE=0, Rc=0
nmacchw.	RT, RA, RB	OE=0, Rc=1
nmacchwo	RT, RA, RB	OE=1, Rc=0
nmacchwo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	174	Rc
0	6	11	16	21	22	31

$nprod_{0:31} \leftarrow -((RA)_{16:31} \times (RB)_{0:15})$ signed

$temp_{0:32} \leftarrow nprod_{0:31} + (RT)$

$(RT) \leftarrow temp_{1:32}$

The low-order halfword of RA is multiplied by the high-order halfword of RB. The negated signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register. The contents of RT are replaced by the low-order 32 bits of the temporary register.

Registers Altered

- RT
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

nmacchws	RT, RA, RB	OE=0, Rc=0
nmacchws.	RT, RA, RB	OE=0, Rc=1
nmacchwso	RT, RA, RB	OE=1, Rc=0
nmacchwso.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	238	Rc
0	6	11	16	21	22	31

$nprod_{0:31} \leftarrow -((RA)_{16:31} \times (RB)_{0:15}$ signed

$temp_{0:32} \leftarrow nprod_{0:31} + (RT)$

if $((nprod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \,\|\, {}^{31}(\neg RT_0))$

else $(RT) \leftarrow temp_{1:32}$

The low-order halfword of RA is multiplied by the high-order halfword of RB. The negated signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register.

If a result does not overflow, the low-order 32 bits of the temporary register are stored in RT.

If a result overflows, the returned result is the nearest representable value. Thus, if a result is less than -2^{31}, the value stored in RT is -2^{31}. Likewise, if a result is greater than $2^{31} - 1$, the value stored in RT is $2^{31} - 1$.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

nmachhw	RT, RA, RB	OE=0, Rc=0
nmachhw.	RT, RA, RB	OE=0, Rc=1
nmachhwo	RT, RA, RB	OE=1, Rc=0
nmachhwo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	46	Rc
0	6	11	16	21	22	31

$nprod_{0:31} \leftarrow -((RA)_{0:15} \times (RB)_{0:15})$ signed

$temp_{0:32} \leftarrow nprod_{0:31} + (RT)$

$(RT) \leftarrow temp_{1:32}$

The high-order halfword of RA is multiplied by the high-order halfword of RB. The negated signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register. The contents of RT are replaced by the low-order 32 bits of the temporary register.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

nmachhws	RT, RA, RB	OE=0, Rc=0
nmachhws.	RT, RA, RB	OE=0, Rc=1
nmachhwso	RT, RA, RB	OE=1, Rc=0
nmachhwso.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	110	Rc
0	6	11	16	21	22	31

$nprod_{0:31} \leftarrow -((RA)_{0:15} \times (RB)_{0:15})$ signed

$temp_{0:32} \leftarrow nprod_{0:31} + (RT)$

if $((nprod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$

else $(RT) \leftarrow temp_{1:32}$

The high-order halfword of RA is multiplied by the high-order halfword of RB. The negated signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register.

If a result does not overflow (i.e., it is accurately representable in 32 bits), the low-order 32 bits of the temporary register are stored in RT.

If a result overflows, the returned result is the nearest representable value. Thus, if a result is less than -2^{31}, the value stored in RT is -2^{31}. Likewise, if a result is greater than $2^{31} - 1$, the value stored in RT is $2^{31} - 1$.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

nmaclhw	RT, RA, RB	OE=0, Rc=0
nmaclhw.	RT, RA, RB	OE=0, Rc=1
nmaclhwo	RT, RA, RB	OE=1, Rc=0
nmachlwo.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	430	Rc
0	6	11	16	21	22	31

$nprod_{0:31} \leftarrow -((RA)_{16:31} \times (RB)_{16:31})$ signed

$temp_{0:32} \leftarrow nprod_{0:31} + (RT)$

$(RT) \leftarrow temp_{1:32}$

The low-order halfword of RA is multiplied by the low-order halfword of RB. The negated signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register. The contents of RT are replaced by the low-order 32 bits of the temporary register.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

nmaclhws	RT, RA, RB	OE=0, Rc=0
nmaclhws.	RT, RA, RB	OE=0, Rc=1
nmaclhwso	RT, RA, RB	OE=1, Rc=0
nmachlwso.	RT, RA, RB	OE=1, Rc=1

4	RT	RA	RB	OE	494	Rc
0	6	11	16	21	22	31

$nprod_{0:31} \leftarrow -((RA)_{16:31} \times (RB)_{16:31})$ signed

$temp_{0:32} \leftarrow nprod_{0:31} + (RT)$

if $((nprod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \,\|\, ^{31}(\neg RT_0))$

else $(RT) \leftarrow temp_{1:32}$

The low-order halfword of RA is multiplied by the low-order halfword of RB. The negated signed product is summed with the contents of RT and the sum is stored in a 33-bit temporary register.

If a result does not overflow, the low-order 32 bits of the temporary register are stored in RT.

If a result overflows, the returned result is the nearest representable value. Thus, if a result is less than -2^{31}, the value stored in RT is -2^{31}. Likewise, if a result is greater than $2^{31} - 1$, the value stored in RT is $2^{31} - 1$.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the Multiply-Accumulate instruction set extensions and complies with the architectural requirements for APUs of the PowerPC Embedded Environment. As such, it is not part of the PowerPC Architecture, nor is it part of the PowerPC Embedded Environment. Programs that use this instruction may not be portable to other implementations.

nor	RA, RS, RB	Rc=0
nor.	RA, RS, RB	Rc=1

31	RT	RA	RB	124	Rc
0	6	11	16	21	31

$(RA) \leftarrow \neg((RS) \vee (RB))$

The contents of register RS is ORed with the contents of register RB; the ones complement of the result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-24. Extended Mnemonics for nor, nor.

Mnemonic	Operands	Function	Other Registers Altered
not	**RA, RS**	**Complement register.** **(RA) ← ¬(RS)** *Extended mnemonic for* **nor RA,RS,RS**	
not.		*Extended mnemonic for* **nor. RA,RS,RS**	**CR[CR0]**

or	RA, RS, RB	Rc=0
or.	RA, RS, RB	Rc=1

31	RS	RA	RB	444	Rc
0	6	11	16	21	31

$(RA) \leftarrow (RS) \vee (RB)$

The contents of register RS is ORed with the contents of register RB; the result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-25. Extended Mnemonics for or, or.

Mnemonic	Operands	Function	Other Registers Altered
mr	**RT, RS**	**Move register.** **(RT) ← (RS)** *Extended mnemonic for* **or RT,RS,RS**	
mr.		*Extended mnemonic for* **or. RT,RS,RS**	**CR[CR0]**

orc	RA, RS, RB	Rc=0
orc.	RA, RS, RB	Rc=1

31	RT	RA	RB	412	Rc
0	6	11	16	21	31

$(RA) \leftarrow (RS) \vee \neg(RB)$

The contents of register RS is ORed with the ones complement of the contents of register RB; the result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

ori RA, RS, IM

24	RS	RA	IM
0	6	11	16 … 31

$$(RA) \leftarrow (RS) \vee (^{16}0 \parallel IM)$$

The IM field is extended to 32 bits by concatenating 16 0-bits on the left. Register RS is ORed with the extended IM field; the result is placed into register RA.

Registers Altered

- RA

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-26. Extended Mnemonics for ori

Mnemonic	Operands	Function	Other Registers Changed
nop		**Preferred no-op; triggers optimizations based on no-ops.** *Extended mnemonic for* **ori 0,0,0**	

oris RA, RS, IM

25	RS	RA	IM
0	6	11	16 … 31

$(RA) \leftarrow (RS) \vee (IM \parallel {}^{16}0)$

The IM Field is extended to 32 bits by concatenating 16 0-bits on the right. Register RS is ORed with the extended IM field and the result is placed into register RA.

Registers Altered

- RA

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

rfci

19		51	
0	6	21	31

(PC) ← (SRR2)
(MSR) ← (SRR3)

The program counter (PC) is restored with the contents of SRR2 and the MSR is restored with the contents of SRR3.

Instruction execution returns to the address contained in the PC.

Registers Altered

- MSR

Programming Note

Execution of this instruction is privileged and context-synchronizing.

Architecture Note

This instruction part of the PowerPC Embedded Operating Environment.

rfi

19		50	
0	6	21	31

(PC) ← (SRR0)
(MSR) ← (SRR1)

The program counter (PC) is restored with the contents of SRR0 and the MSR is restored with the contents of SRR1.

Instruction execution returns to the address contained in the PC.

Registers Altered

- MSR

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged and context-synchronizing.

Architecture Note

This instruction is part of the PowerPC Embedded Operating Environment.

rlwimi	RA, RS, SH, MB, ME	Rc=0
rlwimi.	RA, RS, SH, MB, ME	Rc=1

20	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

r $\leftarrow$ ROTL((RS), SH)
m $\leftarrow$ MASK(MB, ME)
(RA) $\leftarrow$ (r $\wedge$ m) $\vee$ ((RA) $\wedge$ $\neg$m)

The contents of register RS are rotated left by the number of bit positions specified in the SH field. A mask is generated, having 1-bits starting at the bit position specified in the MB field and ending in the bit position specified by the ME field, with 0-bits elsewhere.

If the starting point of the mask is at a higher bit position than the ending point, the 1-bits portion of the mask wraps from the highest bit position back around to the lowest. The rotated data is inserted into register RA, in positions corresponding to the bit positions in the mask that contain a 1-bit.

Registers Altered

- RA
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-27. Extended Mnemonics for rlwimi, rlwimi.

Mnemonic	Operands	Function	Other Registers Altered
inslwi	**RA, RS, n, b**	**Insert from left immediate (n > 0).** **(RA)**$_{b:b+n-1}$ $\leftarrow$ **(RS)**$_{0:n-1}$ *Extended mnemonic for* **rlwimi RA,RS,32−b,b,b+n−1**	
inslwi.		*Extended mnemonic for* **rlwimi. RA,RS,32−b,b,b+n−1**	**CR[CR0]**
insrwi	**RA, RS, n, b**	**Insert from right immediate. (n > 0)** **(RA)**$_{b:b+n-1}$ $\leftarrow$ **(RS)**$_{32-n:31}$ *Extended mnemonic for* **rlwimi RA,RS,32−b−n,b,b+n−1**	
insrwi.		*Extended mnemonic for* **rlwimi. RA,RS,32−b−n,b,b+n−1**	**CR[CR0]**

rlwinm	RA, RS, SH, MB, ME	Rc=0
rlwinm.	RA, RS, SH, MB, ME	Rc=1

21	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

$r \leftarrow$ ROTL((RS), SH)
$m \leftarrow$ MASK(MB, ME)
(RA) $\leftarrow r \wedge m$

The contents of register RS are rotated left by the number of bit positions specified in the SH field. A mask is generated, having 1-bits starting at the bit position specified in the MB field and ending in the bit position specified by the ME field with 0-bits elsewhere.

If the starting point of the mask is at a higher bit position than the ending point, the 1-bits portion of the mask wraps from the highest bit position back around to the lowest. The rotated data is ANDed with the generated mask; the result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-28. Extended Mnemonics for rlwinm, rlwinm.

Mnemonic	Operands	Function	Other Registers Altered
clrlwi	**RA, RS, n**	**Clear left immediate. (*n* < 32)** $(RA)_{0:n-1} \leftarrow {}^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,0,n,31**	
clrlwi.		*Extended mnemonic for* **rlwinm. RA,RS,0,n,31**	**CR[CR0]**
clrlslwi	**RA, RS, b, n**	**Clear left and shift left immediate.** **($n \leq b < 32$)** $(RA)_{b-n:31-n} \leftarrow (RS)_{b:31}$ $(RA)_{32-n:31} \leftarrow {}^{n}0$ $(RA)_{0:b-n-1} \leftarrow {}^{b-n}0$ *Extended mnemonic for* **rlwinm RA,RS,n,b−n,31−n**	
clrlslwi.		*Extended mnemonic for* **rlwinm. RA,RS,n,b−n,31−n**	**CR[CR0]**
clrrwi	**RA, RS, n**	**Clear right immediate. (*n* < 32)** $(RA)_{32-n:31} \leftarrow {}^{n}0$ ***Extended mnemonic* for** **rlwinm RA,RS,0,0,31−n**	
clrrwi.		*Extended mnemonic for* **rlwinm. RA,RS,0,0,31−n**	**CR[CR0]**

Table 9-28. Extended Mnemonics for rlwinm, rlwinm. (Continued)

Mnemonic	Operands	Function	Other Registers Altered
extlwi	**RA, RS, n, b**	**Extract and left justify immediate. (*n* > 0)** $(RA)_{0:n-1} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{n:31} \leftarrow {}^{32-n}0$ *Extended mnemonic for* **rlwinm RA,RS,b,0,n−1**	
extlwi.		*Extended mnemonic for* **rlwinm. RA,RS,b,0,n−1**	**CR[CR0]**
extrwi	**RA, RS, n, b**	**Extract and right justify immediate. (*n* > 0)** $(RA)_{32-n:31} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{0:31-n} \leftarrow {}^{32-n}0$ *Extended mnemonic for* **rlwinm RA,RS,b+n,32−n,31**	
extrwi.		*Extended mnemonic for* **rlwinm. RA,RS,b+n,32−n,31**	**CR[CR0]**
rotlwi	**RA, RS, n**	**Rotate left immediate.** **(RA) ← ROTL((RS), n)** *Extended mnemonic for* **rlwinm RA,RS,n,0,31**	
rotlwi.		*Extended mnemonic for* **rlwinm. RA,RS,n,0,31**	**CR[CR0]**
rotrwi	**RA, RS, n**	**Rotate right immediate.** **(RA) ← ROTL((RS), 32−n)** *Extended mnemonic for* **rlwinm RA,RS,32−n,0,31**	
rotrwi.		*Extended mnemonic for* **rlwinm. RA,RS,32−n,0,31**	**CR[CR0]**
slwi	**RA, RS, n**	**Shift left immediate. (*n* < 32)** $(RA)_{0:31-n} \leftarrow (RS)_{n:31}$ $(RA)_{32-n:31} \leftarrow {}^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,n,0,31−n**	
slwi.		*Extended mnemonic for* **rlwinm. RA,RS,n,0,31−n**	**CR[CR0]**
srwi	**RA, RS, n**	**Shift right immediate. (*n* < 32)** $(RA)_{n:31} \leftarrow (RS)_{0:31-n}$ $(RA)_{0:n-1} \leftarrow {}^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,32−n,n,31**	
srwi.		*Extended mnemonic for* **rlwinm. RA,RS,32−n,n,31**	**CR[CR0]**

rlwnm	RA, RS, RB, MB, ME	Rc=0
rlwnm.	RA, RS, RB, MB, ME	Rc=1

23	RS	RA	RB	MB	ME	Rc
0	6	11	16	21	26	31

$r \leftarrow$ ROTL((RS), $(RB)_{27:31}$)
$m \leftarrow$ MASK(MB, ME)
(RA) $\leftarrow r \wedge m$

The contents of register RS are rotated left by the number of bit positions specified by the contents of register $RB_{27:31}$. A mask is generated, having 1-bits starting at the bit position specified in the MB field and ending in the bit position specified by the ME field with 0-bits elsewhere.

If the starting point of the mask is at a higher bit position than the ending point, the ones portion of the mask wraps from the highest bit position back to the lowest. The rotated data is ANDed with the generated mask and the result is placed into register RA.

Registers Altered

- RA
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-29. Extended Mnemonics for rlwnm, rlwnm.

Mnemonic	Operands	Function	Other Registers Altered
rotlw	**RA, RS, RB**	**Rotate left.** **(RA) $\leftarrow$ ROTL((RS), $(RB)_{27:31}$)** *Extended mnemonic for* **rlwnm RA,RS,RB,0,31**	
rotlw.		*Extended mnemonic for* **rlwnm. RA,RS,RB,0,31**	**CR[CR0]**

sc

0–5	6–29	30	31
17		1	

$(SRR1) \leftarrow (MSR)$
$(SRR0) \leftarrow (PC)$
$PC \leftarrow EVPR_{0:15} \parallel 0x0C00$
$(MSR[WE, EE, PR, DR, IR]) \leftarrow 0$

A system call exception is generated. The contents of the MSR are copied into SRR1 and (4 + address of **sc** instruction) is placed into SRR0.

The program counter (PC) is then loaded with the exception vector address. The exception vector address is calculated by concatenating the high halfword of the Exception Vector Prefix Register (EVPR) to the left of 0x0C00.

The MSR[WE, EE, PR, DR, IR] bits are set to 0.

Program execution continues at the new address in the PC.

The **sc** instruction is context synchronizing.

Registers Altered

- SRR0
- SRR1
- MSR[WE, EE, PR, DR, IR]

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

slw	RA, RS, RB	Rc=0
slw.	RA, RS, RB	Rc=1

31	RS	RA	RB	24	Rc
0	6	11	16	21	31

$n \leftarrow (RB)_{27:31}$
$r \leftarrow ROTL((RS), n)$
if $(RB)_{26} = 0$ then
  $m \leftarrow MASK(0, 31 - n)$
else
  $m \leftarrow {}^{32}0$
$(RA) \leftarrow r \land m$

The contents of register RS are shifted left by the number of bits specified by the contents of register $RB_{27:31}$. Bits shifted left out of the most significant bit are lost, and 0-bits fill vacated bit positions on the right. The result is placed into register RA.

If RB_{26} = 1, register RA is set to zero.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sraw	RA, RS, RB	Rc=0
sraw.	RA, RS, RB	Rc=1

31	RS	RA	RB	792	Rc
0	6	11	16	21	31

$n \leftarrow (RB)_{27:31}$
$r \leftarrow ROTL((RS), 32 - n)$
if $(RB)_{26} = 0$ then
    $m \leftarrow MASK(n, 31)$
else
    $m \leftarrow {}^{32}0$
$s \leftarrow (RS)_0$
$(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$
$XER[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$

The contents of register RS are shifted right by the number of bits specified the contents of register $RB_{27:31}$. Bits shifted out of the least significant bit are lost. Register RS_0 is replicated to fill the vacated positions on the left. The result is placed into register RA.

If register RS contains a negative number and any 1-bits were shifted out of the least significant bit position, XER[CA] is set to 1; otherwise, it is set to 0.

If bit 26 of register RB contains 1, register RA and XER[CA] are set to bit 0 of register RS.

Registers Altered

- RA
- XER[CA]
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

srawi	RA, RS, SH	Rc=0
srawi.	RA, RS, SH	Rc=1

31	RS	RA	SH	824	Rc
0	6	11	16	21	31

n ← SH
r ← ROTL((RS), 32 – n)
m ← MASK(n, 31)
s ← $(RS)_0$
(RA) ← $(r \land m) \lor (^{32}s \land \neg m)$
XER[CA] ← $s \land ((r \land \neg m) \neq 0)$

The contents of register RS are shifted right by the number of bits specified in the SH field. Bits shifted out of the least significant bit are lost. Bit RS_0 is replicated to fill the vacated positions on the left. The result is placed into register RA.

If register RS contains a negative number and any 1-bits were shifted out of the least significant bit position, XER[CA] is set to 1; otherwise, it is set to 0.

Registers Altered

- RA
- XER[CA]
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

srw	RA, RS, RB	Rc=0
srw.	RA, RS, RB	Rc=1

31	RS	RA	RB	536	Rc
0	6	11	16	21	31

n $\leftarrow$ $(RB)_{27:31}$
r $\leftarrow$ ROTL((RS), 32 – n)
if $(RB)_{26}$ = 0 then
  m $\leftarrow$ MASK(n, 31)
else
  m $\leftarrow$ $^{32}0$
(RA) $\leftarrow$ r $\wedge$ m

The contents of register RS are shifted right by the number of bits specified the contents of register $RB_{27:31}$. Bits shifted right out of the least significant bit are lost, and 0-bits fill the vacated bit positions on the left. The result is placed into register RA.

If bit 26 of register RB contains a one, register RA is set to 0.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stb RS, D(RA)

38	RS	RA	D
0	6	11	16 31

$EA \leftarrow (RA|0) + EXTS(D)$
$MS(EA, 1) \leftarrow (RS)_{24:31}$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant byte of register RS is stored into the byte at the EA.

Registers Altered

- None

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stbu RS, D(RA)

39	RS	RA	D
0	6	11	16 … 31

$EA \leftarrow (RA) + EXTS(D)$
$MS(EA, 1) \leftarrow (RS)_{24:31}$
$(RA) \leftarrow EA$

An effective address (EA) is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The EA is placed into register RA.

The least significant byte of register RS is stored into the byte at the EA.

Registers Altered

- RA

Invalid Instruction Forms

RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stbux RS, RA, RB

31	RS	RA	RB	247	
0	6	11	16	21	31

$EA \leftarrow (RA) + (RB)$
$MS(EA, 1) \leftarrow (RS)_{24:31}$
$(RA) \leftarrow EA$

An effective address (EA) is formed by adding an index to the base address in register RA. The index is the contents of register RB. The EA is placed into register RA.

The least significant byte of register RS is stored into the byte at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA

Invalid Instruction Forms

- Reserved fields
- RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stbx RS, RA, RB

31	RS	RA	RB	215	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
MS(EA, 1) $\leftarrow$ $(RS)_{24:31}$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant byte of register RS is stored into the byte at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sth RS, D(RA)

44	RS	RA	D
0	6	11	16 … 31

$EA \leftarrow (RA|0) + EXTS(D)$
$MS(EA, 2) \leftarrow (RS)_{16:31}$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0 and is the contents of register RA otherwise.

The least significant halfword of register RS is stored into the halfword at the EA in main storage.

Registers Altered

- None

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sthbrx RS, RA, RB

31	RS	RA	RB	918	
0	6	11	16	21	31

$EA \leftarrow (RA|0) + (RB)$
$MS(EA, 2) \leftarrow (RS)_{24:31} \parallel (RS)_{16:23}$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant halfword of register RS is byte-reversed. The result is stored into the halfword at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sthu RS, D(RA)

45	RS	RA	D
0	6	11	16 31

$EA \leftarrow (RA) + EXTS(D)$
$MS(EA, 2) \leftarrow (RS)_{16:31}$
$(RA) \leftarrow EA$

An effective address (EA) is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The EA is placed into register RA.

The least significant halfword of register RS is stored into the halfword at the EA.

Registers Altered

- RA

Invalid Instruction Forms

- RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sthux RS, RA, RB

31	RS	RA	RB	439	
0	6	11	16	21	31

$EA \leftarrow (RA) + (RB)$
$MS(EA, 2) \leftarrow (RS)_{16:31}$
$(RA) \leftarrow EA$

An effective address (EA) is formed by adding an index to the base address in register RA. The index is the contents of register RB. The EA is placed into register RA.

The least significant halfword of register RS is stored into the halfword at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA

Invalid Instruction Forms

- Reserved fields
- RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sthx RS, RA, RB

31	RS	RA	RB	407	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
MS(EA, 2) $\leftarrow$ $(RS)_{16:31}$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant halfword of register RS is stored into the halfword at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stmw RS, D(RA)

47	RS	RA	D
0	6	11	16 … 31

$EA \leftarrow (RA|0) + EXTS(D)$
$r \leftarrow RS$
do while $r \leq 31$
  $MS(EA, 4) \leftarrow (GPR(r))$
  $r \leftarrow r + 1$
  $EA \leftarrow EA + 4$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The contents of a series of consecutive registers, starting with register RS and continuing through GPR(31), are stored into consecutive words starting at the EA.

Registers Altered

- None

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stswi RS, RA, NB

31	RS	RA	NB	725	
0	6	11	16	21	31

```
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
```

An effective address (EA) is determined by the RA field. If the RA field contains 0, the EA is 0; otherwise, the EA is the contents of register RA.

A byte count is determined by the NB field. If the NB field contains 0, the byte count is 32; otherwise, the byte count is the contents of the NB field.

The contents of a series of consecutive GPRs (starting with register RS, continuing through GPR(31), wrapping to GPR(0), and continuing to the final byte count) are stored, starting at the EA. The bytes in each GPR are accessed starting with the most significant byte. The byte count determines the number of transferred bytes.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stswx RS, RA, RB

31	RS	RA	RB	661	
0	6	11	16	21	31

```
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
```

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

A byte count is contained in XER[TBC].

The contents of a series of consecutive GPRs (starting with register RS, continuing through GPR(31), wrapping to GPR(0), and continuing to the final byte count) are stored, starting at the EA. The bytes in each GPR are accessed starting with the most significant byte. The byte count determines the number of transferred bytes.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

If XER[TBC] = 0, **stswx** is treated as a no-op.

The PowerPC Architecture states that if XER[TBC] = 0 and if the EA is such that a precise data exception would normally occur (if not for the zero length), **stswx** is treated as a no-op and the precise exception will not occur. Data storage exceptions and alignment exceptions are examples of precise data exceptions.

However, the architecture makes no statement regarding imprecise exceptions related to **stswx** when XER[TBC] = 0. PowerPC processors generate an imprecise exception (machine check) on this instruction when all of the following conditions are true:

- The instruction passes all protection bounds checking
- The address is cacheable

- The address is passed to the data cache
- The address misses in the data cache (resulting in a line fill request)
- The address encounters some form of bus error (non-configured, for example)

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stw RS, D(RA)

36	RS	RA	D
0	6	11	16 ... 31

EA $\leftarrow$ (RA|0) + EXTS(D)
MS(EA, 4) $\leftarrow$ (RS)

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The contents of register RS are stored at the EA.

Registers Altered

- None

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stwbrx RS, RA, RB

31	RS	RA	RB	662	
0	6	11	16	21	31

$EA \leftarrow (RA|0) + (RB)$
$MS(EA, 4) \leftarrow (RS)_{24:31} \parallel (RS)_{16:23} \parallel (RS)_{8:15} \parallel (RS)_{0:7}$

An EA is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The contents of register RS are byte-reversed: the least significant byte becomes the most significant byte, the next least significant byte becomes the next most significant byte, and so on. The result is stored into the word at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stwcx. RS, RA, RB

31	RS	RA	RB	150	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
if RESERVE = 1 then
  MS(EA, 4) $\leftarrow$ (RS)
  RESERVE $\leftarrow$ 0
  (CR[CR0]) $\leftarrow$ $^{2}0 \parallel 1 \parallel XER_{SO}$
else
  (CR[CR0]) $\leftarrow$ $^{2}0 \parallel 0 \parallel XER_{SO}$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

If the reservation bit contains 1 when the instruction is executed, the contents of register RS are stored into the word at the EA and the reservation bit is cleared. If the reservation bit contains 0 when the instruction is executed, no store operation is performed.

CR[CR0] is set as follows:

- $CR[CR0]_{LT, GT}$ are cleared
- $CR[CR0]_{EQ}$ is set to the state of the reservation bit at the start of the instruction
- $CR[CR0]_{SO}$ is set to the contents of the XER[SO] bit

Registers Altered

- $CR[CR0]_{LT, GT, EQ, SO}$

Programming Note

lwarx and the **stwcx.** instruction should paired in a loop, as shown in the following example, to create the effect of an atomic operation to a memory area used as a semaphore between asynchronous processes. Only **lwarx** can set the reservation bit to 1. **stwcx.** sets the reservation bit to 0 upon its completion, whether or not **stwcx.** sent (RS) to memory. $CR[CR0]_{EQ}$ must be examined to determine whether (RS) was sent to memory.

```
loop: lwarx   # read the semaphore from memory; set reservation
“alter”       # change the semaphore bits in register as required
stwcx.        # attempt to store semaphore; reset reservation
bne loop      # an asynchronous process has intervened; try again
```

If the asynchronous process in the code example had paired **lwarx** with a store other than **stwcx.**, the reservation bit would not have been cleared in the asynchronous process, and the code example would have overwritten the semaphore.

Exceptions

An alignment exception occurs if the EA is not word-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stwu RS, D(RA)

37	RS	RA	D
0	6	11	16 … 31

EA $\leftarrow$ (RA) + EXTS(D)
MS(EA, 4) $\leftarrow$ (RS)
(RA) $\leftarrow$ EA

An effective address (EA) is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The EA is placed into register RA.

The contents of register RS are stored into the word at the EA.

Registers Altered

- RA

Invalid Instruction Forms

- RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stwux RS, RA, RB

31	RS	RA	RB	183	
0	6	11	16	21	31

EA $\leftarrow$ (RA) + (RB)
MS(EA, 4) $\leftarrow$ (RS)
(RA) $\leftarrow$ EA

An effective address (EA) is formed by adding an index to the base address in register RA. The index is the contents of register RB. The EA is placed into register RA.

The contents of register RS are stored into the word at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA

Invalid Instruction Forms

- Reserved fields
- RA = 0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stwx RS, RA, RB

31	RS	RA	RB	151	
0	6	11	16	21	31

EA $\leftarrow$ (RA|0) + (RB)
MS(EA,4) $\leftarrow$ (RS)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The contents of register RS are stored into the word at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

subf	RT, RA, RB	OE=0, Rc=0
subf.	RT, RA, RB	OE=0, Rc=1
subfo	RT, RA, RB	OE=1, Rc=0
subfo.	RT, RA, RB	OE=1, Rc=1

31	RT	RA	RB	OE	40	Rc
0	6	11	16	21	22	31

(RT) $\leftarrow$ $\neg$(RA) + (RB) + 1

The sum of the ones complement of register RA, register RB, and 1 is stored into register RT.

Registers Altered

- RT
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-30. Extended Mnemonics for subf, subf., subfo, subfo.

Mnemonic	Operands	Function	Other Registers Altered
sub	**RT, RA, RB**	**Subtract (RB) from (RA).** **(RT) $\leftarrow$ $\neg$(RB) + (RA) + 1.** *Extended mnemonic for* **subf RT,RB,RA**	
sub.		*Extended mnemonic for* **subf. RT,RB,RA**	**CR[CR0]**
subo		*Extended mnemonic for* **subfo RT,RB,RA**	**XER[SO, OV]**
subo.		*Extended mnemonic for* **subfo. RT,RB,RA**	**CR[CR0]** **XER[SO, OV]**

subfc	RT, RA, RB	OE=0, Rc=0
subfc.	RT, RA, RB	OE=0, Rc=1
subfco	RT, RA, RB	OE=1, Rc=0
subfco.	RT, RA, RB	OE=1, Rc=1

31	RT	RA	RB	OE	8	Rc
0	6	11	16	21	22	31

(RT) ← ¬(RA) + (RB) + 1
if ¬(RA) + (RB) + 1 $\overset{u}{>}$ 2^{32} – 1 then
    XER[CA] ← 1
else
    XER[CA] ← 0

The sum of the ones complement of register RA, register RB, and 1 is stored into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Registers Altered

- RT
- XER[CA]
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-31. Extended Mnemonics for subfc, subfc., subfco, subfco.

Mnemonic	Operands	Function	Other Registers Altered
subc	**RT, RA, RB**	**Subtract (RB) from (RA).** **(RT) ← ¬(RB) + (RA) + 1.** **Place carry-out in XER[CA].** *Extended mnemonic for* **subfc RT,RB,RA**	
subc.		*Extended mnemonic for* **subfc. RT,RB,RA**	**CR[CR0]**
subco		*Extended mnemonic for* **subfco RT,RB,RA**	**XER[SO, OV]**
subco.		*Extended mnemonic for* **subfco. RT,RB,RA**	**CR[CR0]** **XER[SO, OV]**

subfe	RT, RA, RB	OE=0, Rc=0
subfe.	RT, RA, RB	OE=0, Rc=1
subfeo	RT, RA, RB	OE=1, Rc=0
subfeo.	RT, RA, RB	OE=1, Rc=1

31	RT	RA	RB	OE	136	Rc
0	6	11	16	21	22	31

(RT) $\leftarrow$ $\neg$(RA) + (RB) + XER[CA]
if $\neg$(RA) + (RB) + XER[CA] $\overset{u}{>}$ $2^{32} - 1$ then
  XER[CA] $\leftarrow$ 1
else
  XER[CA] $\leftarrow$ 0

The sum of the ones complement of register RA, register RB, and XER[CA] is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

subfic RT, RA, IM

8	RT	RA	IM
0	6	11	16 … 31

(RT) ← ¬(RA) + EXTS(IM) + 1
if ¬(RA) + EXTS(IM) + 1 $\overset{u}{>}$ 2^{32} – 1 then
  XER[CA] ← 1
else
  XER[CA] ← 0

The sum of the ones complement of RA, the IM field sign-extended to 32 bits, and 1 is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Registers Altered

- RT
- XER[CA]

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

subfme	RT, RA	OE=0, Rc=0
subfme.	RT, RA	OE=0, Rc=1
subfmeo	RT, RA	OE=1, Rc=0
subfmeo.	RT, RA	OE=1, Rc=1

31	RT	RA		OE	232	Rc
0	6	11	16	21	22	31

$(RT) \leftarrow \neg(RA) - 1 + XER[CA]$
if $\neg(RA) + \text{0xFFFF FFFF} + XER[CA] \overset{u}{>} 2^{32} - 1$ then
  $XER[CA] \leftarrow 1$
else
  $XER[CA] \leftarrow 0$

The sum of the ones complement of register RA, –1, and XER[CA] is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1
- XER[CA]

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

subfze	RT, RA	OE=0, Rc=0
subfze.	RT, RA	OE=0, Rc=1
subfzeo	RT, RA	OE=1, Rc=0
subfzeo.	RT, RA	OE=1, Rc=1

31	RT	RA		OE	200	Rc
0	6	11	16	21	22	31

(RT) $\leftarrow$ $\neg$(RA) + XER[CA]
if $\neg$(RA) + XER[CA] $\overset{u}{>}$ $2^{32} - 1$ then
XER[CA] $\leftarrow$ 1
else
XER[CA] $\leftarrow$ 0

The sum of the ones complement of register RA and XER[CA] is stored into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains 1
- XER[SO, OV] if OE contains 1

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sync

0–5	6–20	21–30	31
31		**598**	

The **sync** instruction guarantees that all instructions initiated by the processor preceding **sync** will complete before **sync** completes, and that no subsequent instructions will be initiated by the processor until after **sync** completes. When **sync** completes, all storage accesses that were initiated by the processor before the **sync** instruction will have been completed with respect to all mechanisms that access storage.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None.

Invalid Instruction Forms

- Reserved fields

Programming Note

Architecturally, the **eieio** instruction orders storage access, not instruction completion. Therefore, non-storage operations that follow **eieio** could complete before storage operations that precede **eieio**. The **sync** instruction guarantees ordering of instruction completion and storage access. For the PPC405, the **eieio** instruction is implemented to behave as a **sync** instruction.

To write code that is portable between various PowerPC implementations, programmers should use the mnemonic that corresponds to the desired behavior.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

tlbia

31		370	
0	6	21	31

All of the entries in the TLB are invalidated and become unavailable for translation by clearing the valid (V) bit in the TLBHI portion of each TLB entry. The rest of the fields in the TLB entries are unmodified.

Registers Altered

- None.

Invalid Instruction Forms

- None.

Programming Note

This instruction is privileged. Translation is not required to be active during the execution of this instruction. The effects of the invalidation are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation.

Architecture Note

This instruction is part of the PowerPC Embedded Operating Environment.

tlbre RT, RA, WS

31	RT	RA	WS	946	
0	6	11	16	21	31

if WS_4 = 1
 (RT) ← TLBLO[($RA_{26:31}$)]
else
 (RT) ← TLBHI[($RA_{26:31}$)]
 (PID) ← TID from TLB[($RA_{26:31}$)]

The contents of the selected TLB entry is placed into register RT (and possibly into PID).

Bits 26:31 of the contents of RA is used as an index into the TLB. If this index specifies a TLB entry that does not exist, the results are undefined.

The WS field specifies which portion (TLBHI or TLBLO) of the entry is loaded into RT. If TLBHI is being accessed, the PID SPR is set to the value of the TID field in the TLB entry.

If the WS field is not 0 or 1, the instruction form is invalid and the result is undefined.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT
- PID (if WS = 0)

Invalid Instruction Forms

- Reserved fields
- Invalid WS value

Programming Notes

This instruction is privileged. Translation is not required to be active during the execution of this instruction.

The contents of RT after the execution of this instruction are interpreted as follows:
If WS = 0 (TLBHI):
 RT[0:21] ← EPN[0:21]
 RT[22:24] ← SIZE[0:2]
 RT[25] ← V
 RT[26] ← E
 RT[27] ← U0
 RT[28:31] ← 0
 PID[24:31] ← TID[0:7]; (note that the TID is copied to the PID, not to RT)
If WS = 1 (TLBLO):
 RT[0:21] ← RPN[0:21]
 RT[22:23] ← EX,WR
 RT[24:27] ← ZSEL[0:3]
 RT[28:31] ← WIMG

Architecture Note

This instruction part of the PowerPC Embedded Operating Environment.

Table 9-32. Extended Mnemonics for tlbre

Mnemonic	Operands	Function	Other Registers Altered
tlbrehi	**RT, RA**	**Load TLBHI portion of the selected TLB entry into RT.** **Load the PID register with the contents of the TID field of the selected TLB entry.** **(RT) $\leftarrow$ TLBHI[(RA)]** **(PID) $\leftarrow$ TLB[(RA)]$_{TID}$** *Extended mnemonic for* **tlbre RT,RA,0**	
tlbrelo	**RT, RA**	**Load TLBLO portion of the selected TLB entry into RT.** **(RT) $\leftarrow$ TLBLO[(RA)]** *Extended mnemonic for* **tlbre RT,RA,1**	

tlbsx	RT, RA, RB	Rc=0
tlbsx.	RT, RA, RB	Rc=1

31	RT	RA	RB	914	Rc
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
if Rc = 1
    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
    (RT) ← Index of matching TLB Entry
    if Rc = 1
        CR[CR0]EQ ← 1
else
    (RT) Undefined
    if Rc = 1
        CR[CR0]EQ ← 0
```

An effective address is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The TLB is searched for a valid entry which translates EA and PID. See XREF for details. The record bit (Rc) specifies whether the results of the search will affect CR[CR0] as shown above. The intention is that $CR[CR0]_{EQ}$ can be tested after a **tlbsx.** instruction if there is a possibility that the search may fail.

Registers Altered

- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Invalid Instruction Forms

- None.

Programming Note

This instruction is privileged. Translation is not required to be active during the execution of this instruction.

Architecture Note

This instruction part of the PowerPC Embedded Operating Environment.

tlbsync

31		566	
0	6	21	31

The **tlbsync** instruction is provided in the PowerPC architecture to support synchronization of TLB operations among the processors of a multi-processor system. In the PPC405, this instruction performs no operation, and is provided to facilitate code portability.

Registers Altered

- None.

Invalid Instruction Forms

- None.

Programming Notes

This instruction is privileged. Translation is not required to be active during the execution of this instruction.

Since the PPC405 does not support tightly-coupled multiprocessor systems, **tlbsync** performs no operation.

Architecture Note

This instruction is part of the PowerPC Embedded Operating Environment.

tlbwe RS, RA, WS

31	RS	RA	WS	978	
0	6	11	16	21	31

if WS_4 = 1
  TLBLO[($RA_{26:31}$)] ← (RS)
else
  TLBHI[($RA_{26:31}$)] ← (RS)
  TID of TLB[($RA_{26:31}$)] ← ($PID_{24:31}$)

The contents of the selected TLB entry is replaced with the contents of register RS (and possibly PID).

Bits 26:31 of the contents of RA are used as an index into the TLB. If this index specifies a TLB entry that does not exist, the results are undefined.

The WS field specifies which portion (TLBHI or TLBLO) of the entry is replaced from RS. For instructions that specify TLBHI, the TID field in the TLB entry is supplied from $PID_{24:31}$.

If the WS field is not 0 or 1, the instruction form is invalid and the result is undefined.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None.

Invalid Instruction Forms

- Reserved fields
- Invalid WS value

Programming Notes

This instruction is privileged. Translation is not required to be active during the execution of this instruction.

The effects of this update are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation. For example, updating a zone selection field within the TLB while in supervisor code should be followed by an **isync** instruction (or other context synchronizing operation) to guarantee that the desired translation and protection domains are used.

tlbwe writes the TLB fields from RS and the PID as follows:

If WS = 0 (TLBHI):
  EPN[0:21] ← RS[0:21]
  SIZE[0:2] ← RS[22:24]
  V ← RS[25]
  E ← RS[26]
  U0 ← RS[27]
  TID[0:7] ← PID[24:31]; (note that the TID is written from the PID, not RS)
If WS = 1 (TLBLO):
  RPN[0:21] ← RT[0:21]
  EX,WR ← RS[22:23]
  ZSEL[0:3] ← RS[24:27]
  WIMG ← RS[28:31]

Architecture Note

This instruction part of the PowerPC Embedded Operating Environment.

Table 9-33. Extended Mnemonics for tlbwe

Mnemonic	Operands	Function	Other Registers Altered
tlbwehi	RS, RA	Write TLBHI portion of the selected TLB entry from RS. Write the TID register of the selected TLB entry from the PID register. TLBHI[(RA)] $\leftarrow$ (RS) $TLB[(RA)]_{TID} \leftarrow (PID_{24:31})$ *Extended mnemonic for* **tlbwe RS,RA,0**	
tlbwelo	RS, RA	Write TLBLO portion of the selected TLB entry from RS. TLBLO[(RA)] $\leftarrow$ (RS) *Extended mnemonic for* **tlbwe RS,RA,1**	

tw TO, RA, RB

31	TO	RA	RB	4	
0	6	11	16	21	31

$$
\begin{array}{lll}
\text{if (} & ((RA) < (RB) \wedge TO_0 = 1) & \vee \\
 & ((RA) > (RB) \wedge TO_1 = 1) & \vee \\
 & ((RA) = (RB) \wedge TO_2 = 1) & \vee \\
 & ((RA) \overset{u}{<} (RB) \wedge TO_3 = 1) & \vee \\
 & ((RA) \overset{u}{>} (RB) \wedge TO_4 = 1) & \text{) then TRAP (see details below)}
\end{array}
$$

Register RA is compared with register RB. If any comparison condition selected by the TO field is true, a TRAP occurs. The behavior of a TRAP depends upon the debug mode of the processor, as described below:

- If TRAP is not enabled as a debug event (DBCR[TDE] = 0 or DBCR[EDM,IDM] = 0,0):

TRAP causes a program interrupt. See *Program Interrupt* on page 123.

(SRR0) $\leftarrow$ address of **tw** instruction
(SRR1) $\leftarrow$ (MSR)
(ESR[PTR]) $\leftarrow$ 1
(MSR[WE, EE, PR, DR, IR]) $\leftarrow$ 0
PC $\leftarrow$ $EVPR_{0:15}$ || 0x0700

- If TRAP is enabled as an external debug event (DBCR[TDE] = 1 and DBCR[EDM] = 1):

TRAP goes to the debug stop state, to be handled by an external debugger with hardware control.
(DBSR[TIE]) $\leftarrow$ 1

In addition, if TRAP is also enabled as an internal debug event (DBCR[IDM] = 1) and debug exceptions are disabled (MSR[DE] = 0), then report an imprecise event:
(DBSR[IDE]) $\leftarrow$ 1
PC $\leftarrow$ address of **tw** instruction

- If TRAP is enabled as an internal debug event and *not* an external debug event (DBCR[TDE] = 1 and DBCR[EDM,IDM] = 0,1) and debug exceptions are enabled (MSR[DE] = 1):

TRAP causes a debug interrupt. See *Debug Interrupt* on page 128.
(SRR2) $\leftarrow$ address of **tw** instruction
(SRR3) $\leftarrow$ (MSR)
(DBSR[TIE]) $\leftarrow$ 1
(MSR[WE, EE, PR, CE, DE, DR, IR]) $\leftarrow$ 0
PC $\leftarrow$ $EVPR_{0:15}$ || 0x2000

- If TRAP is enabled as an internal debug event and *not* an external debug event (DBCR[TDE] = 1 and DBCR[EDM,IDM] = 0,1) and Debug Exceptions are disabled (MSR[DE] = 0):

TRAP reports the debug event as an *imprecise* event and causes a program interrupt. See *Program Interrupt* on page 123.
(SRR0) $\leftarrow$ address of **tw** instruction
(SRR1) $\leftarrow$ (MSR)
(ESR[PTR]) $\leftarrow$ 1
(DBSR[TIE,IDE]) $\leftarrow$ 1,1
(MSR[WE, EE, PR, DR, IR]) $\leftarrow$ 0
PC $\leftarrow$ $EVPR_{0:15}$ || 0x0700

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

This instruction is inserted into the execution stream by a debugger to implement breakpoints, and is not typically used by application code.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-34. Extended Mnemonics for tw

Mnemonic	Operands	Function	Other Registers Altered
trap		**Trap unconditionally.** *Extended mnemonic for* **tw 31,0,0**	
tweq	**RA, RB**	**Trap if (RA) equal to (RB).** *Extended mnemonic for* **tw 4,RA,RB**	
twge	**RA, RB**	**Trap if (RA) greater than or equal to (RB).** *Extended mnemonic for* **tw 12,RA,RB**	
twgt	**RA, RB**	**Trap if (RA) greater than (RB).** *Extended mnemonic for* **tw 8,RA,RB**	
twle	**RA, RB**	**Trap if (RA) less than or equal to (RB).** *Extended mnemonic for* **tw 20,RA,RB**	
twlge	**RA, RB**	**Trap if (RA) logically greater than or equal to (RB).** *Extended mnemonic for* **tw 5,RA,RB**	
twlgt	**RA, RB**	**Trap if (RA) logically greater than (RB).** *Extended mnemonic for* **tw 1,RA,RB**	
twlle	**RA, RB**	**Trap if (RA) logically less than or equal to (RB).** *Extended mnemonic for* **tw 6,RA,RB**	
twllt	**RA, RB**	**Trap if (RA) logically less than (RB).** *Extended mnemonic for* **tw 2,RA,RB**	
twlng	**RA, RB**	**Trap if (RA) logically not greater than (RB).** *Extended mnemonic for* **tw 6,RA,RB**	
twlnl	**RA, RB**	**Trap if (RA) logically not less than (RB).** *Extended mnemonic for* **tw 5,RA,RB**	

Table 9-34. Extended Mnemonics for tw (Continued)

Mnemonic	Operands	Function	Other Registers Altered
twlt	**RA, RB**	**Trap if (RA) less than (RB).** *Extended mnemonic for* **tw 16,RA,RB**	
twne	**RA, RB**	**Trap if (RA) not equal to (RB).** *Extended mnemonic for* **tw 24,RA,RB**	
twng	**RA, RB**	**Trap if (RA) not greater than (RB).** *Extended mnemonic for* **tw 20,RA,RB**	
twnl	**RA, RB**	**Trap if (RA) not less than (RB).** *Extended mnemonic for* **tw 12,RA,RB**	

twi TO, RA, IM

3	TO	RA	IM
0	6	11	16 … 31

$$
\begin{array}{ll}
\text{if (} & ((RA) < \text{EXTS}(IM) \wedge TO_0 = 1) \quad \vee \\
& ((RA) > \text{EXTS}(IM) \wedge TO_1 = 1) \quad \vee \\
& ((RA) = \text{EXTS}(IM) \wedge TO_2 = 1) \quad \vee \\
& ((RA) \stackrel{u}{<} \text{EXTS}(IM) \wedge TO_3 = 1) \quad \vee \\
& ((RA) \stackrel{u}{>} \text{EXTS}(IM) \wedge TO_4 = 1) \quad \text{) then TRAP (see details below)}
\end{array}
$$

Register RA is compared with the IM field, which has been sign-extended to 32 bits. If any comparison condition selected by the TO field is true, a TRAP occurs. The behavior of a TRAP depends upon the Debug Mode of the processor, as described below:

- If TRAP is not enabled as a debug event (DBCR[TDE] = 0 or DBCR[EDM,IDM] = 0,0):

TRAP causes a program interrupt. See *Program Interrupt* on page 123.

(SRR0) $\leftarrow$ address of **twi** instruction
(SRR1) $\leftarrow$ (MSR)
(ESR[PTR]) $\leftarrow$ 1
(MSR[WE, EE, PR, DR, IR]) $\leftarrow$ 0
PC $\leftarrow$ $EVPR_{0:15}$ || 0x0700

- If TRAP is enabled as an External debug event (DBCR[TDE] = 1 and DBCR[EDM] = 1):

TRAP goes to the Debug Stop state, to be handled by an external debugger with hardware control of the PPC405.

(DBSR[TIE]) $\leftarrow$ 1
In addition, if TRAP is also enabled as an Internal debug event (DBCR[IDM] = 1) and Debug Exceptions are disabled (MSR[DE] = 0), then report an imprecise event:
(DBSR[IDE]) $\leftarrow$ 1
PC $\leftarrow$ address of **twi** instruction

- If TRAP is enabled as an Internal debug event and *not* an External debug event (DBCR[TDE] = 1 and DBCR[EDM,IDM] = 0,1) and Debug Exceptions are enabled (MSR[DE] = 1):

TRAP causes a Debug interrupt. See *Debug Interrupt* on page 128.

(SRR2) $\leftarrow$ address of **twi** instruction
(SRR3) $\leftarrow$ (MSR)
(DBSR[TIE]) $\leftarrow$ 1
(MSR[WE, EE, PR, CE, DE, DR, IR]) $\leftarrow$ 0
PC $\leftarrow$ $EVPR_{0:15}$ || 0x2000

- If TRAP is enabled as an Internal debug event and *not* an External debug event (DBCR[TDE] = 1 and DBCR[EDM,IDM] = 0,1) and Debug Exceptions are disabled (MSR[DE] = 0):

TRAP will report the debug event as an *imprecise* event and will cause a Program interrupt. See *Program Interrupt* on page 123.

(SRR0) $\leftarrow$ address of **twi** instruction
(SRR1) $\leftarrow$ (MSR)
(ESR[PTR]) $\leftarrow$ 1
(DBSR[TIE,IDE]) $\leftarrow$ 1,1
(MSR[WE, EE, PR, DR, IR]) $\leftarrow$ 0
PC $\leftarrow$ $EVPR_{0:15}$ || 0x0700

Registers Altered

- None

Programming Note

This instruction is inserted into the execution stream by a debugger to implement breakpoints, and is not typically used by application code.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-35. Extended Mnemonics for twi

Mnemonic	Operands	Function	Other Registers Altered
tweqi	**RA, IM**	**Trap if (RA) equal to EXTS(IM).** *Extended mnemonic for* **twi 4,RA,IM**	
twgei	**RA, IM**	**Trap if (RA) greater than or equal to EXTS(IM).** *Extended mnemonic for* **twi 12,RA,IM**	
twgti	**RA, IM**	**Trap if (RA) greater than EXTS(IM).** *Extended mnemonic for* **twi 8,RA,IM**	
twlei	**RA, IM**	**Trap if (RA) less than or equal to EXTS(IM).** *Extended mnemonic for* **twi 20,RA,IM**	
twlgei	**RA, IM**	**Trap if (RA) logically greater than or equal to EXTS(IM).** *Extended mnemonic for* **twi 5,RA,IM**	
twlgti	**RA, IM**	**Trap if (RA) logically greater than EXTS(IM).** *Extended mnemonic for* **twi 1,RA,IM**	
twllei	**RA, IM**	**Trap if (RA) logically less than or equal to EXTS(IM).** *Extended mnemonic for* **twi 6,RA,IM**	
twllti	**RA, IM**	**Trap if (RA) logically less than EXTS(IM).** *Extended mnemonic for* **twi 2,RA,IM**	
twlngi	**RA, IM**	**Trap if (RA) logically not greater than EXTS(IM).** *Extended mnemonic for* **twi 6,RA,IM**	
twlnli	**RA, IM**	**Trap if (RA) logically not less than EXTS(IM).** *Extended mnemonic for* **twi 5,RA,IM**	
twlti	**RA, IM**	**Trap if (RA) less than EXTS(IM).** *Extended mnemonic for* **twi 16,RA,IM**	
twnei	**RA, IM**	**Trap if (RA) not equal to EXTS(IM).** *Extended mnemonic for* **twi 24,RA,IM**	
twngi	**RA, IM**	**Trap if (RA) not greater than EXTS(IM).** *Extended mnemonic for* **twi 20,RA,IM**	

Table 9-35. Extended Mnemonics for twi (Continued)

Mnemonic	Operands	Function	Other Registers Altered
twnli	**RA, IM**	**Trap if (RA) not less than EXTS(IM).** *Extended mnemonic for* **twi 12,RA,IM**	

wrtee RS

31	RS		131	
0	6	11	21	31

$MSR[EE] \leftarrow (RS)_{16}$

The MSR[EE] is set to the value specified by bit 16 of register RS.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- MSR[EE]

Invalid Instruction Forms:

- Reserved fields

Programming Note

Execution of this instruction is privileged.

This instruction is used to provide atomic update of MSR[EE]. Typical usage is:

```
mfmsr Rn    #save EE in Rn[16]
wrteei 0    #Turn off EE
•           #Code with EE disabled
•
•
wrtee Rn    #restore EE without affecting any MSR changes that occurred in the disabled code
```

Architecture Note

This instruction part of the PowerPC Embedded Operating Environment.

wrteei E

0–5	6–15	16	17–20	21–30	31
31		**E**		**163**	

MSR[EE] $\leftarrow$ E

MSR[EE] is set to the value specified by the E field.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- MSR[EE]

Invalid Instruction Forms:

- Reserved fields

Programming Note

Execution of this instruction is privileged.

This instruction is used to provide an atomic update of MSR[EE]. Typical usage is:

```
mfmsr Rn     #save EE in Rn[16]
wrteei 0     #Turn off EE
•            #Code with EE disabled
•
•
wrtee Rn     #restore EE without affecting any MSR changes that occurred in the disabled code
```

Architecture Note

This instruction part of the PowerPC Embedded Operating Environment.

xor	RA, RS, RB	Rc=0
xor.	RA, RS, RB	Rc=1

31	RS	RA	RB	316	Rc
0	6	11	16	21	31

$(RA) \leftarrow (RS) \oplus (RB)$

The contents of register RS are XORed with the contents of register RB; the result is placed into register RA.

Registers Altered

- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc contains 1
- RA

Architecture Note

This instruction part of the PowerPC Embedded Operating Environment.

xori RA, RS, IM

26	RS	RA	IM
0	6	11	16 … 31

$$(RA) \leftarrow (RS) \oplus (^{16}0 \parallel IM)$$

The IM field is extended to 32 bits by concatenating 16 0-bits on the left. The contents of register RS are XORed with the extended IM field; the result is placed into register RA.

Registers Altered

- RA

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

xoris RA, RS, IM

27	RS	RA	IM
0	6	11	16 … 31

$(RA) \leftarrow (RS) \oplus (IM \,||\, {}^{16}0)$

The IM field is extended to 32 bits by concatenating 16 0-bits on the right. The contents of register RS are XORed with the extended IM field; the result is placed into register RA.

Registers Altered

- RA

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

10. Register Summary

Registers are grouped into categories, based on access mode: General Purpose Registers (GPRs), Special Purpose Registers (SPRs), Time Base Registers (TBRs), the Machine State Register (MSR), the Condition Register (CR), Device Control Registers (DCRs), and memory-mapped I/O (MMIO) registers.

This chapter provides an alphabetical listing and bit definitiions for all the registers provided by the PPC405 processor.

10.1 Reserved Registers

Any register numbers not listed in the tables which follow are *reserved,* and should be neither read nor written. These reserved register numbers may be used for additional functions in future processors.

10.2 Reserved Fields

For all registers with fields marked as reserved, the reserved fields should be written as *zero* and read as *undefined*. That is, when writing to a reserved field, write a zero to that field. When reading from a reserved field, ignore that field.

The recommended coding practice is to perform the initial write to a register with reserved fields as described in the preceding paragraph, and to perform all subsequent writes to the register using a read-modify-write strategy: read the register, alter desired fields with logical instructions, and then write the register.

10.3 General Purpose Registers

The PPC405 processor core provides 32 General Purpose Registers (GPRs). The contents of these registers can be loaded from memory using load instructions and stored to memory using store instructions. GPRs are also addressed by all integer instructions.

Table 10-1. PPC405 General Purpose Registers

Mnemonic	Register Name	GPR Number	Access	See Page
GPR0–GPR31	General Purpose Register 0:31	0x00–0x1F	Read/Write	35

10.4 Machine State Register and Condition Register

The CR and MSR are accessed by means of special instructions,and do not require addressing.

Table 10-2. PPC405 General Purpose Registers

Mnemonic	Register Name	Number	Access	See Page
CR	Condition Register	NA	Read/Write	39
MSR	Machine State Register	NA	Read/Write	114

10.5 Special Purpose Registers

Special Purpose Registers (SPRs), which are part of the PowerPC Embedded Environment, are accessed using the **mtspr** and **mfspr** instructions. SPRs control the use of the debug facilities, timers, interrupts, storage control attributes, and other architected processor resources.

Table 10-3 lists the SPRs, their mnemonics and names, their SPR numbers (SPRNs), and the corresponding SPRF numbers and access mode. Any SPR numbers that are not listed are reserved and should be neither read nor written. The columns under the SPRN heading list the register numbers used as operands in assembler language coding of the **mfspr** and **mtspr** instructions. The column labeled "SPRF" lists the corresponding fields contained in the *machine code* of **mfspr** and **mtspr**. The SPRN field contains the five-bit subfields of the SPRF field, which are *reversed* in the machine code for the **mfspr** and **mtspr** instructions ($SPRN \leftarrow SPRF_{5:9} \parallel SPRF_{0:4}$) for compatibility with the POWER Architecture. Note that the assembler handles the special coding transparently.

All SPRs are privileged, except the Count Register (CTR), the Link Register (LR), SPR General Purpose Registers (SPRG4–SPRG7, read-only), User SPR General Purpose Register (USPRG0), and the Fixed-point Exception Register (XER). Note that access to the Time Base Lower (TBL) and Time Base Upper (TBU) registers, when addressed as SPRs, is write-only and privileged. However, when addressed as Time Base Registers (TBRs), read access to these registers is not privileged. See "Time Base Registers" on page 355. for more information.

Table 10-3. Special Purpose Registers

Mnemonic	Register Name	SPRN	SPRF	Access	See Page
CCR0	Core Configuration Register 0	0x3B3	0x27D	Read/Write	77
CTR	Count Register	0x009	0x120	Read/Write	36
DAC1	Data Address Compare 1	0x3F6	0x2DF	Read/Write	147
DAC2	Data Address Compare 2	0x3F7	0x2FF	Read/Write	147
DBCR0	Debug Control Register 0	0x3F2	0x25F	Read/Write	143
DBCR1	Debug Control Register 1	0x3BD	0x3BD	Read/Write	144
DBSR	Debug Status Register	0x3F0	0x21F	Read/Clear	145
DCCR	Data Cache Cachability Register	0x3FA	0x35F	Read/Write	106
DCWR	Data Cache Write-through Register	0x3BA	0x35D	Read/Write	106
DEAR	Data Error Address Register	0x3D5	0x2BE	Read/Write	118
DVC1	Data Value Compare 1	0x3B6	0x2DD	Read/Write	147
DVC2	Data Value Compare 2	0x3B7	0x2FD	Read/Write	147
ESR	Exception Syndrome Register	0x3D4	0x29E	Read/Write	116
EVPR	Exception Vector Prefix Register	0x3D6	0x2DE	Read/Write	116
IAC1	Instruction Address Compare 1	0x3F4	0x29F	Read/Write	147
IAC2	Instruction Address Compare 2	0x3F5	0x2B5	Read/Write	147
IAC3	Instruction Address Compare 3	0x3B4	0x29D	Read/Write	147
IAC4	Instruction Address Compare 4	0x3B5	0x2BD	Read/Write	147
ICCR	Instruction Cache Cachability Register	0x3FB	0x37F	Read/Write	105
ICDBDR	Instruction Cache Debug Data Register	0x3D3	0x27E	Read-only	80
LR	Link Register	0x008	0x100	Read/Write	37
PID	Process ID	0x3B1	0x23D	Read/Write	102
PIT	Programmable Interval Timer	0x3DB	0x37E	Read/Write	131

Table 10-3. Special Purpose Registers (Continued)

Mnemonic	Register Name	SPRN	SPRF	Access	See Page
PVR	Processor Version Register	0x11F	0x3E8	Read-only	39
SGR	Storage Guarded Register	0x3B9	0x33D	Read/Write	107
SLER	Storage Little Endian Register	0x3BB	0x37D	Read/Write	107
SPRG0	SPR General 0	0x110	0x208	Read/Write	39
SPRG1	SPR General 1	0x111	0x228	Read/Write	39
SPRG2	SPR General 2	0x112	0x248	Read/Write	39
SPRG3	SPR General 3	0x113	0x268	Read/Write	39
SPRG4	SPR General 4	0x104	0x088	Read-only	39
SPRG4	SPR General 4	0x114	0x288	Read/Write	39
SPRG5	SPR General 5	0x105	0x0A8	Read-only	39
SPRG5	SPR General 5	0x115	0x2A8	Read/Write	39
SPRG6	SPR General 6	0x106	0x0C8	Read-only	39
SPRG6	SPR General 6	0x116	0x2C8	Read/Write	39
SPRG7	SPR General 7	0x107	0x0E8	Read-only	39
SPRG7	SPR General 7	0x117	0x2E8	Read/Write	39
SRR0	Save/Restore Register 0	0x01A	0x340	Read/Write	115
SRR1	Save/Restore Register 1	0x01B	0x360	Read/Write	115
SRR2	Save/Restore Register 2	0x3DE	0x3DE	Read/Write	115
SRR3	Save/Restore Register 3	0x3DF	0x3FE	Read/Write	115
SU0R	Storage User-defined 0 Register	0x3BC	0x39D	Read/Write	105
TBL	Time Base Lower	0x11C	0x388	Write-only	130
TBU	Time Base Upper	0x11D	0x3A8	Write-only	130
TCR	Timer Control Register	0x3DA	0x35E	Read/Write	135
TSR	Timer Status Register	0x3D8	0x31E	Read/Clear	135
USPRG0	User SPR General 0	0x100	0x008	Read/Write	39
XER	Fixed Point Exception Register	0x001	0x020	Read/Write	37
ZPR	Zone Protection Register	0x3B0	0x21D	Privileged	103

10.6 Time Base Registers

The PowerPC Architecture provides a 64-bit time base. *Timer Facilities* on page 129 describes the architected time base. In the PPC405, the time base is implemented as two 32-bit time base registers (TBRs). The low-order 32 bits of the time base are read from the TBL and the high-order 32 bits are read from the TBL.

User-mode access to the TBRs is read-only, and there is no explicitly privileged read access to the time base.

The **mftb** instruction reads from TBL and TBU. (Writing the time base is accomplished by moving the contents of a GPR to a pair of SPRs, which are also called TBL and TBU, using the **mtspr** instruction.)

Table 10-4 shows the mnemonics, names, and numbers of the TBRs. The columns under “TBRN” list the register numbers used as operands in assembler language coding of the **mftb** and **mtspr** instructions. The column labeled “TBRF” lists the corresponding fields contained in the *machine code* of **mftb** and **mtspr**. The TBRN field contains two five-bit subfields of the TBRF field; the subfields are *reversed* in the machine code for the **mftb** and **mtspr** instructions ($TBRN \leftarrow TBRF_{5:9} \parallel TBRF_{0:4}$). Note that the assembler handles the special coding transparently.

Table 10-4. Time Base Registers

Mnemonic	Register Name	TBRN		TBRF	Access
		Decimal	Hex		
TBL	Time Base Lower (Read-only)	268	0x10C	0x188	Read-only
TBU	Time Base Upper (Read-only)	269	0x10D	0x1A8	Read-only

10.7 Device Control Registers

DCRs may be used to control various on-chip system functions, such as the operation of on-chip buses, peripherals, and certain processor function behaviors. The DCR access instructions are **mtdcr** (move to device control register) and **mfdcr** (move from device control register), which move data between GPRs and the DCRs. Some DCRs are directly accessed, that is, they are accessed using their DCR numbers. Other DCRs are indirectly accessed. Such DCRs are accessed by writing an offset to a directly accessed DCR and then reading the data at the offset in another directly accessed DCR.

DCRs are unique to the chip in which this processor is instantiated and are not a part of the processor. Refer to the appropriate chip user’s manual for details on the DCRs.

Appendix A. Instruction Summary

This appendix contains PPC405 instructions summarized alphabetically and by opcode.

Appendix A.1 on page 357, illustrates the PPC405 instruction forms (allowed arrangements of fields within instructions).

Appendix A.2 on page 362 lists all PPC405 mnemonics, including extended mnemonics, alphabetically. A short functional description is included for each mnemonic.

Appendix A.3 on page 388, lists all PPC405 instructions, sorted by primary and secondary opcodes. Extended mnemonics are not included in the opcode list.

A.1 Instruction Formats

Instructions are four bytes long. Instruction addresses are always word-aligned.

Instruction bits 0 through 5 always contain the primary opcode. Many instructions have an extended opcode in another field. Remaining instruction bits contain additional fields. All instruction fields belong to one of the following categories:

- Defined

  These instructions contain values, such as opcodes, that cannot be altered. The instruction format diagrams specify the values of defined fields.

- Variable

  These fields contain operands, such as GPR selectors and immediate values, that can vary from execution to execution. The instruction format diagrams specify the operands in the variable fields.

- Reserved

  Bits in reserved fields should be set to 0. In the instruction format diagrams, /, //, or /// indicate reserved fields.

If any bit in a defined field does not contain the expected value, the instruction is illegal and an illegal instruction exception occurs. If any bit in a reserved field does not contain 0, the instruction form is invalid; its result is architecturally undefined. The PPC405 executes all invalid instruction forms without causing an illegal instruction exception.

A.1.1 Instruction Fields

PPC405 instructions contain various combinations of the following fields, as indicated in the instruction format diagrams that follow the field definitions. Numbers, enclosed in parentheses, that follow the field names indicate bit positions; bit fields are indicated by starting and stopping bit positions separated by colons.

AA (30) Absolute address bit.

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.

BA (11:15) Specifies a bit in the CR used as a source of a CR-logical instruction.

BB (16:20) Specifies a bit in the CR used as a source of a CR-logical instruction.

BD (16:29)	An immediate field specifying a 14-bit signed twos complement branch displacement. This field is concatenated on the right with 0b00 and sign-extended to 32 bits.
BF (6:8)	Specifies a field in the CR used as a target in a compare or **mcrf** instruction.
BFA (11:13)	Specifies a field in the CR used as a source in a **mcrf** instruction.
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 *BO Field on Conditional Branches* on page 51.
BT (6:10)	Specifies a bit in the CR used as a target as the result of a CR-Logical instruction.
D (16:31)	Specifies a 16-bit signed twos-complement integer displacement for load/store instructions.
DCRN (11:20)	Specifies a device control register (DCR).
FXM (12:19)	Field mask used to identify CR fields to be updated by the **mtcrf** instruction.
IM (16:31)	An immediate field used to specify a 16-bit value (either signed integer or unsigned).
LI (6:29)	An immediate field specifying a 24-bit signed twos complement branch displacement; this field is concatenated on the right with b'00' 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 store.
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.
OE (21)	Enables setting the OV and SO fields in the fixed-point exception register (XER) for extended arithmetic.
RA (11:15)	A GPR used as a source or target.
RB (16:20)	A GPR used as a source.
Rc (31)	Record bit. 0 Do not set the CR. 1 Set the CR to reflect the result of an operation. See *Condition Register (CR)* on page 39 for a further discussion of how the CR bits are set.
RS (6:10)	A GPR used as a source.
RT (6:10)	A GPR used as a target.
SH (16:20)	Specifies a shift amount.
SPRF (11:20)	Specifies a special purpose register (SPR).
TO (6:10)	Specifies the conditions on which to trap, as described under **tw** and **twi** instructions.
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.

A.1.2 Instruction Format Diagrams

The instruction formats (also called *forms*) illustrated in Figure A-1 through Figure A-9 are valid combinations of instruction fields. Table A-2 on page -388 indicates which form is utilized by each PPC405 opcode. Fields indicated by slashes (/, //, or ///) are reserved. The figures are adapted from the PowerPC User Instruction Set Architecture.

A.1.2.1 I-Form

Figure A-1. I Instruction Format

OPCD	LI
0	6 … 31

A.1.2.2 B-Form

Figure A-2. B Instruction Format

OPCD	BO	BI	BD	AA	LK
0	6	11	16	30	31

A.1.2.3 SC-Form

Figure A-3. SC Instruction Format

OPCD	///	///	///	1	/
0	6	11	16	30	31

A.1.2.4 D-Form

Figure A-4. D Instruction Format

OPCD				RA	
OPCD	RT			RA	D
OPCD	RS			RA	SI
OPCD	RS			RA	D
OPCD	RS			RA	UI
OPCD	BF	/	L	RA	SI
OPCD	BF	/	L	RA	UI
OPCD	TO			RA	SI
0	6			11	16 … 31

A.1.2.5 X-Form

Figure A-5. X Instruction Format

<table>
<tr><td>OPCD</td><td colspan="3">RT</td><td colspan="2">RA</td><td colspan="2">RB</td><td>XO</td><td>Rc</td></tr>
<tr><td>OPCD</td><td colspan="3">RT</td><td colspan="2">RA</td><td colspan="2">RB</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">RT</td><td colspan="2">RA</td><td colspan="2">NB</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">RT</td><td colspan="2">RA</td><td colspan="2">WS</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">RT</td><td colspan="2">///</td><td colspan="2">RB</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">RT</td><td colspan="2">///</td><td colspan="2">///</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">RS</td><td colspan="2">RA</td><td colspan="2">RB</td><td>XO</td><td>Rc</td></tr>
<tr><td>OPCD</td><td colspan="3">RS</td><td colspan="2">RA</td><td colspan="2">RB</td><td>XO</td><td>1</td></tr>
<tr><td>OPCD</td><td colspan="3">RS</td><td colspan="2">RA</td><td colspan="2">RB</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">RS</td><td colspan="2">RA</td><td colspan="2">NB</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">RS</td><td colspan="2">RA</td><td colspan="2">WS</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">RS</td><td colspan="2">RA</td><td colspan="2">SH</td><td>XO</td><td>Rc</td></tr>
<tr><td>OPCD</td><td colspan="3">RS</td><td colspan="2">RA</td><td colspan="2">///</td><td>XO</td><td>Rc</td></tr>
<tr><td>OPCD</td><td colspan="3">RS</td><td colspan="2">///</td><td colspan="2">RB</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">RS</td><td colspan="2">///</td><td colspan="2">///</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td>BF</td><td>/</td><td>L</td><td colspan="2">RA</td><td colspan="2">RB</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td>BF</td><td colspan="2">//</td><td>BFA</td><td>//</td><td colspan="2">///</td><td>XO</td><td>Rc</td></tr>
<tr><td>OPCD</td><td>BF</td><td colspan="2">//</td><td colspan="2">///</td><td colspan="2">///</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td>BF</td><td colspan="2">//</td><td colspan="2">///</td><td colspan="2">U</td><td>XO</td><td>Rc</td></tr>
<tr><td>OPCD</td><td>BF</td><td colspan="2">//</td><td colspan="2">///</td><td colspan="2">///</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">TO</td><td colspan="2">RA</td><td colspan="2">RB</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">BT</td><td colspan="2">///</td><td colspan="2">///</td><td>XO</td><td>Rc</td></tr>
<tr><td>OPCD</td><td colspan="3">///</td><td colspan="2">RA</td><td colspan="2">RB</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">///</td><td colspan="2">///</td><td colspan="2">///</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="3">///</td><td colspan="2">///</td><td>E</td><td>//</td><td>XO</td><td>/</td></tr>
<tr><td>0</td><td colspan="3">6</td><td colspan="2">11</td><td colspan="2">16</td><td>21</td><td>31</td></tr>
</table>

A.1.2.6 XL-Form

Figure A-6. XL Instruction Format

<table>
<tr><td>OPCD</td><td colspan="2">BT</td><td colspan="2">BA</td><td>BB</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="2">BC</td><td colspan="2">BI</td><td>///</td><td>XO</td><td>LK</td></tr>
<tr><td>OPCD</td><td>BF</td><td>//</td><td>BFA</td><td>//</td><td>///</td><td>XO</td><td>/</td></tr>
<tr><td>OPCD</td><td colspan="2">///</td><td colspan="2">///</td><td>///</td><td>XO</td><td>/</td></tr>
<tr><td>0</td><td colspan="2">6</td><td colspan="2">11</td><td>16</td><td>21</td><td>31</td></tr>
</table>

A.1.2.7 XFX-Form

Figure A-7. XFX Instruction Format

OPCD	RT		SPRF		XO	/
OPCD	RT		DCRF		XO	/
OPCD	RT	/	FXM	/	XO	/
OPCD	RS		SPRF		XO	/
OPCD	RS		DCRF		XO	/
0	6	11	16		21	31

A.1.2.8 X0-Form

Figure A-8. XO Instruction Format

OPCD	RT	RA	RB	OE	XO	Rc
OPCD	RT	RA	RB	OE	XO	Rc
OPCD	RT	RA	///	/	XO	Rc
0	6	11	16	21	22	31

A.1.2.9 M-Form

Figure A-9. M Instruction Format

OPCD	RS	RA	RB	MB	ME	Rc
OPCD	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

A.2 List of Implemented Instructions—Alphabetical

Table A-1 summarizes the PPC405 instruction set, including required extended mnemonics. All mnemonics are listed alphabetically, without regard to whether the mnemonic is realized in hardware or software. When an instruction supports multiple hardware mnemonics (for example, **b**, **ba**, **bl**, **bla** are all forms of **b**), the instruction is alphabetized under the root form. The hardware instructions are described in detail in *Instruction Set* on page 157 which is also alphabetized under the root form. That section also describes the instruction operands and notation.

Programming Note: Bit 4 of the BO field provides a hint about the most likely outcome of a conditional branch. (See *Branch Prediction* on page 52 for a detailed description of branch prediction.) Assemblers should set BO_4 = 0 unless a specific reason exists otherwise. In the BO field values specified in the table below, BO_4 = 0 has always been assumed. The assembler must allow the programmer to specify branch prediction. To do this, the assembler supports a suffixes for the conditional branch mnemonics:

\+ Predict branch to be taken.

– Predict branch not to be taken.

For example, **bc** could also be coded as **bc+** or **bc–**, and **bne** could also be coded **bne+** or **bne–**. These alternate codings set BO_4 = 1 only if the requested prediction differs from the standard prediction.See *Branch Prediction* on page 52 for more information.

Table A-1. PPC405 Instruction Syntax Summary

Mnemonic	Operands	Function	Other Registers Changed	Page
add	RT, RA, RB	Add (RA) to (RB). Place result in RT.		161
add.			CR[CR0]	
addo			XER[SO, OV]	
addo.			CR[CR0] XER[SO, OV]	
addc	RT, RA, RB	Add (RA) to (RB). Place result in RT. Place carry-out in XER[CA].		162
addc.			CR[CR0]	
addco			XER[SO, OV]	
addco.			CR[CR0] XER[SO, OV]	
adde	RT, RA, RB	Add XER[CA], (RA), (RB). Place result in RT. Place carry-out in XER[CA].		163
adde.			CR[CR0]	
addeo			XER[SO, OV]	
addeo.			CR[CR0] XER[SO, OV]	
addi	RT, RA, IM	Add EXTS(IM) to (RA|0). Place result in RT.		164
addic	RT, RA, IM	Add EXTS(IM) to (RA|0). Place result in RT. Place carry-out in XER[CA].		166
addic.	RT, RA, IM	Add EXTS(IM) to (RA|0). Place result in RT. Place carry-out in XER[CA].	CR[CR0]	166
addis	RT, RA, IM	Add (IM $\|\|$ $^{16}0$) to (RA|0). Place result in RT.		167
addme	RT, RA	Add XER[CA], (RA), (-1). Place result in RT. Place carry-out in XER[CA].		168
addme.			CR[CR0]	
addmeo			XER[SO, OV]	
addmeo.			CR[CR0] XER[SO, OV]	
addze	RT, RA	Add XER[CA] to (RA). Place result in RT. Place carry-out in XER[CA].		169
addze.			CR[CR0]	
addzeo			XER[SO, OV]	
addzeo.			CR[CR0] XER[SO, OV]	
and	RA, RS, RB	AND (RS) with (RB). Place result in RA.		170
and.			CR[CR0]	
andc	RA, RS, RB	AND (RS) with ¬(RB). Place result in RA.		171
andc.			CR[CR0]	
andi.	RA, RS, IM	AND (RS) with ($^{16}0$ $\|\|$ IM). Place result in RA.	CR[CR0]	172
andis.	RA, RS, IM	AND (RS) with (IM $\|\|$ $^{16}0$). Place result in RA.	CR[CR0]	173

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
b	target	Branch unconditional relative. $LI \leftarrow (target - CIA)_{6:29}$ $NIA \leftarrow CIA + EXTS(LI \parallel {}^{2}0)$		174
ba		Branch unconditional absolute. $LI \leftarrow target_{6:29}$ $NIA \leftarrow EXTS(LI \parallel {}^{2}0)$		
bl		Branch unconditional relative. $LI \leftarrow (target - CIA)_{6:29}$ $NIA \leftarrow CIA + EXTS(LI \parallel {}^{2}0)$	(LR) $\leftarrow$ CIA + 4.	
bla		Branch unconditional absolute. $LI \leftarrow target_{6:29}$ $NIA \leftarrow EXTS(LI \parallel {}^{2}0)$	(LR) $\leftarrow$ CIA + 4.	
bc	BO, BI, target	Branch conditional relative. $BD \leftarrow (target - CIA)_{16:29}$ $NIA \leftarrow CIA + EXTS(BD \parallel {}^{2}0)$	CTR if BO_2 = 0.	175
bca		Branch conditional absolute. $BD \leftarrow target_{16:29}$ $NIA \leftarrow EXTS(BD \parallel {}^{2}0)$	CTR if BO_2 = 0.	
bcl		Branch conditional relative. $BD \leftarrow (target - CIA)_{16:29}$ $NIA \leftarrow CIA + EXTS(BD \parallel {}^{2}0)$	CTR if BO_2 = 0. (LR) $\leftarrow$ CIA + 4.	
bcla		Branch conditional absolute. $BD \leftarrow target_{16:29}$ $NIA \leftarrow EXTS(BD \parallel {}^{2}0)$	CTR if BO_2 = 0. (LR) $\leftarrow$ CIA + 4.	
bcctr	BO, BI	Branch conditional to address in CTR. Using (CTR) at exit from instruction, $NIA \leftarrow CTR_{0:29} \parallel {}^{2}0.$	CTR if BO_2 = 0.	181
bcctrl			CTR if BO_2 = 0. (LR) $\leftarrow$ CIA + 4.	
bclr	BO, BI	Branch conditional to address in LR. Using (LR) at entry to instruction, $NIA \leftarrow LR_{0:29} \parallel {}^{2}0.$	CTR if BO_2 = 0.	184
bclrl			CTR if BO_2 = 0. (LR) $\leftarrow$ CIA + 4.	
bctr		Branch unconditionally to address in CTR. *Extended mnemonic for* **bcctr 20,0**		181
bctrl		*Extended mnemonic for* **bcctrl 20,0**	(LR) $\leftarrow$ CIA + 4.	
bdnz	target	Decrement CTR. Branch if CTR $\neq$ 0. *Extended mnemonic for* **bc 16,0,target**		175
bdnza		*Extended mnemonic for* **bca 16,0,target**		
bdnzl		*Extended mnemonic for* **bcl 16,0,target**	(LR) $\leftarrow$ CIA + 4.	
bdnzla		*Extended mnemonic for* **bcla 16,0,target**	(LR) $\leftarrow$ CIA + 4.	
bdnzlr		Decrement CTR. Branch if CTR $\neq$ 0 to address in LR. *Extended mnemonic for* **bclr 16,0**		175
bdnzlrl		*Extended mnemonic for* **bclrl 16,0**	(LR) $\leftarrow$ CIA + 4.	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bdnzf	cr_bit, target	Decrement CTR. Branch if CTR ≠ 0 AND CR_{cr_bit} = 0. *Extended mnemonic for* **bc 0,cr_bit,target**		175
bdnzfa		*Extended mnemonic for* **bca 0,cr_bit,target**		
bdnzfl		*Extended mnemonic for* **bcl 0,cr_bit,target**	(LR) ← CIA + 4.	
bdnzfla		*Extended mnemonic for* **bcla 0,cr_bit,target**	(LR) ← CIA + 4.	
bdnzflr	cr_bit	Decrement CTR. Branch if CTR ≠ 0 AND CR_{cr_bit} = 0 to address in LR. *Extended mnemonic for* **bclr 0,cr_bit**		175
bdnzflrl		*Extended mnemonic for* **bclrl 0,cr_bit**	(LR) ← CIA + 4.	
bdnzt	cr_bit, target	Decrement CTR. Branch if CTR ≠ 0 AND CR_{cr_bit} = 1. *Extended mnemonic for* **bc 8,cr_bit,target**		175
bdnzta		*Extended mnemonic for* **bca 8,cr_bit,target**		
bdnztl		*Extended mnemonic for* **bcl 8,cr_bit,target**	(LR) ← CIA + 4.	
bdnztla		*Extended mnemonic for* **bcla 8,cr_bit,target**	(LR) ← CIA + 4.	
bdnztlr	cr_bit	Decrement CTR. Branch if CTR ≠ 0 AND CR_{cr_bit} = 1 to address in LR. *Extended mnemonic for* **bclr 8,cr_bit**		175
bdnztlrl		*Extended mnemonic for* **bclrl 8,cr_bit**	(LR) ← CIA + 4.	
bdz	target	Decrement CTR. Branch if CTR = 0. *Extended mnemonic for* **bc 18,0,target**		175
bdza		*Extended mnemonic for* **bca 18,0,target**		
bdzl		*Extended mnemonic for* **bcl 18,0,target**	(LR) ← CIA + 4.	
bdzla		*Extended mnemonic for* **bcla 18,0,target**	(LR) ← CIA + 4.	
bdzlr		Decrement CTR. Branch if CTR = 0 to address in LR. *Extended mnemonic for* **bclr 18,0**		175
bdzlrl		*Extended mnemonic for* **bclrl 18,0**	(LR) ← CIA + 4.	
bdzf	cr_bit, target	Decrement CTR. Branch if CTR = 0 AND CR_{cr_bit} = 0. *Extended mnemonic for* **bc 2,cr_bit,target**		175
bdzfa		*Extended mnemonic for* **bca 2,cr_bit,target**		
bdzfl		*Extended mnemonic for* **bcl 2,cr_bit,target**	(LR) ← CIA + 4.	
bdzfla		*Extended mnemonic for* **bcla 2,cr_bit,target**	(LR) ← CIA + 4.	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bdzflr	cr_bit	Decrement CTR. Branch if CTR = 0 AND CR_{cr_bit} = 0 to address in LR. *Extended mnemonic for* **bclr 2,cr_bit**		175
bdzflrl		*Extended mnemonic for* **bclrl 2,cr_bit**	(LR) ← CIA + 4.	
bdzt	cr_bit, target	Decrement CTR. Branch if CTR = 0 AND CR_{cr_bit} = 1. *Extended mnemonic for* **bc 10,cr_bit,target**		175
bdzta		*Extended mnemonic for* **bca 10,cr_bit,target**		
bdztl		*Extended mnemonic for* **bcl 10,cr_bit,target**	(LR) ← CIA + 4.	
bdztla		*Extended mnemonic for* **bcla 10,cr_bit,target**	(LR) ← CIA + 4.	
bdztlr	cr_bit	Decrement CTR. Branch if CTR = 0 AND CR_{cr_bit} = 1to address in LR. *Extended mnemonic for* **bclr 10,cr_bit**		184
bdztlrl		*Extended mnemonic for* **bclrl 10,cr_bit**	(LR) ← CIA + 4.	
beq	[cr_field], target	Branch if equal. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 12,4∗cr_field+2,target**		184
beqa		*Extended mnemonic for* **bca 12,4∗cr_field+2,target**		
beql		*Extended mnemonic for* **bcl 12,4∗cr_field+2,target**	(LR) ← CIA + 4.	
beqla		*Extended mnemonic for* **bcla 12,4∗cr_field+2,target**	(LR) ← CIA + 4.	
beqctr	[cr_field]	Branch if equal to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4∗cr_field+2**		181
beqctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+2**	(LR) ← CIA + 4.	
beqlr	[cr_field]	Branch if equal to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 12,4∗cr_field+2**		184
beqlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+2**	(LR) ← CIA + 4.	
bf	cr_bit, target	Branch if CR_{cr_bit} = 0. *Extended mnemonic for* **bc 4,cr_bit,target**		175
bfa		*Extended mnemonic for* **bca 4,cr_bit,target**		
bfl		*Extended mnemonic for* **bcl 4,cr_bit,target**	(LR) ← CIA + 4.	
bfla		*Extended mnemonic for* **bcla 4,cr_bit,target**	(LR) ← CIA + 4.	
bfctr	cr_bit	Branch if CR_{cr_bit} = 0 to address in CTR. *Extended mnemonic for* **bcctr 4,cr_bit**		181
bfctrl		*Extended mnemonic for* **bcctrl 4,cr_bit**	(LR) ← CIA + 4.	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bflr	cr_bit	Branch if CR_{cr_bit} = 0 to address in LR. *Extended mnemonic for* **bclr 4,cr_bit**		184
bflrl		*Extended mnemonic for* **bclrl 4,cr_bit**	(LR) ← CIA + 4.	
bge	[cr_field], target	Branch if greater than or equal. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+0,target**		175
bgea		*Extended mnemonic for* **bca 4,4∗cr_field+0,target**		
bgel		*Extended mnemonic for* **bcl 4,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bgela		*Extended mnemonic for* **bcla 4,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bgectr	[cr_field]	Branch if greater than or equal to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+0**		181
bgectrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+0**	(LR) ← CIA + 4.	
bgelr	[cr_field]	Branch if greater than or equal to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+0**		184
bgelrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+0**	(LR) ← CIA + 4.	
bgt	[cr_field], target	Branch if greater than. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 12,4∗cr_field+1,target**		175
bgta		*Extended mnemonic for* **bca 12,4∗cr_field+1,target**		
bgtl		*Extended mnemonic for* **bcl 12,4∗cr_field+1,target**	(LR) ← CIA + 4.	
bgtla		*Extended mnemonic for* **bcla 12,4∗cr_field+1,target**	(LR) ← CIA + 4.	
bgtctr	[cr_field]	Branch if greater than to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4∗cr_field+1**		181
bgtctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+1**	(LR) ← CIA + 4.	
bgtlr	[cr_field]	Branch if greater than to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 12,4∗cr_field+1**		184
bgtlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+1**	(LR) ← CIA + 4.	
ble	[cr_field], target	Branch if less than or equal. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+1,target**		175
blea		*Extended mnemonic for* **bca 4,4∗cr_field+1,target**		
blel		*Extended mnemonic for* **bcl 4,4∗cr_field+1,target**	(LR) ← CIA + 4.	
blela		*Extended mnemonic for* **bcla 4,4∗cr_field+1,target**	(LR) ← CIA + 4.	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
blectr	[cr_field]	Branch if less than or equal to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+1**		181
blectrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+1**	(LR) ← CIA + 4.	
blelr	[cr_field]	Branch if less than or equal to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+1**		184
blelrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+1**	(LR) ← CIA + 4.	
blr		Branch unconditionally to address in LR. *Extended mnemonic for* **bclr 20,0**		184
blrl		*Extended mnemonic for* **bclrl 20,0**	(LR) ← CIA + 4.	
blt	[cr_field], target	Branch if less than. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 12,4∗cr_field+0,target**		175
blta		*Extended mnemonic for* **bca 12,4∗cr_field+0,target**		
bltl		*Extended mnemonic for* **bcl 12,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bltla		*Extended mnemonic for* **bcla 12,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bltctr	[cr_field]	Branch if less than to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4∗cr_field+0**		181
bltctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+0**	(LR) ← CIA + 4.	
bltlr	[cr_field]	Branch if less than to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 12,4∗cr_field+0**		184
bltlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+0**	(LR) ← CIA + 4.	
bne	[cr_field], target	Branch if not equal. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+2,target**		175
bnea		*Extended mnemonic for* **bca 4,4∗cr_field+2,target**		
bnel		*Extended mnemonic for* **bcl 4,4∗cr_field+2,target**	(LR) ← CIA + 4.	
bnela		*Extended mnemonic for* **bcla 4,4∗cr_field+2,target**	(LR) ← CIA + 4.	
bnectr	[cr_field]	Branch if not equal to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+2**		181
bnectrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+2**	(LR) ← CIA + 4.	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bnelr	[cr_field]	Branch if not equal to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+2**		184
bnelrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+2**	(LR) ← CIA + 4.	
bng	[cr_field], target	Branch if not greater than. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+1,target**		175
bnga		*Extended mnemonic for* **bca 4,4∗cr_field+1,target**		
bngl		*Extended mnemonic for* **bcl 4,4∗cr_field+1,target**	(LR) ← CIA + 4.	
bngla		*Extended mnemonic for* **bcla 4,4∗cr_field+1,target**	(LR) ← CIA + 4.	
bngctr	[cr_field]	Branch if not greater than to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+1**		181
bngctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+1**	(LR) ← CIA + 4.	
bnglr	[cr_field]	Branch if not greater than to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+1**		184
bnglrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+1**	(LR) ← CIA + 4.	
bnl	[cr_field], target	Branch if not less than. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+0,target**		175
bnla		*Extended mnemonic for* **bca 4,4∗cr_field+0,target**		
bnll		*Extended mnemonic for* **bcl 4,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bnlla		*Extended mnemonic for* **bcla 4,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bnlctr	[cr_field]	Branch if not less than to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+0**		181
bnlctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+0**	(LR) ← CIA + 4.	
bnllr	[cr_field]	Branch if not less than to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+0**		184
bnllrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+0**	(LR) ← CIA + 4.	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bns	[cr_field], target	Branch if not summary overflow. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+3,target**		175
bnsa		*Extended mnemonic for* **bca 4,4∗cr_field+3,target**		
bnsl		*Extended mnemonic for* **bcl 4,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bnsla		*Extended mnemonic for* **bcla 4,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bnsctr	[cr_field]	Branch if not summary overflow to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+3**		181
bnsctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+3**	(LR) ← CIA + 4.	
bnslr	[cr_field]	Branch if not summary overflow to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+3**		184
bnslrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+3**	(LR) ← CIA + 4.	
bnu	[cr_field], target	Branch if not unordered. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+3,target**		175
bnua		*Extended mnemonic for* **bca 4,4∗cr_field+3,target**		
bnul		*Extended mnemonic for* **bcl 4,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bnula		*Extended mnemonic* for **bcla 4,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bnuctr	[cr_field]	Branch if not unordered to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+3**		181
bnuctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+3**	(LR) ← CIA + 4.	
bnulr	[cr_field]	Branch if not unordered to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+3**		184
bnulrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+3**	(LR) ← CIA + 4.	
bso	[cr_field], target	Branch if summary overflow. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 12,4∗cr_field+3,target**		175
bsoa		*Extended mnemonic for* **bca 12,4∗cr_field+3,target**		
bsol		*Extended mnemonic for* **bcl 12,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bsola		*Extended mnemonic for* **bcla 12,4∗cr_field+3,target**	(LR) ← CIA + 4.	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bsoctr	[cr_field]	Branch if summary overflow to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4∗cr_field+3**		181
bsoctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+3**	(LR) ← CIA + 4.	
bsolr	[cr_field]	Branch if summary overflow to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 12,4∗cr_field+3**		184
bsolrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+3**	(LR) ← CIA + 4.	
bt	cr_bit, target	Branch if CR_{cr_bit} = 1. *Extended mnemonic for* **bc 12,cr_bit,target**		175
bta		*Extended mnemonic for* **bca 12,cr_bit,target**		
btl		*Extended mnemonic for* **bcl 12,cr_bit,target**	(LR) ← CIA + 4.	
btla		*Extended mnemonic for* **bcla 12,cr_bit,target**	(LR) ← CIA + 4.	
btctr	cr_bit	Branch if CR_{cr_bit} = 1 to address in CTR. *Extended mnemonic for* **bcctr 12,cr_bit**		181
btctrl		*Extended mnemonic for* **bcctrl 12,cr_bit**	(LR) ← CIA + 4.	
btlr	cr_bit	Branch if CR_{cr_bit} = 1, to address in LR. *Extended mnemonic for* **bclr 12,cr_bit**		184
btlrl		*Extended mnemonic for* **bclrl 12,cr_bit**	(LR) ← CIA + 4.	
bun	[cr_field], target	Branch if unordered. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 12,4∗cr_field+3,target**		175
buna		*Extended mnemonic for* **bca 12,4∗cr_field+3,target**		
bunl		*Extended mnemonic for* **bcl 12,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bunla		*Extended mnemonic for* **bcla 12,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bunctr	[cr_field]	Branch if unordered to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4∗cr_field+3**		181
bunctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+3**	(LR) ← CIA + 4.	
bunlr	[cr_field]	Branch if unordered, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 12,4∗cr_field+3**		184
bunlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+3**	(LR) ← CIA + 4.	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
clrlwi	RA, RS, n	Clear left immediate. (n < 32) $(RA)_{0:n-1} \leftarrow {}^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,0,n,31**		300
clrlwi.		*Extended mnemonic for* **rlwinm. RA,RS,0,n,31**	CR[CR0]	
clrlslwi	RA, RS, b, n	Clear left and shift left immediate. $(n \le b < 32)$ $(RA)_{b-n:31-n} \leftarrow (RS)_{b:31}$ $(RA)_{32-n:31} \leftarrow {}^{n}0$ $(RA)_{0:b-n-1} \leftarrow {}^{b-n}0$ *Extended mnemonic for* **rlwinm RA,RS,n,b−n,31−n**		300
clrlslwi.		*Extended mnemonic for* **rlwinm. RA,RS,n,b−n,31−n**	CR[CR0]	
clrrwi	RA, RS, n	Clear right immediate. (n < 32) $(RA)_{32-n:31} \leftarrow {}^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,0,0,31−n**		300
clrrwi.		*Extended mnemonic for* **rlwinm. RA,RS,0,0,31−n**	CR[CR0]	
cmp	BF, 0, RA, RB	Compare (RA) to (RB), signed. Results in CR[CRn], where n = BF.		188
cmpi	BF, 0, RA, IM	Compare (RA) to EXTS(IM), signed. Results in CR[CRn], where n = BF.		189
cmpl	BF, 0, RA, RB	Compare (RA) to (RB), unsigned. Results in CR[CRn], where n = BF.		190
cmpli	BF, 0, RA, IM	Compare (RA) to $({}^{16}0 \parallel IM)$, unsigned. Results in CR[CRn], where n = BF.		191
cmplw	[BF,] RA, RB	Compare Logical Word. Use CR0 if BF is omitted. *Extended mnemonic for* **cmpl BF,0,RA,RB**		190
cmplwi	[BF,] RA, IM	Compare Logical Word Immediate. Use CR0 if BF is omitted. *Extended mnemonic for* **cmpli BF,0,RA,IM**		191
cmpw	[BF,] RA, RB	Compare Word. Use CR0 if BF is omitted. *Extended mnemonic for* **cmp BF,0,RA,RB**		188
cmpwi	[BF,] RA, IM	Compare Word Immediate. Use CR0 if BF is omitted. *Extended mnemonic for* **cmpi BF,0,RA,IM**		189
cntlzw	RA, RS	Count leading zeros in RS. Place result in RA.		192
cntlzw.			CR[CR0]	
crand	BT, BA, BB	AND bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		193
crandc	BT, BA, BB	AND bit (CR_{BA}) with $\neg(CR_{BB})$. Place result in CR_{BT}.		194
crclr	bx	Condition register clear. *Extended mnemonic for* **crxor bx,bx,bx**		200
creqv	BT, BA, BB	Equivalence of bit CR_{BA} with CR_{BB}. $CR_{BT} \leftarrow \neg(CR_{BA} \oplus CR_{BB})$		195
crmove	bx, by	Condition register move. *Extended mnemonic for* **cror bx,by,by**		198
crnand	BT, BA, BB	NAND bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		196

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
crnor	BT, BA, BB	NOR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		197
crnot	bx, by	Condition register not. *Extended mnemonic for* **crnor bx,by,by**		197
cror	BT, BA, BB	OR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		198
crorc	BT, BA, BB	OR bit (CR_{BA}) with ¬(CR_{BB}). Place result in CR_{BT}.		199
crset	bx	Condition register set. *Extended mnemonic for* **creqv bx,bx,bx**		195
crxor	BT, BA, BB	XOR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		200
dcba	RA, RB	Speculatively establish the data cache block which contains the effective address (RA|0) + (RB).		201
dcbf	RA, RB	Flush (store, then invalidate) the data cache block which contains the effective address (RA|0) + (RB).		203
dcbi	RA, RB	Invalidate the data cache block which contains the effective address (RA|0) + (RB).		204
dcbst	RA, RB	Store the data cache block which contains the effective address (RA|0) + (RB).		205
dcbt	RA, RB	Load the data cache block which contains the effective address (RA|0) + (RB).		206
dcbtst	RA,RB	Load the data cache block which contains the effective address (RA|0) + (RB).		207
dcbz	RA, RB	Zero the data cache block which contains the effective address (RA|0) + (RB).		208
dccci	RA, RB	Invalidate the data cache congruence class associated with the effective address (RA|0) + (RB).		210
dcread	RT, RA, RB	Read either tag or data information from the data cache congruence class associated with the effective address (RA|0) + (RB). Place the results in RT.		211
divw	RT, RA, RB	Divide (RA) by (RB), signed. Place result in RT.		213
divw.			CR[CR0]	
divwo			XER[SO, OV]	
divwo.			CR[CR0] XER[SO, OV]	
divwu	RT, RA, RB	Divide (RA) by (RB), unsigned. Place result in RT.		214
divwu.			CR[CR0]	
divwuo			XER[SO, OV]	
divwuo.			CR[CR0] XER[SO, OV]	
eieio		Storage synchronization. All loads and stores that precede the **eieio** instruction complete before any loads and stores that follow the instruction access main storage. Implemented as **sync**, which is more restrictive.		215
eqv	RA, RS, RB	Equivalence of (RS) with (RB). $(RA) \leftarrow \neg((RS) \oplus (RB))$		216
eqv.			CR[CR0]	
extlwi	RA, RS, n, b	Extract and left justify immediate. (n > 0) $(RA)_{0:n-1} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{n:31} \leftarrow {}^{32-n}0$ *Extended mnemonic for* **rlwinm RA,RS,b,0,n−1**		300
extlwi.		*Extended mnemonic for* **rlwinm. RA,RS,b,0,n−1**	CR[CR0]	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
extrwi	RA, RS, n, b	Extract and right justify immediate. (n > 0) $(RA)_{32-n:31} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{0:31-n} \leftarrow {}^{32-n}0$ *Extended mnemonic for* **rlwinm RA,RS,b+n,32−n,31**		300
extrwi.		*Extended mnemonic for* **rlwinm. RA,RS,b+n,32−n,31**	CR[CR0]	
extsb	RA, RS	Extend the sign of byte $(RS)_{24:31}$. Place the result in RA.		217
extsb.			CR[CR0]	
extsh	RA, RS	Extend the sign of halfword $(RS)_{16:31}$. Place the result in RA.		218
extsh.			CR[CR0]	
icbi	RA, RB	Invalidate the instruction cache block which contains the effective address (RA|0) + (RB).		219
icbt	RA, RB	Load the instruction cache block which contains the effective address (RA|0) + (RB).		220
iccci	RA, RB	Invalidate instruction cache.		221
icread	RA, RB	Read either tag or data information from the instruction cache congruence class associated with the effective address (RA|0) + (RB). Place the results in ICDBDR.		222
inslwi	RA, RS, n, b	Insert from left immediate. (n > 0) $(RA)_{b:b+n-1} \leftarrow (RS)_{0:n-1}$ *Extended mnemonic for* **rlwimi RA,RS,32−b,b,b+n−1**		299
inslwi.		*Extended mnemonic for* **rlwimi. RA,RS,32−b,b,b+n−1**	CR[CR0]	
insrwi	RA, RS, n, b	Insert from right immediate. (n > 0) $(RA)_{b:b+n-1} \leftarrow (RS)_{32-n:31}$ *Extended mnemonic for* **rlwimi RA,RS,32−b−n,b,b+n−1**		299
insrwi.		*Extended mnemonic for* **rlwimi. RA,RS,32−b−n,b,b+n−1**	CR[CR0]	
isync		Synchronize execution context by flushing the prefetch queue.		224
la	RT, D(RA)	Load address. ($RA \neq 0$) D is an offset from a base address that is assumed to be (RA). $(RT) \leftarrow (RA) + EXTS(D)$ *Extended mnemonic for* **addi RT,RA,D**		164
lbz	RT, D(RA)	Load byte from EA = (RA|0) + EXTS(D) and pad left with zeroes, $(RT) \leftarrow {}^{24}0 \parallel MS(EA,1)$.		225
lbzu	RT, D(RA)	Load byte from EA = (RA|0) + EXTS(D) and pad left with zeroes, $(RT) \leftarrow {}^{24}0 \parallel MS(EA,1)$. Update the base address, $(RA) \leftarrow EA$.		226
lbzux	RT, RA, RB	Load byte from EA = (RA|0) + (RB) and pad left with zeroes, $(RT) \leftarrow {}^{24}0 \parallel MS(EA,1)$. Update the base address, $(RA) \leftarrow EA$.		227
lbzx	RT, RA, RB	Load byte from EA = (RA|0) + (RB) and pad left with zeroes, $(RT) \leftarrow {}^{24}0 \parallel MS(EA,1)$.		228
lha	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and sign extend, $(RT) \leftarrow EXTS(MS(EA,2))$.		229

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
lhau	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and sign extend, (RT) ← EXTS(MS(EA,2)). Update the base address, (RA) ← EA.		230
lhaux	RT, RA, RB	Load halfword from EA = (RA|0) + (RB) and sign extend, (RT) ← EXTS(MS(EA,2)). Update the base address, (RA) ← EA.		231
lhax	RT, RA, RB	Load halfword from EA = (RA|0) + (RB) and sign extend, (RT) ← EXTS(MS(EA,2)).		232
lhbrx	RT, RA, RB	Load halfword from EA = (RA|0) + (RB), then reverse byte order and pad left with zeroes, (RT) ← $^{16}0$ || MS(EA+1,1) || MS(EA,1).		233
lhz	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) ← $^{16}0$ || MS(EA,2).		234
lhzu	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) ← $^{16}0$ || MS(EA,2). Update the base address, (RA) ← EA.		235
lhzux	RT, RA, RB	Load halfword from EA = (RA|0) + (RB) and pad left with zeroes, (RT) ← $^{16}0$ || MS(EA,2). Update the base address, (RA) ← EA.		236
lhzx	RT, RA, RB	Load halfword from EA = (RA|0) + (RB) and pad left with zeroes, (RT) ← $^{16}0$ || MS(EA,2).		237
li	RT, IM	Load immediate. (RT) ← EXTS(IM) *Extended mnemonic for* **addi RT,0,value**		164
lis	RT, IM	Load immediate shifted. (RT) ← (IM || $^{16}0$) *Extended mnemonic for* **addis RT,0,value**		167
lmw	RT, D(RA)	Load multiple words starting from EA = (RA|0) + EXTS(D). Place into consecutive registers RT through GPR(31). RA is not altered unless RA = GPR(31).		238
lswi	RT, RA, NB	Load consecutive bytes from EA=(RA|0). Number of bytes n=32 if NB=0, else n=NB. Stack bytes into words in CEIL(n/4) consecutive registers starting with RT, to R_{FINAL} ← ((RT + CEIL(n/4) – 1) % 32). GPR(0) is consecutive to GPR(31). RA is not altered unless RA = R_{FINAL}.		239
lswx	RT, RA, RB	Load consecutive bytes from EA=(RA|0)+(RB). Number of bytes n=XER[TBC]. Stack bytes into words in CEIL(n/4) consecutive registers starting with RT, to R_{FINAL} ← ((RT + CEIL(n/4) – 1) % 32). GPR(0) is consecutive to GPR(31). RA is not altered unless RA = R_{FINAL}. RB is not altered unless RB = R_{FINAL}. If n=0, content of RT is undefined.		241
lwarx	RT, RA, RB	Load word from EA = (RA|0) + (RB) and place in RT, (RT) ← MS(EA,4). Set the Reservation bit.		243
lwbrx	RT, RA, RB	Load word from EA = (RA|0) + (RB) then reverse byte order, (RT) ← MS(EA+3,1) || MS(EA+2,1) || MS(EA+1,1) || MS(EA,1).		244
lwz	RT, D(RA)	Load word from EA = (RA|0) + EXTS(D) and place in RT, (RT) ← MS(EA,4).		245

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
lwzu	RT, D(RA)	Load word from EA = (RA|0) + EXTS(D) and place in RT, (RT) $\leftarrow$ MS(EA,4). Update the base address, (RA) $\leftarrow$ EA.		246
lwzux	RT, RA, RB	Load word from EA = (RA|0) + (RB) and place in RT, (RT) $\leftarrow$ MS(EA,4). Update the base address, (RA) $\leftarrow$ EA.		247
lwzx	RT, RA, RB	Load word from EA = (RA|0) + (RB) and place in RT, (RT) $\leftarrow$ MS(EA,4).		248
macchw	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		249
macchw.			CR[CR0]	
macchwo			XER[SO, OV]	
macchwo.			CR[CR0] XER[SO, OV]	
macchws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \,\|\|^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		250
macchws.			CR[CR0]	
macchwso			XER[SO, OV]	
macchwso.			CR[CR0] XER[SO, OV]	
macchwsu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow (temp_{1:32} \vee {}^{32}temp_0)$		251
macchwsu.			CR[CR0]	
macchwsuo			XER[SO, OV]	
macchwsuo.			CR[CR0] XER[SO, OV]	
macchwu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		252
macchwu.			CR[CR0]	
macchwuo			XER[SO, OV]	
macchwuo.			CR[CR0] XER[SO, OV]	
machhw	RT, RA, RB	$prod_{0:15} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		253
machhw.			CR[CR0]	
machhwo			XER[SO, OV]	
machhwo.			CR[CR0] XER[SO, OV]	
machhws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \,\|\|^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		254
machhws.			CR[CR0]	
machhwso			XER[SO, OV]	
machhwso.			CR[CR0] XER[SO, OV]	
machhwsu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow (temp_{1:32} \vee {}^{32}temp_0)$		255
machhwsu.			CR[CR0]	
machhwsuo			XER[SO, OV]	
machhwsuo.			CR[CR0] XER[SO, OV]	
machhwu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		256
machhwu.			CR[CR0]	
machhwuo			XER[SO, OV]	
machhwuo.			CR[CR0] XER[SO, OV]	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
maclhw	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		257
maclhw.			CR[CR0]	
maclhwo			XER[SO, OV]	
maclhwo.			CR[CR0] XER[SO, OV]	
maclhws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		258
maclhws.			CR[CR0]	
maclhwso			XER[SO, OV]	
maclhwso.			CR[CR0] XER[SO, OV]	
maclhwsu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow (temp_{1:32} \vee {}^{32}temp_0)$		259
maclhwsu.			CR[CR0]	
maclhwsuo			XER[SO, OV]	
maclhwsuo.			CR[CR0] XER[SO, OV]	
maclhwu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		260
maclhwu.			CR[CR0]	
maclhwuo			XER[SO, OV]	
maclhwuo.			CR[CR0] XER[SO, OV]	
mcrf	BF, BFA	Move CR field, (CR[CRn]) $\leftarrow$ (CR[CRm]) where m $\leftarrow$ BFA and n $\leftarrow$ BF.		261
mcrxr	BF	Move XER[0:3] into field CRn, where n$\leftarrow$BF. CR[CRn] $\leftarrow$ (XER[SO, OV, CA]). (XER[SO, OV, CA]) $\leftarrow$ $^{3}0$.		262
mfcr	RT	Move from CR to RT, (RT) $\leftarrow$ (CR).		263
mfdcr	RT, DCRN	Move from DCR to RT, (RT) $\leftarrow$ (DCR(DCRN)).		264
mfmsr	RT	Move from MSR to RT, (RT) $\leftarrow$ (MSR).		265

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
mfccr0 **mfctr** **mfdac1** **mfdac2** **mfdear** **mfdbcr0** **mfdbcr1** **mfdbsr** **mfdccr** **mfdcwr** **mfdvc1** **mfdvc2** **mfesr** **mfevpr** **mfiac1** **mfiac2** **mfiac3** **mfiac4** **mficcr** **mficdbdr** **mflr** **mfpid** **mfpit** **mfpvr** **mfsgr** **mfsler** **mfsprg0** **mfsprg1** **mfsprg2** **mfsprg3** **mfsprg4** **mfsprg5** **mfsprg6** **mfsprg7** **mfsrr0** **mfsrr1** **mfsrr2** **mfsrr3** **mfsu0r** **mftcr** **mftsr** **mfxer** **mfzpr**	RT	Move from special purpose register (SPR) SPRN. *Extended mnemonic for* **mfspr RT,SPRN** See *Table 10-3* on page 354 for listing of valid SPRN values.		266
mfspr	RT, SPRN	Move from SPR to RT, (RT) ← (SPR(SPRN)).		266
mftb	RT, TBRN	Move from TBR to RT, (RT) ← (TBR(TBRN)).		268
mftb	RT	Move the contents of TBL into RT, (RT) ← (TBL) *Extended mnemonic for* **mftb RT,TBL**		268
mftbu	RT	Move the contents of TBU into RT, (RT) ← (TBU) *Extended mnemonic for* **mftb RT,TBU**		268
mr	RT, RS	Move register. (RT) ← (RS) *Extended mnemonic for* **or RT,RS,RS**		293
mr.		*Extended mnemonic for* **or. RT,RS,RS**	CR[CR0]	
mtcr	RS	Move to Condition Register. *Extended mnemonic for* **mtcrf 0xFF,RS**		269

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
mtcrf	FXM, RS	Move some or all of the contents of RS into CR as specified by FXM field, mask $\leftarrow$ $^4(FXM_0) \parallel {}^4(FXM_1) \parallel \ldots \parallel {}^4(FXM_6) \parallel {}^4(FXM_7)$. (CR)$\leftarrow$((RS) $\wedge$ mask) $\vee$ (CR) $\wedge$ $\neg$mask).		269
mtdcr	DCRN, RS	Move to DCR from RS, (DCR(DCRN)) $\leftarrow$ (RS).		270
mtmsr	RS	Move to MSR from RS, (MSR) $\leftarrow$ (RS).		271
mtccr0 **mtctr** **mtdac1** **mtdac2** **mtdbcr0** **mtdbcr1** **mtdbsr** **mtdccr** **mtdear** **mtdcwr** **mtdvc1** **mtdvc2** **mtesr** **mtevpr** **mtiac1** **mtiac2** **mtiac3** **mtiac4** **mticcr** **mticdbdr** **mtlr** **mtpid** **mtpit** **mtpvr** **mtsgr** **mtsler** **mtsprg0** **mtsprg1** **mtsprg2** **mtsprg3** **mtsprg4** **mtsprg5** **mtsprg6** **mtsprg7** **mtsrr0** **mtsrr1** **mtsrr2** **mtsrr3** **mtsu0r** **mttbl** **mttbu** **mttcr** **mttsr** **mtxer** **mtzpr**	RS	Move to SPR SPRN. *Extended mnemonic for* **mtspr SPRN,RS** See *Table 10-3* on page 354 for listing of valid SPRN values.		272
mtspr	SPRN, RS	Move to SPR from RS, (SPR(SPRN)) $\leftarrow$ (RS).		272
mulchw	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ signed		274
mulchw.			CR[CR0]	
mulchwu	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ unsigned		275
mulchwu.			CR[CR0]	
mulhhw	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ signed		276
mulhhw.			CR[CR0]	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
mulhhwu	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{0:15}$ x $(RB)_{0:15}$ unsigned		277
mulhhwu.			CR[CR0]	
mullhw	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{16:31}$ x $(RB)_{16:31}$ signed		280
mullhw.			CR[CR0]	
mullhwu	RT, RA, RB	$(RT)_{16:31} \leftarrow (RA)_{16:31}$ x $(RB)_{16:31}$ unsigned		281
mullhwu.			CR[CR0]	
mulhw	RT, RA, RB	Multiply (RA) and (RB), signed. Place high-order result in RT. $prod_{0:63} \leftarrow (RA) \times (RB)$ (signed). $(RT) \leftarrow prod_{0:31}$.		278
mulhw.			CR[CR0]	
mulhwu	RT, RA, RB	Multiply (RA) and (RB), unsigned. Place high-order result in RT. $prod_{0:63} \leftarrow (RA) \times (RB)$ (unsigned). $(RT) \leftarrow prod_{0:31}$.		279
mulhwu.			CR[CR0]	
mulli	RT, RA, IM	Multiply (RA) and IM, signed. Place low-order result in RT. $prod_{0:47} \leftarrow (RA) \times IM$ (signed) $(RT) \leftarrow prod_{16:47}$		282
mullw	RT, RA, RB	Multiply (RA) and (RB), signed. Place low-order result in RT. $prod_{0:63} \leftarrow (RA) \times (RB)$ (signed). $(RT) \leftarrow prod_{32:63}$.		283
mullw.			CR[CR0]	
mullwo			XER[SO, OV]	
mullwo.			CR[CR0] XER[SO, OV]	
nand	RA, RS, RB	NAND (RS) with (RB). Place result in RA.		284
nand.			CR[CR0]	
neg	RT, RA	Negative (twos complement) of RA. $(RT) \leftarrow \neg(RA) + 1$		285
neg.			CR[CR0]	
nego			XER[SO, OV]	
nego.			CR[CR0] XER[SO, OV]	
nmacchw	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{16:31}$ x $(RB)_{0:15})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		286
nmacchw.			CR[CR0]	
nmacchwo			XER[SO, OV]	
nmacchwo.			CR[CR0] XER[SO, OV]	
nmacchws	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{16:31}$ x $(RB)_{0:15})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ if $((nprod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		287
nmacchws.			CR[CR0]	
nmacchwso			XER[SO, OV]	
nmacchwso.			CR[CR0] XER[SO, OV]	
nmachhw	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{0:15}$ x $(RB)_{0:15})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		288
nmachhw.			CR[CR0]	
nmachhwo			XER[SO, OV]	
nmachhwo.			CR[CR0] XER[SO, OV]	
nmachhws	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{0:15}$ x $(RB)_{0:15})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ if $((nprod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		289
nmachhws.			CR[CR0]	
nmachhwso			XER[SO, OV]	
nmachhwso.			CR[CR0] XER[SO, OV]	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
nmaclhw	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{16:31} \times (RB)_{16:31})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ if $((nprod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		290
nmaclhw.			CR[CR0]	
nmaclhwo			XER[SO, OV]	
nmaclhwo.			CR[CR0] XER[SO, OV]	
nmachlws	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{0:15} \times (RB)_{0:15})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ if $((nprod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		291
nmachlws.			CR[CR0]	
nmachlwso			XER[SO, OV]	
nmachlwso.			CR[CR0] XER[SO, OV]	
nop		Preferred no-op, triggers optimizations based on no-ops. *Extended mnemonic for* **ori 0,0,0**		295
nor	RA, RS, RB	NOR (RS) with (RB). Place result in RA.		292
nor.			CR[CR0]	
not	RA, RS	Complement register. $(RA) \leftarrow \neg(RS)$ *Extended mnemonic for* **nor RA,RS,RS**		292
not.		*Extended mnemonic for* **nor. RA,RS,RS**	CR[CR0]	
or	RA, RS, RB	OR (RS) with (RB). Place result in RA.		293
or.			CR[CR0]	
orc	RA, RS, RB	OR (RS) with $\neg$(RB). Place result in RA.		294
orc.			CR[CR0]	
ori	RA, RS, IM	OR (RS) with $({}^{16}0 \parallel IM)$. Place result in RA.		295
oris	RA, RS, IM	OR (RS) with $(IM \parallel {}^{16}0)$. Place result in RA.		296
rfci		Return from critical interrupt $(PC) \leftarrow (SRR2)$. $(MSR) \leftarrow (SRR3)$.		297
rfi		Return from interrupt. $(PC) \leftarrow (SRR0)$. $(MSR) \leftarrow (SRR1)$.		298
rlwimi	RA, RS, SH, MB, ME	Rotate left word immediate, then insert according to mask. $r \leftarrow ROTL((RS), SH)$ $m \leftarrow MASK(MB, ME)$ $(RA) \leftarrow (r \wedge m) \vee ((RA) \wedge \neg m)$		299
rlwimi.			CR[CR0]	
rlwinm	RA, RS, SH, MB, ME	Rotate left word immediate, then AND with mask. $r \leftarrow ROTL((RS), SH)$ $m \leftarrow MASK(MB, ME)$ $(RA) \leftarrow (r \wedge m)$		300
rlwinm.			CR[CR0]	
rlwnm	RA, RS, RB, MB, ME	Rotate left word, then AND with mask. $r \leftarrow ROTL((RS), (RB)_{27:31})$ $m \leftarrow MASK(MB, ME)$ $(RA) \leftarrow (r \wedge m)$		302
rlwnm.			CR[CR0]	
rotlw	RA, RS, RB	Rotate left. $(RA) \leftarrow ROTL((RS), (RB)_{27:31})$ *Extended mnemonic for* **rlwnm RA,RS,RB,0,31**		302
rotlw.		*Extended mnemonic for* **rlwnm. RA,RS,RB,0,31**	CR[CR0]	

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
rotlwi	RA, RS, n	Rotate left immediate. $(RA) \leftarrow$ ROTL((RS), n) *Extended mnemonic for* **rlwinm RA,RS,n,0,31**		300
rotlwi.		*Extended mnemonic for* **rlwinm. RA,RS,n,0,31**	CR[CR0]	
rotrwi	RA, RS, n	Rotate right immediate. $(RA) \leftarrow$ ROTL((RS), 32–n) *Extended mnemonic for* **rlwinm RA,RS,32–n,0,31**		300
rotrwi.		*Extended mnemonic for* **rlwinm. RA,RS,32–n,0,31**	CR[CR0]	
sc		System call exception is generated. $(SRR1) \leftarrow (MSR)$ $(SRR0) \leftarrow (PC)$ $PC \leftarrow EVPR_{0:15} \Vert$ x'0C00' (MSR[WE, PR, EE, PE, DR, IR]) $\leftarrow 0$		303
slw	RA, RS, RB	Shift left (RS) by $(RB)_{27:31}$. $n \leftarrow (RB)_{27:31}$. $r \leftarrow$ ROTL((RS), n). if $(RB)_{26} = 0$ then $m \leftarrow$ MASK(0, 31 – n) else $m \leftarrow {}^{32}0$. $(RA) \leftarrow r \wedge m$.		304
slw.			CR[CR0]	
slwi	RA, RS, n	Shift left immediate. (n < 32) $(RA)_{0:31-n} \leftarrow (RS)_{n:31}$ $(RA)_{32-n:31} \leftarrow {}^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,n,0,31–n**		300
slwi.		*Extended mnemonic for* **rlwinm. RA,RS,n,0,31–n**	CR[CR0]	
sraw	RA, RS, RB	Shift right algebraic (RS) by $(RB)_{27:31}$. $n \leftarrow (RB)_{27:31}$. $r \leftarrow$ ROTL((RS), 32 – n). if $(RB)_{26} = 0$ then $m \leftarrow$ MASK(n, 31) else $m \leftarrow {}^{32}0$. $s \leftarrow (RS)_0$. $(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$. $XER[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		305
sraw.			CR[CR0]	
srawi	RA, RS, SH	Shift right algebraic (RS) by SH. $n \leftarrow SH$. $r \leftarrow$ ROTL((RS), 32 – n). $m \leftarrow$ MASK(n, 31). $s \leftarrow (RS)_0$. $(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$. $XER[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		306
srawi.			CR[CR0]	
srw	RA, RS, RB	Shift right (RS) by $(RB)_{27:31}$. $n \leftarrow (RB)_{27:31}$. $r \leftarrow$ ROTL((RS), 32 – n). if $(RB)_{26} = 0$ then $m \leftarrow$ MASK(n, 31) else $m \leftarrow {}^{32}0$. $(RA) \leftarrow r \wedge m$.		307
srw.			CR[CR0]	
srwi	RA, RS, n	Shift right immediate. (n < 32) $(RA)_{n:31} \leftarrow (RS)_{0:31-n}$ $(RA)_{0:n-1} \leftarrow {}^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,32–n,n,31**		300
srwi.		*Extended mnemonic for* **rlwinm. RA,RS,32–n,n,31**	CR[CR0]	
stb	RS, D(RA)	Store byte $(RS)_{24:31}$ in memory at EA = $(RA \vert 0)$ + EXTS(D).		308
stbu	RS, D(RA)	Store byte $(RS)_{24:31}$ in memory at EA = $(RA \vert 0)$ + EXTS(D). Update the base address, $(RA) \leftarrow EA$.		309

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
stbux	RS, RA, RB	Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + (RB). Update the base address, (RA) $\leftarrow$ EA.		310
stbx	RS, RA, RB	Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + (RB).		311
sth	RS, D(RA)	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + EXTS(D).		312
sthbrx	RS, RA, RB	Store halfword $(RS)_{16:31}$ byte-reversed in memory at EA = (RA|0) + (RB). MS(EA, 2) $\leftarrow$ $(RS)_{24:31} \parallel (RS)_{16:23}$		313
sthu	RS, D(RA)	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + EXTS(D). Update the base address, (RA) $\leftarrow$ EA.		314
sthux	RS, RA, RB	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + (RB). Update the base address, (RA) $\leftarrow$ EA.		315
sthx	RS, RA, RB	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + (RB).		316
stmw	RS, D(RA)	Store consecutive words from RS through GPR(31) in memory starting at EA = (RA|0) + EXTS(D).		317
stswi	RS, RA, NB	Store consecutive bytes in memory starting at EA=(RA|0). Number of bytes n=32 if NB=0, else n=NB. Bytes are unstacked from CEIL(n/4) consecutive registers starting with RS. GPR(0) is consecutive to GPR(31).		318
stswx	RS, RA, RB	Store consecutive bytes in memory starting at EA=(RA|0)+(RB). Number of bytes n=XER[TBC]. Bytes are unstacked from CEIL(n/4) consecutive registers starting with RS. GPR(0) is consecutive to GPR(31).		319
stw	RS, D(RA)	Store word (RS) in memory at EA = (RA|0) + EXTS(D).		321
stwbrx	RS, RA, RB	Store word (RS) byte-reversed in memory at EA = (RA|0) + (RB). MS(EA, 4) $\leftarrow$ $(RS)_{24:31} \parallel (RS)_{16:23} \parallel$ $(RS)_{8:15} \parallel (RS)_{0:7}$		322

stwcx.	RS, RA, RB	Store word (RS) in memory at EA = (RA|0) + (RB)
		only if reservation bit is set.
		if RESERVE = 1 then
		MS(EA, 4) $\leftarrow$ (RS)
		RESERVE $\leftarrow$ 0
		(CR[CR0]) $\leftarrow$ $^{2}0 \parallel 1 \parallel XER_{SO}$
		else
		(CR[CR0]) $\leftarrow$ $^{2}0 \parallel 0 \parallel XER_{SO}$.
				323

stwu	RS, D(RA)	Store word (RS) in memory at EA = (RA|0) + EXTS(D). Update the base address, (RA) $\leftarrow$ EA.		324
stwux	RS, RA, RB	Store word (RS) in memory at EA = (RA|0) + (RB). Update the base address, (RA) $\leftarrow$ EA.		325
stwx	RS, RA, RB	Store word (RS) in memory at EA = (RA|0) + (RB).		326

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
sub	RT, RA, RB	Subtract (RB) from (RA). (RT) ← ¬(RB) + (RA) + 1. *Extended mnemonic for* **subf RT,RB,RA**		327
sub.		*Extended mnemonic for* **subf. RT,RB,RA**	CR[CR0]	
subo		*Extended mnemonic for* **subfo RT,RB,RA**	XER[SO, OV]	
subo.		*Extended mnemonic for* **subfo. RT,RB,RA**	CR[CR0] XER[SO, OV]	
subc	RT, RA, RB	Subtract (RB) from (RA). (RT) ← ¬(RB) + (RA) + 1. Place carry-out in XER[CA]. *Extended mnemonic for* **subfc RT,RB,RA**		328
subc.		*Extended mnemonic for* **subfc. RT,RB,RA**	CR[CR0]	
subco		*Extended mnemonic for* **subfco RT,RB,RA**	XER[SO, OV]	
subco.		*Extended mnemonic for* **subfco. RT,RB,RA**	CR[CR0] XER[SO, OV]	
subf	RT, RA, RB	Subtract (RA) from (RB). (RT) ← ¬(RA) + (RB) + 1.		327
subf.			CR[CR0]	
subfo			XER[SO, OV]	
subfo.			CR[CR0] XER[SO, OV]	
subfc	RT, RA, RB	Subtract (RA) from (RB). (RT) ← ¬(RA) + (RB) + 1. Place carry-out in XER[CA].		328
subfc.			CR[CR0]	
subfco			XER[SO, OV]	
subfco.			CR[CR0] XER[SO, OV]	
subfe	RT, RA, RB	Subtract (RA) from (RB) with carry-in. (RT) ← ¬(RA) + (RB) + XER[CA]. Place carry-out in XER[CA].		329
subfe.			CR[CR0]	
subfeo			XER[SO, OV]	
subfeo.			CR[CR0] XER[SO, OV]	
subfic	RT, RA, IM	Subtract (RA) from EXTS(IM). (RT) ← ¬(RA) + EXTS(IM) + 1. Place carry-out in XER[CA].		330
subfme	RT, RA, RB	Subtract (RA) from (–1) with carry-in. (RT) ← ¬(RA) + (–1) + XER[CA]. Place carry-out in XER[CA].		331
subfme.			CR[CR0]	
subfmeo			XER[SO, OV]	
subfmeo.			CR[CR0] XER[SO, OV]	
subfze	RT, RA, RB	Subtract (RA) from zero with carry-in. (RT) ← ¬(RA) + XER[CA]. Place carry-out in XER[CA].		332
subfze.			CR[CR0]	
subfzeo			XER[SO, OV]	
subfzeo.			CR[CR0] XER[SO, OV]	
subi	RT, RA, IM	Subtract EXTS(IM) from (RA|0). Place result in RT. *Extended mnemonic for* **addi RT,RA,–IM**		164

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
subic	RT, RA, IM	Subtract EXTS(IM) from (RA). Place result in RT. Place carry-out in XER[CA]. *Extended mnemonic for* **addic RT,RA,–IM**		165
subic.	RT, RA, IM	Subtract EXTS(IM) from (RA). Place result in RT. Place carry-out in XER[CA]. *Extended mnemonic for* **addic. RT,RA,–IM**	CR[CR0]	166
subis	RT, RA, IM	Subtract (IM $\|\|$ $^{16}0$) from (RA|0). Place result in RT. *Extended mnemonic for* **addis RT,RA,–IM**		167
sync		Synchronization. All instructions that precede **sync** complete before any instructions that follow **sync** begin. When **sync** completes, all storage accesses initiated prior to **sync** will have completed.		333
tlbia		All TLB entries are invalidated and become unavailable for translation by clearing the valid (V) bit in the TLBHI portion of each TLB entry. The rest of the TLB fields unmodified.		334
tlbre	RT, RA,WS	If WS = 0: Load TLBHI of the selected TLB entry into RT. Load PID with the contents of the TID field of the selected TLB entry. (RT) ← TLBHI[(RA)] (PID) ← TLB[(RA)]TID If WS = 1: Load TLBLO portion of the selected TLB entry into RT. (RT) ← TLBLO[(RA)]		335
tlbrehi	RT, RA	Load TLBHI of the selected TLB entry into RT. Load PID with the contents of the TID field of the selected TLB entry. (RT) ← TLBHI[(RA)] (PID) ← TLB[(RA)]TID *Extended mnemonic for* **tlbre RT,RA,0**		335
tlbrelo	RT, RA	Load TLBLO of the selected TLB entry into RT. (RT) ← TLBLO[(RA)] *Extended mnemonic for* **tlbre RT,RA,1**		335
tlbsx	RT, RA, RB	Search the TLB for a valid entry that translates the EA. EA = (RA|0) + (RB). If found, (RT) ← Index of TLB entry. If not found, (RT) Undefined.		337
tlbsx.		If found, (RT) ← Index of TLB entry. $CR[CR0]_{EQ}$ ← 1. If not found, (RT) Undefined. $CR[CR0]_{EQ}$ ← 1.	$CR[CR0]_{LT,GT,SO}$	
tlbsync		**tlbsync** does not complete until all previous TLB-update instructions executed by this processor have been received and completed by all other processors. For the PPC405, **tlbsync** is a no-op.		338
tlbwe	RS, RA,WS	If WS = 0: Write TLBHI of the selected TLB entry from RS. Write the TID field of the selected TLB entry from the PID register. TLBHI[(RA)] ← (RS) TLB[(RA)]TID ← (PID)24:31 If WS = 1: Write TLBLO portion of the selected TLB entry from RS. TLBLO[(RA)] ← (RS)		339

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
tlbwehi	RS, RA	Write TLBHI of the selected TLB entry from RS. Write the TID field of the selected TLB entry from the PID register. TLBHI[(RA)] ← (RS) TLB[(RA)]TID ← (PID)24:31 *Extended mnemonic for* **tlbwe RS,RA,0**		339
tlbwelo	RS, RA	Write TLBLO of the selected TLB entry from RS. TLBLO[(RA)] ← (RS) *Extended mnemonic for* **tlbwe RS,RA,1**		339
trap		Trap unconditionally. *Extended mnemonic for* **tw 31,0,0**		341
tweq	RA, RB	Trap if (RA) equal to (RB). *Extended mnemonic for* **tw 4,RA,RB**		
twge		Trap if (RA) greater than or equal to (RB). *Extended mnemonic for* **tw 12,RA,RB**		
twgt		Trap if (RA) greater than (RB). *Extended mnemonic for* **tw 8,RA,RB**		
twle		Trap if (RA) less than or equal to (RB). *Extended mnemonic for* **tw 20,RA,RB**		
twlge		Trap if (RA) logically greater than or equal to (RB). *Extended mnemonic for* **tw 5,RA,RB**		
twlgt		Trap if (RA) logically greater than (RB). *Extended mnemonic for* **tw 1,RA,RB**		
twlle		Trap if (RA) logically less than or equal to (RB). *Extended mnemonic for* **tw 6,RA,RB**		
twllt		Trap if (RA) logically less than (RB). *Extended mnemonic for* **tw 2,RA,RB**		
twlng		Trap if (RA) logically not greater than (RB). *Extended mnemonic for* **tw 6,RA,RB**		
twlnl		Trap if (RA) logically not less than (RB). *Extended mnemonic for* **tw 5,RA,RB**		
twlt		Trap if (RA) less than (RB). *Extended mnemonic for* **tw 16,RA,RB**		
twne		Trap if (RA) not equal to (RB). *Extended mnemonic for* **tw 24,RA,RB**		
twng		Trap if (RA) not greater than (RB). *Extended mnemonic for* *tw 20,RA,RB*		
twnl		Trap if (RA) not less than (RB). *Extended mnemonic for* **tw 12,RA,RB**		
tw	TO, RA, RB	Trap exception is generated if, comparing (RA) with (RB), any condition specified by TO is true.		341

Table A-1. PPC405 Instruction Syntax Summary (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
tweqi	RA, IM	Trap if (RA) equal to EXTS(IM). *Extended mnemonic for* **twi 4,RA,IM**		344
twgei		Trap if (RA) greater than or equal to EXTS(IM). *Extended mnemonic for* **twi 12,RA,IM**		
twgti		Trap if (RA) greater than EXTS(IM). *Extended mnemonic for* **twi 8,RA,IM**		
twlei		Trap if (RA) less than or equal to EXTS(IM). *Extended mnemonic for* **twi 20,RA,IM**		
twlgei		Trap if (RA) logically greater than or equal to EXTS(IM). *Extended mnemonic for* **wi 5,RA,IM**		
twlgti		Trap if (RA) logically greater than EXTS(IM). *Extended mnemonic for* **twi 1,RA,IM**		
twllei		Trap if (RA) logically less than or equal to EXTS(IM). *Extended mnemonic for* **twi 6,RA,IM**		
twllti		Trap if (RA) logically less than EXTS(IM). *Extended mnemonic for* **twi 2,RA,IM**		
twlngi		Trap if (RA) logically not greater than EXTS(IM). *Extended mnemonic for* **twi 6,RA,IM**		
twlnli		Trap if (RA) logically not less than EXTS(IM). *Extended mnemonic for* **twi 5,RA,IM**		
twlti		Trap if (RA) less than EXTS(IM). *Extended mnemonic for* **twi 16,RA,IM**		
twnei		Trap if (RA) not equal to EXTS(IM). *Extended mnemonic for* **twi 24,RA,IM**		
twngi		Trap if (RA) not greater than EXTS(IM). *Extended mnemonic for* **twi 20,RA,IM**		
twnli		Trap if (RA) not less than EXTS(IM). *Extended mnemonic for* **twi 12,RA,IM**		
twi	TO, RA, IM	Trap exception is generated if, comparing (RA) with EXTS(IM), any condition specified by TO is true.		344
wrtee	RS	Write value of RS_{16} to MSR[EE].		347
wrteei	E	Write value of E to MSR[EE].		348
xor	RA, RS, RB	XOR (RS) with (RB). Place result in RA.		349
xor.			CR[CR0]	
xori	RA, RS, IM	XOR (RS) with ($^{16}0 \parallel$ IM). Place result in RA.		350
xoris	RA, RS, IM	XOR (RS) with (IM $\parallel$ $^{16}0$). Place result in RA.		351

A.3 List of Instructions—by Opcode

All instructions are four bytes long and word aligned. All instructions have a primary opcode field (shown as field OPCD in Figure A-1 through Figure A-9, beginning on page -360) in bits 0:5. Some instructions also have a secondary opcode field (shown as field XO in Figure A-1 through Figure A-9). PPC405 instructions, sorted by primary and secondary opcode, are listed in Table A-2.

The “Form” indicated in the table refers to the arrangement of valid field combinations within the four-byte instruction. See “Instruction Formats,” on page -357, for the field layouts of each form.

Form X has a 10-bit secondary opcode field, while form XO uses only the low-order 9-bits of that field. Form XO uses the high-order secondary opcode bit (the tenth bit) as a variable; therefore, every form XO instruction really consumes two secondary opcodes from the 10-bit secondary-opcode space. The implicitly consumed secondary opcode is listed in parentheses for form XO instructions in the following table.

Table A-2. PPC405 Instructions by Opcode

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
3		D	**twi**	TO, RA, IM	344
4	8	X	**mulhhwu**	RT, RA, RB	277
			mulhhwu.		
4	12 (524)	XO	**machhwu**	RT, RA, RB	256
			machhwu.		
			machhwuo		
			machhwuo.		
4	40	X	**mulhhw**	RT, RA, RB	276
			mulhhw.		
4	44 (556)	XO	**machhw**	RT, RA, RB	253
			machhw.		
			machhwo		
			machhwo.		
4	46 (558)	XO	**nmachhw**	RT, RA, RB	288
			nmachhw.		
			nmachhwo		
			nmachhwo		
4	76 (588)	XO	**machhwsu**	RT, RA, RB	255
			machhwsu.		
			machhwsuo		
			machhwsuo.		
4	108 (620)	XO	**machhws**	RT, RA, RB	254
			machhws.		
			machhwso		
			machhwso.		
4	110 (622)	XO	**nmachhws**	RT, RA, RB	289
			nmachhws.		
			nmachhwso		
			nmachhwso.		

Table A-2. PPC405 Instructions by Opcode (Continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
4	136	X	**mulchwu** **mulchwu.**	RT, RA, RB	275
4	140 (652)	XO	**macchwu** **macchwu.** **macchwuo** **machhwuo.**	RT, RA, RB	252
4	168	X	**mulchw** **mulchw.**	RT, RA, RB	274
4	172 (684)	XO	**macchw** **macchw.** **macchwo** **macchwo.**	RT, RA, RB	249
4	174 (686)	XO	**nmacchw** **nmacchw.** **nmacchwo** **nmacchwo.**	RT, RA, RB	286
4	204 (716)	XO	**macchwsu** **macchwsu.** **macchwsuo** **macchwsuo.**	RT, RA, RB	251
4	236 (748)	XO	**macchws** **macchws.** **macchwso** **macchwso.**	RT, RA, RB	250
4	238 (750)	XO	**nmacchws** **nmacchws.** **nmacchwso** **nmacchwso.**	RT, RA, RB	287
4	392	X	**mullhwu** **mullhwu.**	RT, RA, RB	281
4	396 (908)	XO	**maclhwu** **maclhwu.** **maclhwuo** **maclhwuo.**	RT, RA, RB	260
4	424	X	**mullhw** **mullhw.**	RT, RA, RB	280
4	428 (940)	XO	**maclhw** **maclhw.** **maclhwo** **maclhwo.**	RT, RA, RB	257

Table A-2. PPC405 Instructions by Opcode (Continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
4	430 (942)	XO	**nmaclhw**	RT, RA, RB	290
			nmaclhw.		
			nmaclhwo		
			nmaclhwo.		
4	492 (972)	XO	**maclhws**	RT, RA, RB	258
			maclhws.		
			maclhwso		
			maclhwso.		
4	460 (1004)	XO	**maclhwsu**	RT, RA, RB	259
			maclhwsu.		
			maclhwsuo		
			maclhwsuo.		
4	494 (1006)	XO	**nmaclhws**	RT, RA, RB	291
			nmaclhws.		
			nmaclhwso		
			nmaclhwso.		
7		D	**mulli**	RT, RA, IM	282
8		D	**subfic**	RT, RA, IM	330
10		D	**cmpli**	BF, 0, RA, IM	191
11		D	**cmpi**	BF, 0, RA, IM	189
12		D	**addic**	RT, RA, IM	165
13		D	**addic.**	RT, RA, IM	166
14		D	**addi**	RT, RA, IM	164
15		D	**addis**	RT, RA, IM	167
16		B	**bc**	BO, BI, target	175
			bca		
			bcl		
			bcla		
17		SC	**sc**		303
18		I	**b**	target	174
			ba		
			bl		
			bla		
19	0	XL	**mcrf**	BF, BFA	261
19	16	XL	**bclr**	BO, BI	184
			bclrl		
19	33	XL	**crnor**	BT, BA, BB	197
19	50	XL	**rfi**		298
19	51	XL	**rfci**		297
19	129	XL	**crandc**	BT, BA, BB	194
19	150	XL	**isync**		224
19	193	XL	**crxor**	BT, BA, BB	200
19	225	XL	**crnand**	BT, BA, BB	196
19	257	XL	**crand**	BT, BA, BB	193
19	289	XL	**creqv**	BT, BA, BB	195

Table A-2. PPC405 Instructions by Opcode (Continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
19	417	XL	**crorc**	BT, BA, BB	199
19	449	XL	**cror**	BT, BA, BB	198
19	528	XL	**bcctr**	BO, BI	181
			bcctrl		
20		M	**rlwimi**	RA, RS, SH, MB, ME	299
			rlwimi.		
21		M	**rlwinm**	RA, RS, SH, MB, ME	300
			rlwinm.		
23		M	**rlwnm**	RA, RS, RB, MB, ME	302
			rlwnm.		
24		D	**ori**	RA, RS, IM	295
25		D	**oris**	RA, RS, IM	296
26		D	**xori**	RA, RS, IM	350
27		D	**xoris**	RA, RS, IM	351
28		D	**andi.**	RA, RS, IM	172
29		D	**andis.**	RA, RS, IM	173
31	0	X	**cmp**	BF, 0, RA, RB	188
31	4	X	**tw**	TO, RA, RB	341
31	8 (520)	XO	**subfc**	RT, RA, RB	328
			subfc.		
			subfco		
			subfco.		
31	10 (522)	XO	**addc**	RT, RA, RB	162
			addc.		
			addco		
			addco.		
31	11	XO	**mulhwu**	RT, RA, RB	281
			mulhwu.		
31	19	X	**mfcr**	RT	263
31	20	X	**lwarx**	RT, RA, RB	243
31	23	X	**lwzx**	RT, RA, RB	248
31	24	X	**slw**	RA, RS, RB	304
			slw.		
31	26	X	**cntlzw**	RA, RS	192
			cntlzw.		
31	28	X	**and**	RA, RS, RB	170
			and.		
31	32	X	**cmpl**	BF, 0, RA, RB	190
31	40 (552)	XO	**subf**	RT, RA, RB	327
			subf.		
			subfo		
			subfo.		
31	54	X	**dcbst**	RA, RB	205
31	55	X	**lwzux**	RT, RA, RB	247
31	60	X	**andc**	RA, RS, RB	171
			andc.		

Table A-2. PPC405 Instructions by Opcode (Continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	75	XO	**mulhw**	RT, RA, RB	280
			mulhw.		
31	83	X	**mfmsr**	RT	265
31	86	X	**dcbf**	RA, RB	203
31	87	X	**lbzx**	RT, RA, RB	228
31	104 (616)	XO	**neg**	RT, RA	285
			neg.		
			nego		
			nego.		
31	119	X	**lbzux**	RT, RA, RB	227
31	124	X	**nor**	RA, RS, RB	292
			nor.		
31	131	X	**wrtee**	RS	347
31	136 (648)	XO	**subfe**	RT, RA, RB	329
			subfe.		
			subfeo		
			subfeo.		
31	138 (650)	XO	**adde**	RT, RA, RB	163
			adde.		
			addeo		
			addeo.		
31	144	XFX	**mtcrf**	FXM, RS	269
31	146	X	**mtmsr**	RS	271
31	150	X	**stwcx.**	RS, RA, RB	323
31	151	X	**stwx**	RS, RA, RB	326
31	163	X	**wrteei**	E	348
31	183	X	**stwux**	RS, RA, RB	325
31	200 (712)	XO	**subfze**	RT, RA, RB	332
			subfze.		
			subfzeo		
			subfzeo.		
31	202 (714)	XO	**addze**	RT, RA	169
			addze.		
			addzeo		
			addzeo.		
31	215	X	**stbx**	RS, RA, RB	311
31	232 (744)	XO	**subfme**	RT, RA, RB	331
			subfme.		
			subfmeo		
			subfmeo.		
31	234 (746)	XO	**addme**	RT, RA	168
			addme.		
			addmeo		
			addmeo.		

Table A-2. PPC405 Instructions by Opcode (Continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	235 (747)	XO	**mullw**	RT, RA, RB	283
			mullw.		
			mullwo		
			mullwo.		
31	246	X	**dcbtst**	RA,RB	207
31	247	X	**stbux**	RS, RA, RB	310
31	262	X	**icbt**	RA, RB	220
31	266 (778)	XO	**add**	RT, RA, RB	161
			add.		
			addo		
			addo.		
31	278	X	**dcbt**	RA, RB	206
31	279	X	**lhzx**	RT, RA, RB	237
31	284	X	**eqv**	RA, RS, RB	216
			eqv.		
31	311	X	**lhzux**	RT, RA, RB	236
31	316	X	**xor**	RA, RS, RB	349
			xor.		
31	323	XFX	**mfdcr**	RT, DCRN	264
31	339	XFX	**mfspr**	RT, SPRN	266
31	343	X	**lhax**	RT, RA, RB	232
31	370	X	**tlbia**		334
31	371	XFX	**mftb**	RT, TBRN	268
31	375	X	**lhaux**	RT, RA, RB	231
31	407	X	**sthx**	RS, RA, RB	315
31	412	X	**orc**	RA, RS, RB	294
			orc.		
31	439	X	**sthux**	RS, RA, RB	315
31	444	X	**or**	RA, RS, RB	293
			or.		
31	451	XFX	**mtdcr**	DCRN, RS	270
31	454	X	**dccci**	RA, RB	210
31	459 (971)	XO	**divwu**	RT, RA, RB	214
			divwu.		
			divwuo		
			divwuo.		
31	467	XFX	**mtspr**	SPRN, RS	272
31	470	X	**dcbi**	RA, RB	204
31	476	X	**nand**	RA, RS, RB	284
			nand.		
31	486	X	**dcread**	RT, RA, RB	211
31	491 (1003)	XO	**divw**	RT, RA, RB	213
			divw.		
			divwo		
			divwo.		
31	512	X	**mcrxr**	BF	262
31	533	X	**lswx**	RT, RA, RB	241

Table A-2. PPC405 Instructions by Opcode (Continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	534	X	**lwbrx**	RT, RA, RB	244
31	536	X	**srw** **srw.**	RA, RS, RB	307
31	566	X	**tlbsync**		338
31	597	X	**lswi**	RT, RA, NB	239
31	598	X	**sync**		333
31	661	X	**stswx**	RS, RA, RB	319
31	662	X	**stwbrx**	RS, RA, RB	322
31	725	X	**stswi**	RS, RA, NB	318
31	758	X	**dcba**	RA, RB	201
31	790	X	**lhbrx**	RT, RA, RB	233
31	792	X	**sraw** **sraw.**	RA, RS, RB	305
31	824	X	**srawi** **srawi.**	RA, RS, SH	306
31	854	X	**eieio**		215
31	914	X	**tlbsx** **tlbsx.**	RT, RA, RB	337
31	918	X	**sthbrx**	RS, RA, RB	313
31	922	X	**extsh** **extsh.**	RA, RS	218
31	946	X	**tlbre**	RT, RA,WS	335
31	954	X	**extsb** **extsb.**	RA, RS	217
31	966	X	**iccci**	RA, RB	221
31	978	X	**tlbwe**	RS, RA,WS	339
31	982	X	**icbi**	RA, RB	219
31	998	X	**icread**	RA, RB	222
31	1014	X	**dcbz**	RA, RB	208
32		D	**lwz**	RT, D(RA)	245
33		D	**lwzu**	RT, D(RA)	246
34		D	**lbz**	RT, D(RA)	225
35		D	**lbzu**	RT, D(RA)	226
36		D	**stw**	RS, D(RA)	321
37		D	**stwu**	RS, D(RA)	324
38		D	**stb**	RS, D(RA)	308
39		D	**stbu**	RS, D(RA)	309
40		D	**lhz**	RT, D(RA)	234
41		D	**lhzu**	RT, D(RA)	235
42		D	**lha**	RT, D(RA)	229
43		D	**lhau**	RT, D(RA)	230
44		D	**sth**	RS, D(RA)	312
45		D	**sthu**	RS, D(RA)	314
46		D	**lmw**	RT, D(RA)	238
47		D	**stmw**	RS, D(RA)	317

Appendix B. Instructions by Category

Instruction Set on page 157 contains detailed descriptions of the instructions, their operands, and notation.

Table B-1 summarizes the instruction categories in the PPC405 instruction set. The instructions within each category are listed in subsequent tables.

Table B-1. PPC405 Instruction Set Categories

Storage Reference	load, store
Arithmetic and Logical	add, subtract, negate, multiply, divide, and, andc, or, orc, xor, nand, nor, xnor, sign extension, count leading zeros, multiply accumulate
Comparison	compare, compare logical, compare immediate
Branch	branch, branch conditional, branch to LR, branch to CTR
CR Logical	crand, crandc, cror, crorc, crnand, crnor, crxor, crxnor, move CR field
Rotate/Shift	rotate and insert, rotate and mask, shift left, shift right
Cache Control	invalidate, touch, zero, flush, store, read
Interrupt Control	write to external interrupt enable bit, move to/from MSR, return from interrupt, return from critical interrupt
Processor Management	system call, synchronize, trap, move to/from DCRs, move to/from SPRs, move to/from CR

B.1 Implementation-Specific Instructions

To meet the functional requirements of processors for embedded systems and real-time applications, the PPC405 defines the implementation-specific instructions summarized in Table B-2.

Table B-2. Implementation-specific Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
dccci	RA, RB	Invalidate the data cache congruence class associated with the effective address (EA) (RA|0) + (RB).		210
dcread	RT, RA, RB	Read either tag or data information from the data cache congruence class associated with the EA (RA|0) + (RB). Place the results in RT.		211
iccci	RA, RB	Invalidate instruction cache.		221
icread	RA, RB	Read either tag or data information from the instruction cache congruence class associated with the EA (RA|0) + (RB). Place the results in ICDBDR.		222
macchw	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		222
macchw.			CR[CR0]	
macchwo			XER[SO, OV]	
macchwo.			CR[CR0] XER[SO, OV]	
macchws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		250
macchws.			CR[CR0]	
macchwso			XER[SO, OV]	
macchwso.			CR[CR0] XER[SO, OV]	

Table B-2. Implementation-specific Instructions (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
macchwsu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31}$ x $(RB)_{0:15}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow (temp_{1:32} \vee {}^{32}temp_0)$		251
macchwsu.			CR[CR0]	
macchwsuo			XER[SO, OV]	
macchwsuo.			CR[CR0] XER[SO, OV]	
macchwu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31}$ x $(RB)_{0:15}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		252
macchwu.			CR[CR0]	
macchwuo			XER[SO, OV]	
macchwuo.			CR[CR0] XER[SO, OV]	
machhw	RT, RA, RB	$prod_{0:15} \leftarrow (RA)_{16:31}$ x $(RB)_{0:15}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		253
machhw.			CR[CR0]	
machhwo			XER[SO, OV]	
machhwo.			CR[CR0] XER[SO, OV]	
machhws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{0:15}$ x $(RB)_{0:15}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \,\|\, {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		254
machhws.			CR[CR0]	
machhwso			XER[SO, OV]	
machhwso.			CR[CR0] XER[SO, OV]	
machhwsu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{0:15}$ x $(RB)_{0:15}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow (temp_{1:32} \vee {}^{32}temp_0)$		255
machhwsu.			CR[CR0]	
machhwsuo			XER[SO, OV]	
machhwsuo.			CR[CR0] XER[SO, OV]	
machhwu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{0:15}$ x $(RB)_{0:15}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		256
machhwu.			CR[CR0]	
machhwuo			XER[SO, OV]	
machhwuo.			CR[CR0] XER[SO, OV]	
maclhw	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31}$ x $(RB)_{16:31}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		257
maclhw.			CR[CR0]	
maclhwo			XER[SO, OV]	
maclhwo.			CR[CR0] XER[SO, OV]	

Table B-2. Implementation-specific Instructions (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
maclhws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31}$ x $(RB)_{16:31}$ signed $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		258
maclhws.			CR[CR0]	
maclhwso			XER[SO, OV]	
maclhwso.			CR[CR0] XER[SO, OV]	
maclhwsu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31}$ x $(RB)_{16:31}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow (temp_{1:32} \vee {}^{32}temp_0)$		259
maclhwsu.			CR[CR0]	
maclhwsuo			XER[SO, OV]	
maclhwsuo.			CR[CR0] XER[SO, OV]	
maclhwu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31}$ x $(RB)_{16:31}$ unsigned $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		260
maclhwu.			CR[CR0]	
maclhwuo			XER[SO, OV]	
maclhwuo.			CR[CR0] XER[SO, OV]	
mulchw	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{16:31}$ x $(RB)_{0:15}$ signed		274
mulchw.			CR[CR0]	
mulchwu	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{16:31}$ x $(RB)_{0:15}$ unsigned		275
mulchwu.			CR[CR0]	
mulhhw	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{0:15}$ x $(RB)_{0:15}$ signed		276
mulhhw.			CR[CR0]	
mulhhwu	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{0:15}$ x $(RB)_{0:15}$ unsigned		277
mulhhwu.			CR[CR0]	
mullhw	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{16:31}$ x $(RB)_{16:31}$ signed		280
mullhw.			CR[CR0]	
mullhwu	RT, RA, RB	$(RT)_{16:31} \leftarrow (RA)_{0:15}$ x $(RB)_{16:31}$ unsigned		281
mullhwu.			CR[CR0]	
nmacchw	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{16:31}$ x $(RB)_{0:15})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		286
nmacchw.			CR[CR0]	
nmacchwo			XER[SO, OV]	
nmacchwo.			CR[CR0] XER[SO, OV]	
nmacchws	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{16:31}$ x $(RB)_{0:15})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ if $((nprod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		287
nmacchws.			CR[CR0]	
nmacchwso			XER[SO, OV]	
nmacchwso.			CR[CR0] XER[SO, OV]	

Table B-2. Implementation-specific Instructions (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
nmachhw	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{0:15} \times (RB)_{0:15})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		288
nmachhw.			CR[CR0]	
nmachhwo			XER[SO, OV]	
nmachhwo.			CR[CR0] XER[SO, OV]	
nmachhws	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{0:15} \times (RB)_{0:15})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ if $((nprod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		289
nmachhws.			CR[CR0]	
nmachhwso			XER[SO, OV]	
nmachhwso.			CR[CR0] XER[SO, OV]	
nmaclhw	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{16:31} \times (RB)_{16:31})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		290
nmaclhw.			CR[CR0]	
nmaclhwo			XER[SO, OV]	
nmaclhwo.			CR[CR0] XER[SO, OV]	
nmaclhws	RT, RA, RB	$nprod_{0:31} \leftarrow -((RA)_{16:31} \times (RB)_{16:31})$ signed $temp_{0:32} \leftarrow nprod_{0:31} + (RT)$ if $((nprod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		291
nmaclhws.			CR[CR0]	
nmaclhwso			XER[SO, OV]	
nmaclhwso.			CR[CR0] XER[SO, OV]	

B.2 Instructions in the PowerPC Embedded Environment

To meet the functional requirements of processors for embedded systems and real-time applications, the PowerPC Embedded Environment defines instructions that are not part of the PowerPC Architecture.

Table B-3 summarizes the PPC405 instructions in the PowerPC Embedded Environment.

Table B-3. Instructions in the IBM PowerPC Embedded Environment

Mnemonic	Operands	Function	Other Registers Changed	Page
dcba	RA, RB	Speculatively establish the data cache block which contains the EA (RA|0) + (RB).		201
dcbf	RA, RB	Flush (store, then invalidate) the data cache block which contains the EA (RA|0) + (RB).		203
dcbi	RA, RB	Invalidate the data cache block which contains the EA (RA|0) + (RB).		204
dcbst	RA, RB	Store the data cache block which contains the EA (RA|0) + (RB).		205
dcbt	RA, RB	Load the data cache block which contains the EA (RA|0) + (RB).		206
dcbtst	RA,RB	Load the data cache block which contains the EA (RA|0) + (RB).		207

Table B-3. Instructions in the IBM PowerPC Embedded Environment (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
dcbz	RA, RB	Zero the data cache block which contains the EA (RA|0) + (RB).		208
eieio		Storage synchronization. All loads and stores that precede the **eieio** instruction complete before any loads and stores that follow the instruction access main storage. Implemented as **sync**, which is more restrictive.		215
icbi	RA, RB	Invalidate the instruction cache block which contains the EA (RA|0) + (RB).		219
icbt	RA, RB	Load the instruction cache block which contains the EA (RA|0) + (RB).		220
isync		Synchronize execution context by flushing the prefetch queue.		224
mfdcr	RT, DCRN	Move from DCR to RT, (RT) $\leftarrow$ (DCR(DCRN)).		264
mfmsr	RT	Move from MSR to RT, (RT) $\leftarrow$ (MSR).		265
mfspr	RT, SPRN	Move from SPR to RT, (RT) $\leftarrow$ (SPR(SPRN)). Privileged for all SPRs except LR, CTR, TBHU, TBLU, and XER.		266
mftb	RT	Move the contents of a Time Base Register (TBR) into RT, TBRN $\leftarrow$ $TBRF_{5:9} \parallel TBRF_{0:4}$ (RT) $\leftarrow$ (TBR(TBRN))		268
mtdcr	DCRN, RS	Move to DCR from RS, (DCR(DCRN)) $\leftarrow$ (RS).		270
mtmsr	RS	Move to MSR from RS, (MSR) $\leftarrow$ (RS).		271
mtspr	SPRN, RS	Move to SPR from RS, (SPR(SPRN)) $\leftarrow$ (RS). Privileged for all SPRs except LR, CTR, and XER.		272
rfci		Return from critical interrupt (PC) $\leftarrow$ (SRR2). (MSR) $\leftarrow$ (SRR3).		297
rfi		Return from interrupt. (PC) $\leftarrow$ (SRR0). (MSR) $\leftarrow$ (SRR1).		298
tlbia		All of the entries in the TLB are invalidated and become unavailable for translation by clearing the valid (V) bit in the TLBHI portion of each TLB entry. The rest of the fields in the TLB entries are unmodified.		334
tlbre	RT, RA,WS	If WS = 0: Load TLBHI portion of the selected TLB entry into RT. Load the PID register with the contents of the TID field of the selected TLB entry. (RT) $\leftarrow$ TLBHI[(RA)] (PID) $\leftarrow$ $TLB[(RA)]_{TID}$ If WS = 1: Load TLBLO portion of the selected TLB entry into RT. (RT) $\leftarrow$ TLBLO[(RA)]		335

Table B-3. Instructions in the IBM PowerPC Embedded Environment (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
tlbsx	RT,RA,RB	Search the TLB array for a valid entry which translates the EA EA = (RA|0) + (RB). If found, (RT) ← Index of TLB entry. If not found, (RT) Undefined.		337
tlbsx.		If found, (RT) ← Index of TLB entry. $CR[CR0]_{EQ}$ ← 1. If not found, (RT) Undefined. $CR[CR0]_{EQ}$ ← 1.	$CR[CR0]_{LT,GT,SO}$	
tlbsync		**tlbsync** does not complete until all previous TLB-update instructions executed by this processor have been received and completed by all other processors. For the PPC405, **tlbsync** is a no-op.		338
tlbwe	RS, RA,WS	If WS = 0: Write TLBHI portion of the selected TLB entry from RS. Write the TID field of the selected TLB entry from the PID register. TLBHI[(RA)] ← (RS) $TLB[(RA)]_{TID}$ ← $(PID)_{24:31}$ If WS = 1: Write TLBLO portion of the selected TLB entry from RS. TLBLO[(RA)] ← (RS)		339
wrtee	RS	Write value of RS_{16} to MSR[EE].		347
wrteei	E	Write value of E to MSR[EE].		348

B.3 Privileged Instructions

Table B-4 lists instructions that are under control of the MSR[PR] bit. These instructions are not allowed to be executed when MSR[PR] = 1:

Table B-4. Privileged Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
dcbi	RA, RB	Invalidate the data cache block which contains the EA (RA|0) + (RB).		204
dccci	RA, RB	Invalidate the data cache congruence class associated with the EA (RA|0) + (RB).		210
dcread	RT, RA, RB	Read either tag or data information from the data cache congruence class associated with the EA (RA|0) + (RB). Place the results in RT.		211
iccci	RA, RB	Invalidate instruction cache.		221
icread	RA, RB	Read either tag or data information from the instruction cache congruence class associated with the EA (RA|0) + (RB). Place the results in ICDBDR.		221
mfdcr	RT, DCRN	Move from DCR to RT, (RT) ← (DCR(DCRN)).		264
mfmsr	RT	Move from MSR to RT, (RT) ← (MSR).		265

Table B-4. Privileged Instructions (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
mfspr	RT, SPRN	Move from SPR to RT, (RT) ← (SPR(SPRN)). Privileged for all SPRs except LR, CTR, TBHU, TBLU, and XER.		272
mtdcr	DCRN, RS	Move to DCR from RS, (DCR(DCRN)) ← (RS).		270
mtmsr	RS	Move to MSR from RS, (MSR) ← (RS).		271
mtspr	SPRN, RS	Move to SPR from RS, (SPR(SPRN)) ← (RS). Privileged for all SPRs except LR, CTR, and XER.		272
rfci		Return from critical interrupt (PC) ← (SRR2). (MSR) ← (SRR3).		297
rfi		Return from interrupt. (PC) ← (SRR0). (MSR) ← (SRR1).		298
tlbre	RT, RA,WS	If WS = 0: Load TLBHI portion of the selected TLB entry into RT. Load the PID register with the contents of the TID field of the selected TLB entry. (RT) ← TLBHI[(RA)] (PID) ← $TLB[(RA)]_{TID}$ If WS = 1: Load TLBLO portion of the selected TLB entry into RT. (RT) ← TLBLO[(RA)]		335
tlbsx	RT,RA,RB	Search the TLB array for a valid entry which translates the EA EA = (RA|0) + (RB). If found, (RT) ← Index of TLB entry. If not found, (RT) Undefined.		337
tlbsx.		If found, (RT) ← Index of TLB entry. $CR[CR0]_{EQ}$ ← 1. If not found, (RT) Undefined. $CR[CR0]_{EQ}$ ← 1.	$CR[CR0]_{LT,GT,SO}$	
tlbwe	RS, RA,WS	If WS = 0: Write TLBHI portion of the selected TLB entry from RS. Write the TID field of the selected TLB entry from the PID register. TLBHI[(RA)] ← (RS) $TLB[(RA)]_{TID}$ ← $(PID)_{24:31}$ If WS = 1: Write TLBLO portion of the selected TLB entry from RS. TLBLO[(RA)] ← (RS)		339
wrtee	RS	Write value of RS_{16} to the External Enable bit (MSR[EE]).		347
wrteei	E	Write value of E to the External Enable bit (MSR[EE]).		348

B.4 Assembler Extended Mnemonics

In the appendix "Assembler Extended Mnemonics" of the PowerPC Architecture, it is required that a PowerPC assembler support at least a minimal set of extended mnemonics. These mnemonics encode to the opcodes of other instructions; the only benefit of extended mnemonics is improved usability. Code using extended mnemonics can be easier to write and to understand. Table B-5 lists the extended mnemonics required for the PPC405.

Note for every Branch Conditional mnemonic:

Bit 4 of the BO field provides a hint about the most likely outcome of a conditional branch. (*Branch Prediction* on page 52 describes branch prediction). Assemblers should set BO_4 = 0 unless a specific reason exists otherwise. In the BO field values specified in the following table, BO_4 = 0 has always been assumed. The assembler must allow the programmer to specify branch prediction. To do this, the assembler will support a suffix to every conditional branch mnemonic, as follows:

+ Predict branch to be taken.

– Predict branch not to be taken.

As specific examples, **bc** also could be coded as **bc+** or **bc**–, and **bne** also could be coded **bne+** or **bne**–. These alternate codings set BO_4 = 1 only if the requested prediction differs from the standard prediction (see *Branch Prediction* on page 52).

Table B-5. Extended Mnemonics for PPC405

Mnemonic	Operands	Function	Other Registers Changed	Page
bctr		Branch unconditionally to address in CTR. *Extended mnemonic for* **bcctr 20,0**		181
bctrl		*Extended mnemonic for* **bcctrl 20,0**	(LR) ← CIA + 4	
bdnz	target	Decrement CTR. Branch if CTR ≠ 0. *Extended mnemonic for* **bc 16,0,target**		175
bdnza		*Extended mnemonic for* **bca 16,0,target**		
bdnzl		*Extended mnemonic for* **bcl 16,0,target**	(LR) ← CIA + 4.	
bdnzla		*Extended mnemonic for* **bcla 16,0,target**	(LR) ← CIA + 4.	
bdnzlr		Decrement CTR. Branch, if CTR ≠ 0,to address in LR. *Extended mnemonic for* **bclr 16,0**		175
bdnzlrl		*Extended mnemonic* for **bclrl 16,0**	(LR) ← CIA + 4.	

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bdnzf	cr_bit, target	Decrement CTR. Branch if CTR ≠ 0 AND CR_{cr_bit} = 0. *Extended mnemonic for* **bc 0,cr_bit,target**		175
bdnzfa		*Extended mnemonic for* **bca 0,cr_bit,target**		
bdnzfl		*Extended mnemonic for* **bcl 0,cr_bit,target**	(LR) ← CIA + 4.	
bdnzfla		*Extended mnemonic for* **bcla 0,cr_bit,target**	(LR) ← CIA + 4.	
bdnzflr	cr_bit	Decrement CTR. Branch, if CTR ≠ 0 AND CR_{cr_bit} = 0, to address in LR. *Extended mnemonic for* **bclr 0,cr_bit**		175
bdnzflrl		*Extended mnemonic for* **bclrl 0,cr_bit**	(LR) ← CIA + 4.	
bdnzt	cr_bit, target	Decrement CTR. Branch if CTR ≠ 0 AND CR_{cr_bit} = 1. *Extended mnemonic for* **bc 8,cr_bit,target**		175
bdnzta		*Extended mnemonic for* **bca 8,cr_bit,target**		
bdnztl		*Extended mnemonic for* **bcl 8,cr_bit,target**	(LR) ← CIA + 4.	
bdnztla		*Extended mnemonic for* **bcla 8,cr_bit,target**	(LR) ← CIA + 4.	
bdnztlr	cr_bit	Decrement CTR. Branch, if CTR ≠ 0 AND CR_{cr_bit} = 1, to address in LR. *Extended mnemonic for* **bclr 8,cr_bit**		175
bdnztlrl		*Extended mnemonic for* **bclrl 8,cr_bit**	(LR) ← CIA + 4.	
bdz	target	Decrement CTR. Branch if CTR = 0. *Extended mnemonic for* **bc 18,0,target**		175
bdza		*Extended mnemonic* for **bca 18,0,target**		
bdzl		*Extended mnemonic for* **bcl 18,0,target**	(LR) ← CIA + 4.	
bdzla		*Extended mnemonic for* **bcla 18,0,target**	(LR) ← CIA + 4.	
bdzlr		Decrement CTR. Branch, if CTR = 0, to address in LR. *Extended mnemonic for* **bclr 18,0**		175
bdzlrl		*Extended mnemonic for* **bclrl 18,0**	(LR) ← CIA + 4.	

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bdzf	cr_bit, target	Decrement CTR. Branch if CTR = 0 AND CR_{cr_bit} = 0. *Extended mnemonic for* **bc 2,cr_bit,target**		175
bdzfa		*Extended mnemonic for* **bca 2,cr_bit,target**		
bdzfl		*Extended mnemonic for* **bcl 2,cr_bit,target**	(LR) ← CIA + 4.	
bdzfla		*Extended mnemonic for* **bcla 2,cr_bit,target**	(LR) ← CIA + 4.	
bdzflr	cr_bit	Decrement CTR. Branch, if CTR = 0 AND CR_{cr_bit} = 0 to address in LR. *Extended mnemonic for* **bclr 2,cr_bit**		175
bdzflrl		*Extended mnemonic for* **bclrl 2,cr_bit**	(LR) ← CIA + 4.	
bdzt	cr_bit, target	Decrement CTR. Branch if CTR = 0 AND CR_{cr_bit} = 1. *Extended mnemonic for* **bc 10,cr_bit,target**		175
bdzta		*Extended mnemonic for* **bca 10,cr_bit,target**		
bdztl		*Extended mnemonic for* **bcl 10,cr_bit,target**	(LR) ← CIA + 4.	
bdztla		*Extended mnemonic for* **bcla 10,cr_bit,target**	(LR) ← CIA + 4.	
bdztlr	cr_bit	Decrement CTR. Branch, if CTR = 0 AND CR_{cr_bit} = 1, to address in LR. *Extended mnemonic for* **bclr 10,cr_bit**		184
bdztlrl		*Extended mnemonic for* **bclrl 10,cr_bit**	(LR) ← CIA + 4.	
beq	[cr_field,] target	Branch if equal. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 12,4∗cr_field+2,target**		184
beqa		*Extended mnemonic for* **bca 12,4∗cr_field+2,target**		
beql		*Extended mnemonic for* **bcl 12,4∗cr_field+2,target**	(LR) ← CIA + 4.	
beqla		*Extended mnemonic for* **bcla 12,4∗cr_field+2,target**	(LR) ← CIA + 4.	
beqctr	[cr_field]	Branch, if equal, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4∗cr_field+2**		181
beqctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+2**	(LR) ← CIA + 4.	

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
beqlr	[cr_field]	Branch, if equal, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 12,4∗cr_field+2**		184
beqlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+2**	(LR) ← CIA + 4.	
bf	cr_bit, target	Branch if CR_{cr_bit} = 0. *Extended mnemonic for* **bc 4,cr_bit,target**		175
bfa		*Extended mnemonic for* **bca 4,cr_bit,target**		
bfl		*Extended mnemonic for* **bcl 4,cr_bit,target**	(LR) ← CIA + 4.	
bfla		*Extended mnemonic for* **bcla 4,cr_bit,target**	(LR) ← CIA + 4.	
bfctr	cr_bit	Branch, if CR_{cr_bit} = 0, to address in CTR. *Extended mnemonic for* **bcctr 4,cr_bit**		181
bfctrl		*Extended mnemonic for* **bcctrl 4,cr_bit**	(LR) ← CIA + 4.	
bflr	cr_bit	Branch, if CR_{cr_bit} = 0, to address in LR. *Extended mnemonic for* **bclr 4,cr_bit**		184
bflrl		*Extended mnemonic for* **bclrl 4,cr_bit**	(LR) ← CIA + 4.	
bge	[cr_field,] target	Branch if greater than or equal. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+0,target**		175
bgea		*Extended mnemonic for* **bca 4,4∗cr_field+0,target**		
bgel		*Extended mnemonic for* **bcl 4,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bgela		*Extended mnemonic for* **bcla 4,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bgectr	[cr_field]	Branch, if greater than or equal, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+0**		181
bgectrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+0**	(LR) ← CIA + 4.	
bgelr	[cr_field]	Branch, if greater than or equal, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+0**		184
bgelrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+0**	(LR) ← CIA + 4.	

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bgt	[cr_field,] target	Branch if greater than. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 12,4∗cr_field+1,target**		175
bgta		*Extended mnemonic for* **bca 12,4∗cr_field+1,target**		
bgtl		*Extended mnemonic for* **bcl 12,4∗cr_field+1,target**	(LR) ← CIA + 4.	
bgtla		*Extended mnemonic for* **bcla 12,4∗cr_field+1,target**	(LR) ← CIA + 4.	
bgtctr	[cr_field]	Branch, if greater than, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4∗cr_field+1**		181
bgtctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+1**	(LR) ← CIA + 4.	
bgtlr	[cr_field]	Branch, if greater than, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 12,4∗cr_field+1**		184
bgtlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+1**	(LR) ← CIA + 4.	
ble	[cr_field,] target	Branch if less than or equal. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+1,target**		175
blea		*Extended mnemonic for* **bca 4,4∗cr_field+1,target**		
blel		*Extended mnemonic for* **bcl 4,4∗cr_field+1,target**	(LR) ← CIA + 4.	
blela		*Extended mnemonic for* **bcla 4,4∗cr_field+1,target**	(LR) ← CIA + 4.	
blectr	[cr_field]	Branch, if less than or equal, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+1**		181
blectrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+1**	(LR) ← CIA + 4.	
blelr	[cr_field]	Branch, if less than or equal, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+1**		184
blelrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+1**	(LR) ← CIA + 4.	
blr		Branch, unconditionally, to address in LR. *Extended mnemonic for* **bclr 20,0**		184
blrl		*Extended mnemonic for* **bclrl 20,0**	(LR) ← CIA + 4.	

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
blt	[cr_field,] target	Branch if less than. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 12,4∗cr_field+0,target**		175
blta		*Extended mnemonic for* **bca 12,4∗cr_field+0,target**		
bltl		*Extended mnemonic for* **bcl 12,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bltla		*Extended mnemonic for* **bcla 12,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bltctr	[cr_field]	Branch, if less than, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4∗cr_field+0**		181
bltctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+0**	(LR) ← CIA + 4.	
bltlr	[cr_field]	Branch, if less than, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 12,4∗cr_field+0**		184
bltlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+0**	(LR) ← CIA + 4.	
bne	[cr_field,] target	Branch if not equal. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+2,target**		175
bnea		*Extended mnemonic for* **bca 4,4∗cr_field+2,target**		
bnel		*Extended mnemonic for* **bcl 4,4∗cr_field+2,target**	(LR) ← CIA + 4.	
bnela		*Extended mnemonic for* **bcla 4,4∗cr_field+2,target**	(LR) ← CIA + 4.	
bnectr	[cr_field]	Branch, if not equal, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+2**		181
bnectrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+2**	(LR) ← CIA + 4.	
bnelr	[cr_field]	Branch, if not equal, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+2**		184
bnelrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+2**	(LR) ← CIA + 4.	

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bng	[cr_field,] target	Branch, if not greater than. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+1,target**		175
bnga		*Extended mnemonic for* **bca 4,4∗cr_field+1,target**		
bngl		*Extended mnemonic for* **bcl 4,4∗cr_field+1,target**	(LR) ← CIA + 4.	
bngla		*Extended mnemonic for* **bcla 4,4∗cr_field+1,target**	(LR) ← CIA + 4.	
bngctr	[cr_field]	Branch, if not greater than, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+1**		181
bngctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+1**	(LR) ← CIA + 4.	
bnglr	[cr_field]	Branch, if not greater than, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+1**		184
bnglrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+1**	(LR) ← CIA + 4.	
bnl	[cr_field,] target	Branch if not less than. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+0,target**		175
bnla		*Extended mnemonic for* **bca 4,4∗cr_field+0,target**		
bnll		*Extended mnemonic for* **bcl 4,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bnlla		*Extended mnemonic for* **bcla 4,4∗cr_field+0,target**	(LR) ← CIA + 4.	
bnlctr	[cr_field]	Branch, if not less than, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+0**		181
bnlctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+0**	(LR) ← CIA + 4.	
bnllr	[cr_field]	Branch, if not less than, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+0**		184
bnllrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+0**	(LR) ← CIA + 4.	

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bns	[cr_field,] target	Branch if not summary overflow. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+3,target**		175
bnsa		*Extended mnemonic for* **bca 4,4∗cr_field+3,target**		
bnsl		*Extended mnemonic for* **bcl 4,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bnsla		*Extended mnemonic for* **bcla 4,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bnsctr	[cr_field]	Branch, if not summary overflow, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+3**		181
bnsctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+3**	(LR) ← CIA + 4.	
bnslr	[cr_field]	Branch, if not summary overflow, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+3**		184
bnslrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+3**	(LR) ← CIA + 4.	
bnu	[cr_field,] target	Branch if not unordered. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+3,target**		175
bnua		*Extended mnemonic for* **bca 4,4∗cr_field+3,target**		
bnul		*Extended mnemonic for* **bcl 4,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bnula		*Extended mnemonic for* **bcla 4,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bnuctr	[cr_field]	Branch, if not unordered, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+3**		181
bnuctrl		*Extended mnemonic for* **bcctrl 4,4∗cr_field+3**	(LR) ← CIA + 4.	
bnulr	[cr_field]	Branch, if not unordered, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+3**		184
bnulrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+3**	(LR) ← CIA + 4.	

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bso	[cr_field,] target	Branch if summary overflow. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 12,4∗cr_field+3,target**		175
bsoa		*Extended mnemonic for* **bca 12,4∗cr_field+3,target**		
bsol		*Extended mnemonic for* **bcl 12,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bsola		*Extended mnemonic for* **bcla 12,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bsoctr	[cr_field]	Branch, if summary overflow, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4∗cr_field+3**		181
bsoctrl		*Extended mnemonic for* **bcctrl 12,4∗cr_field+3**	(LR) ← CIA + 4.	
bsolr	[cr_field]	Branch, if summary overflow, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 12,4∗cr_field+3**		184
bsolrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+3**	(LR) ← CIA + 4.	
bt	cr_bit, target	Branch if $CR_{cr\_bit} = 1$. *Extended mnemonic for* **bc 12,cr_bit,target**		175
bta		*Extended mnemonic for* **bca 12,cr_bit,target**		
btl		*Extended mnemonic for* **bcl 12,cr_bit,target**	(LR) ← CIA + 4.	
btla		*Extended mnemonic for* **bcla 12,cr_bit,target**	(LR) ← CIA + 4.	
btctr	cr_bit	Branch if $CR_{cr\_bit} = 1$, to address in CTR. *Extended mnemonic for* **bcctr 12,cr_bit**		181
btctrl		*Extended mnemonic for* **bcctrl 12,cr_bit**	(LR) ← CIA + 4.	
btlr	cr_bit	Branch, if $CR_{cr\_bit} = 1$, to address in LR. *Extended mnemonic for* **bclr 12,cr_bit**		184
btlrl		*Extended mnemonic for* **bclrl 12,cr_bit**	(LR) ← CIA + 4.	

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bun	[cr_field,] target	Branch if unordered. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bc 12,4∗cr_field+3,target**		175
buna		*Extended mnemonic for* **bca 12,4∗cr_field+3,target**		
bunl		*Extended mnemonic for* **bcl 12,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bunla		*Extended mnemonic for* **bcla 12,4∗cr_field+3,target**	(LR) ← CIA + 4.	
bunctr	[cr_field]	Branch, if unordered, to address in CTR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4∗cr_field+3**		181
bunctrl		*Extended mnemonic* for **bcctrl 12,4∗cr_field+3**	(LR) ← CIA + 4.	
bunlr	[cr_field]	Branch, if unordered, to address in LR. Use CR0 if cr_field is omitted. *Extended mnemonic for* **bclr 12,4∗cr_field+3**		184
bunlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+3**	(LR) ← CIA + 4.	
clrlwi	RA, RS, n	Clear left immediate. (n < 32) $(RA)_{0:n-1} \leftarrow {}^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,0,n,31**		300
clrlwi.		*Extended mnemonic for* **rlwinm. RA,RS,0,n,31**	CR[CR0]	
clrlslwi	RA, RS, b, n	Clear left and shift left immediate. (n ≤ b < 32) $(RA)_{b-n:31-n} \leftarrow_{n} (RS)_{b:31}$ $(RA)_{32-n:31} \leftarrow {}^{n}0$ $(RA)_{0:b-n-1} \leftarrow {}^{b-n}0$ *Extended mnemonic for* **rlwinm RA,RS,n,b−n,31−n**		300
clrlslwi.		*Extended mnemonic for* **rlwinm. RA,RS,n,b−n,31−n**	CR[CR0]	
clrrwi	RA, RS, n	Clear right immediate. (n < 32) $(RA)_{32-n:31} \leftarrow {}^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,0,0,31−n**		300
clrrwi.		*Extended mnemonic for* **rlwinm. RA,RS,0,0,31−n**	CR[CR0]	
cmplw	[BF,] RA, RB	Compare Logical Word. Use CR0 if BF is omitted. *Extended mnemonic for* **cmpl BF,0,RA,RB**		190
cmplwi	[BF,] RA, IM	Compare Logical Word Immediate. Use CR0 if BF is omitted. *Extended mnemonic for* **cmpli BF,0,RA,IM**		191
cmpw	[BF,] RA, RB	Compare Word. Use CR0 if BF is omitted. *Extended mnemonic for* **cmp BF,0,RA,RB**		188

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
cmpwi	[BF,] RA, IM	Compare Word Immediate. Use CR0 if BF is omitted. *Extended mnemonic for* **cmpi BF,0,RA,IM**		189
crclr	bx	Condition register clear. *Extended mnemonic for* **crxor bx,bx,bx**		200
crmove	bx, by	Condition register move. *Extended mnemonic for* **cror bx,by,by**		198
crnot	bx, by	Condition register not. *Extended mnemonic for* **crnor bx,by,by**		197
crset	bx	Condition register set. *Extended mnemonic for* **creqv bx,bx,bx**		195
extlwi	RA, RS, n, b	Extract and left justify immediate. (n > 0) $(RA)_{0:n-1} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{n:31} \leftarrow {}^{32-n}0$ *Extended mnemonic for* **rlwinm RA,RS,b,0,n–1**		300
extlwi.		*Extended mnemonic for* **rlwinm. RA,RS,b,0,n–1**	CR[CR0]	
extrwi	RA, RS, n, b	Extract and right justify immediate. (n > 0) $(RA)_{32-n:31} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{0:31-n} \leftarrow {}^{32-n}0$ *Extended mnemonic for* **rlwinm RA,RS,b+n,32–n,31**		300
extrwi.		*Extended mnemonic for* **rlwinm. RA,RS,b+n,32–n,31**	CR[CR0]	
inslwi	RA, RS, n, b	Insert from left immediate. (n > 0) $(RA)_{b:b+n-1} \leftarrow (RS)_{0:n-1}$ *Extended mnemonic for* **rlwimi RA,RS,32–b,b,b+n–1**		299
inslwi.		*Extended mnemonic for* **rlwimi. RA,RS,32–b,b,b+n–1**	CR[CR0]	
insrwi	RA, RS, n, b	Insert from right immediate. (n > 0) $(RA)_{b:b+n-1} \leftarrow (RS)_{32-n:31}$ *Extended mnemonic for* **rlwimi RA,RS,32–b–n,b,b+n–1**		299
insrwi.		*Extended mnemonic for* **rlwimi. RA,RS,32–b–n,b,b+n–1**	CR[CR0]	
la	RT, D(RA)	Load address. ($RA \neq 0$) D is an offset from a base address that is assumed to be (RA). *(RT) ← (RA) + EXTS(D)* *Extended mnemonic* for **addi RT,RA,D**		164
li	RT, IM	Load immediate. (RT) ← EXTS(IM) *Extended mnemonic for* **addi RT,0,value**		164
lis	RT, IM	Load immediate shifted. $(RT) \leftarrow (IM \parallel {}^{16}0)$ *Extended mnemonic for* **addis RT,0,value**		167

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
mfccr0 **mfctr** **mfdac1** **mfdac2** **mfdear** **mfdbcr0** **mfdbcr1** **mfdbsr** **mfdccr** **mfdcwr** **mfdvc1** **mfdvc2** **mfesr** **mfevpr** **mfiac1** **mfiac2** **mfiac3** **mfiac4** **mficcr** **mficdbdr** **mflr** **mfpid** **mfpit** **mfpvr** **mfsgr** **mfsler** **mfsprg0** **mfsprg1** **mfsprg2** **mfsprg3** **mfsprg4** **mfsprg5** **mfsprg6** **mfsprg7** **mfsrr0** **mfsrr1** **mfsrr2** **mfsrr3** **mfsu0r** **mftcr** **mftsr** **mfxer** **mfzpr**	RT	Move from special purpose register (SPR) SPRN. *Extended mnemonic for* **mfspr RT,SPRN** See *Table 10-3* on page 354 for listing of valid SPRN values.		266
mftb	RT	Move the contents of TBL into RT, (RT) ← (TBL) *Extended mnemonic for* **mftb RT,TBL**		268
mftbu	RT	Move the contents of TBU into RT, (RT) ← (TBU) *Extended mnemonic for* **mftb RT,TBU**		268
mr	RT, RS	Move register. (RT) ← (RS) *Extended mnemonic for* **or RT,RS,RS**		293
mr.		*Extended mnemonic for* **or. RT,RS,RS**	CR[CR0]	
mtcr	RS	Move to Condition Register. *Extended mnemonic for* **mtcrf 0xFF,RS**		269

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
mtccr0 **mtctr** **mtdac1** **mtdac2** **mtdbcr0** **mtdbcr1** **mtdbsr** **mtdccr** **mtdear** **mtdcwr** **mtdvc1** **mtdvc2** **mtesr** **mtevpr** **mtiac1** **mtiac2** **mtiac3** **mtiac4** **mticcr** **mticdbdr** **mtlr** **mtpid** **mtpit** **mtpvr** **mtsgr** **mtsler** **mtsprg0** **mtsprg1** **mtsprg2** **mtsprg3** **mtsprg4** **mtsprg5** **mtsprg6** **mtsprg7** **mtsrr0** **mtsrr1** **mtsrr2** **mtsrr3** **mtsu0r** **mttcr** **mttsr** **mtxer** **mtzpr**	RS	Move to SPR SPRN. *Extended mnemonic* for **mtspr SPRN,RS** See *Table 10-3* on page 354 for listing of valid SPRN values.		272
nop		Preferred no-op; triggers optimizations based on no-ops. *Extended mnemonic for* **ori 0,0,0**		295
not	RA, RS	Complement register. (RA) ← ¬(RS) *Extended mnemonic for* **nor RA,RS,RS**		292
not.		*Extended mnemonic for* **nor. RA,RS,RS**	CR[CR0]	
rotlw	RA, RS, RB	Rotate left. (RA) ← ROTL((RS), $(RB)_{27:31}$) *Extended mnemonic for* **rlwnm RA,RS,RB,0,31**		302
rotlw.		*Extended mnemonic for* **rlwnm. RA,RS,RB,0,31**	CR[CR0]	

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
rotlwi	RA, RS, n	Rotate left immediate. (RA) $\leftarrow$ ROTL((RS), n) *Extended mnemonic for* **rlwinm RA,RS,n,0,31**		300
rotlwi.		*Extended mnemonic for* **rlwinm. RA,RS,n,0,31**	CR[CR0]	
rotrwi	RA, RS, n	Rotate right immediate. (RA) $\leftarrow$ ROTL((RS), 32−n) *Extended mnemonic for* **rlwinm RA,RS,32−n,0,31**		300
rotrwi.		*Extended mnemonic for* **rlwinm. RA,RS,32−n,0,31**	CR[CR0]	
slwi	RA, RS, n	Shift left immediate. (n < 32) $(RA)_{0:31-n} \leftarrow (RS)_{n:31}$ $(RA)_{32-n:31} \leftarrow {}^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,n,0,31−n**		300
slwi.		*Extended mnemonic for* **rlwinm. RA,RS,n,0,31−n**	CR[CR0]	
srwi	RA, RS, n	Shift right immediate. (n < 32) $(RA)_{n:31} \leftarrow (RS)_{0:31-n}$ $(RA)_{0:n-1} \leftarrow {}^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,32−n,n,31**		300
srwi.		*Extended mnemonic for* **rlwinm. RA,RS,32−n,n,31**	CR[CR0]	
sub	RT, RA, RB	Subtract (RB) from (RA). (RT) $\leftarrow$ ¬(RB) + (RA) + 1. *Extended mnemonic for* **subf RT,RB,RA**		327
sub.		*Extended mnemonic for* **subf. RT,RB,RA**	CR[CR0]	
subo		*Extended mnemonic for* **subfo RT,RB,RA**	XER[SO, OV]	
subo.		*Extended mnemonic for* **subfo. RT,RB,RA**	CR[CR0] XER[SO, OV]	
subc	RT, RA, RB	Subtract (RB) from (RA). (RT) $\leftarrow$ ¬(RB) + (RA) + 1. Place carry-out in XER[CA]. *Extended mnemonic for* **subfc RT,RB,RA**		328
subc.		*Extended mnemonic for* **subfc. RT,RB,RA**	CR[CR0]	
subco		*Extended mnemonic for* **subfco RT,RB,RA**	XER[SO, OV]	
subco.		*Extended mnemonic for* **subfco. RT,RB,RA**	CR[CR0] XER[SO, OV]	
subi	RT, RA, IM	Subtract EXTS(IM) from (RA|0). Place result in RT. *Extended mnemonic for* **addi RT,RA,−IM**		164

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
subic	RT, RA, IM	Subtract EXTS(IM) from (RA). Place result in RT. Place carry-out in XER[CA]. *Extended mnemonic for* **addic RT,RA,–IM**		165
subic.	RT, RA, IM	Subtract EXTS(IM) from (RA). Place result in RT. Place carry-out in XER[CA]. *Extended mnemonic for* **addic. RT,RA,–IM**	CR[CR0]	166
subis	RT, RA, IM	Subtract (IM $\|\|$ $^{16}0$) from (RA|0). Place result in RT. *Extended mnemonic for* **addis RT,RA,–IM**		167

Table B-5. Extended Mnemonics for PPC405 (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
tweqi	RA, IM	Trap if (RA) equal to EXTS(IM). *Extended mnemonic for* **twi 4,RA,IM**		344
twgei		Trap if (RA) greater than or equal to EXTS(IM). *Extended mnemonic for* **twi 12,RA,IM**		
twgti		Trap if (RA) greater than EXTS(IM). *Extended mnemonic for* **twi 8,RA,IM**		
twlei		Trap if (RA) less than or equal to EXTS(IM). *Extended mnemonic for* **twi 20,RA,IM**		
twlgei		Trap if (RA) logically greater than or equal to EXTS(IM). *Extended mnemonic for* **twi 5,RA,IM**		
twlgti		Trap if (RA) logically greater than EXTS(IM). *Extended mnemonic for* **twi 1,RA,IM**		
twllei		Trap if (RA) logically less than or equal to EXTS(IM). *Extended mnemonic for* **twi 6,RA,IM**		
twllti		Trap if (RA) logically less than EXTS(IM). *Extended mnemonic for* **twi 2,RA,IM**		
twlngi		Trap if (RA) logically not greater than EXTS(IM). *Extended mnemonic for* **twi 6,RA,IM**		
twlnli		Trap if (RA) logically not less than EXTS(IM). *Extended mnemonic for* **twi 5,RA,IM**		
twlti		Trap if (RA) less than EXTS(IM). *Extended mnemonic for* **twi 16,RA,IM**		
twnei		Trap if (RA) not equal to EXTS(IM). *Extended mnemonic for* **twi 24,RA,IM**		
twngi		Trap if (RA) not greater than EXTS(IM). *Extended mnemonic for* **twi 20,RA,IM**		
twnli		Trap if (RA) not less than EXTS(IM). *Extended mnemonic for* **twi 12,RA,IM**		

B.5 Storage Reference Instructions

The PPC405 uses load and store instructions to transfer data between memory and the general purpose registers. Load and store instructions operate on byte, halfword and word data. The storage reference instructions also support loading or storing multiple registers, character strings, and byte-reversed data. Table B-6 shows the storage reference instructions available for use in the PPC405.

Table B-6. Storage Reference Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
lbz	RT, D(RA)	Load byte from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) $\leftarrow$ $^{24}0$ || MS(EA,1).		225
lbzu	RT, D(RA)	Load byte from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) $\leftarrow$ $^{24}0$ || MS(EA,1). Update the base address, (RA) $\leftarrow$ EA.		226
lbzux	RT, RA, RB	Load byte from EA = (RA|0) + (RB) and pad left with zeroes, (RT) $\leftarrow$ $^{24}0$ || MS(EA,1). Update the base address, (RA) $\leftarrow$ EA.		227
lbzx	RT, RA, RB	Load byte from EA = (RA|0) + (RB) and pad left with zeroes, (RT) $\leftarrow$ $^{24}0$ || MS(EA,1).		228
lha	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and sign extend, (RT) $\leftarrow$ EXTS(MS(EA,2)).		229
lhau	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and sign extend, (RT) $\leftarrow$ EXTS(MS(EA,2)). Update the base address, (RA) $\leftarrow$ EA.		230
lhaux	RT, RA, RB	Load halfword from EA = (RA|0) + (RB) and sign extend, (RT) $\leftarrow$ EXTS(MS(EA,2)). Update the base address, (RA) $\leftarrow$ EA.		231
lhax	RT, RA, RB	Load halfword from EA = (RA|0) + (RB) and sign extend, (RT) $\leftarrow$ EXTS(MS(EA,2)).		232
lhbrx	RT, RA, RB	Load halfword from EA = (RA|0) + (RB), then reverse byte order and pad left with zeroes, (RT) $\leftarrow$ $^{16}0$ || MS(EA+1,1) || MS(EA,1).		233
lhz	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) $\leftarrow$ $^{16}0$ || MS(EA,2).		234
lhzu	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) $\leftarrow$ $^{16}0$ || MS(EA,2). Update the base address, (RA) $\leftarrow$ EA.		235
lhzux	RT, RA, RB	Load halfword from EA = (RA|0) + (RB) and pad left with zeroes, (RT) $\leftarrow$ $^{16}0$ || MS(EA,2). Update the base address, (RA) $\leftarrow$ EA.		236
lhzx	RT, RA, RB	Load halfword from EA = (RA|0) + (RB) and pad left with zeroes, (RT) $\leftarrow$ $^{16}0$ || MS(EA,2).		237
lmw	RT, D(RA)	Load multiple words starting from EA = (RA|0) + EXTS(D). Place into consecutive registers, RT through GPR(31). RA is not altered unless RA = GPR(31).		238
lswi	RT, RA, NB	Load consecutive bytes from EA = (RA|0). Number of bytes n = 32 if NB = 0, else n = NB. Stack bytes into words in CEIL(n/4) consecutive registers starting with RT, to R_{FINAL} $\leftarrow$ ((RT + CEIL(n/4) – 1) % 32). GPR(0) is consecutive to GPR(31). RA is not altered unless RA = R_{FINAL}.		239

Table B-6. Storage Reference Instructions (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
lswx	RT, RA, RB	Load consecutive bytes from EA=(RA|0)+(RB). Number of bytes n = XER[TBC]. Stack bytes into words in CEIL(n/4) consecutive registers starting with RT, to $R_{FINAL} \leftarrow$ ((RT + CEIL(n/4) – 1) % 32). GPR(0) is consecutive to GPR(31). RA is not altered unless RA = R_{FINAL}. RB is not altered unless RB = R_{FINAL}. If n=0, content of RT is undefined.		241
lwarx	RT, RA, RB	Load word from EA = (RA|0) + (RB)and place in RT, (RT) $\leftarrow$ MS(EA,4). Set the Reservation bit.		243
lwbrx	RT, RA, RB	Load word from EA = (RA|0) + (RB) then reverse byte order, (RT) $\leftarrow$ MS(EA+3,1) || MS(EA+2,1) || MS(EA+1,1) || MS(EA,1).		244
lwz	RT, D(RA)	Load word from EA = (RA|0) + EXTS(D) and place in RT, (RT) $\leftarrow$ MS(EA,4).		245
lwzu	RT, D(RA)	Load word from EA = (RA|0) + EXTS(D) and place in RT, (RT) $\leftarrow$ MS(EA,4). Update the base address, (RA) $\leftarrow$ EA.		246
lwzux	RT, RA, RB	Load word from EA = (RA|0) + (RB) and place in RT, (RT) $\leftarrow$ MS(EA,4). Update the base address, (RA) $\leftarrow$ EA.		247
lwzx	RT, RA, RB	Load word from EA = (RA|0) + (RB) and place in RT, (RT) $\leftarrow$ MS(EA,4).		248
stb	RS, D(RA)	Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + EXTS(D).		308
stbu	RS, D(RA)	Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + EXTS(D). Update the base address, (RA) $\leftarrow$ EA.		309
stbux	RS, RA, RB	Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + (RB). Update the base address, (RA) $\leftarrow$ EA.		310
stbx	RS, RA, RB	Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + (RB).		311
sth	RS, D(RA)	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + EXTS(D).		312
sthbrx	RS, RA, RB	Store halfword $(RS)_{16:31}$ byte-reversed in memory at EA = (RA|0) + (RB). MS(EA, 2) $\leftarrow$ $(RS)_{24:31}$ || $(RS)_{16:23}$		313
sthu	RS, D(RA)	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + EXTS(D). Update the base address, (RA) $\leftarrow$ EA.		314
sthux	RS, RA, RB	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + (RB). Update the base address, (RA) $\leftarrow$ EA.		315
sthx	RS, RA, RB	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + (RB).		316
stmw	RS, D(RA)	Store consecutive words from RS through GPR(31) in memory starting at EA = (RA|0) + EXTS(D).		317

Table B-6. Storage Reference Instructions (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
stswi	RS, RA, NB	Store consecutive bytes in memory starting at EA=(RA|0). Number of bytes n = 32 if NB = 0, else n = NB. Bytes are unstacked from CEIL(n/4) consecutive registers starting with RS. GPR(0) is consecutive to GPR(31).		318
stswx	RS, RA, RB	Store consecutive bytes in memory starting at EA=(RA|0)+(RB). Number of bytes n = XER[TBC]. Bytes are unstacked from CEIL(n/4) consecutive registers starting with RS. GPR(0) is consecutive to GPR(31).		319
stw	RS, D(RA)	Store word (RS) in memory at EA = (RA|0) + EXTS(D).		321
stwbrx	RS, RA, RB	Store word (RS) byte-reversed in memory at EA = (RA|0) + (RB). MS(EA, 4) $\leftarrow$ $(RS)_{24:31} \parallel (RS)_{16:23} \parallel$ $(RS)_{8:15} \parallel (RS)_{0:7}$		322

stwcx.	RS, RA, RB	Store word (RS) in memory at EA = (RA|0) + (RB) only if the reservation bit is set.
		if RESERVE = 1 then
		MS(EA, 4) $\leftarrow$ (RS)
		RESERVE $\leftarrow$ 0
		(CR[CR0]) $\leftarrow$ $^{2}0 \parallel 1 \parallel XER_{so}$
		else
		(CR[CR0]) $\leftarrow$ $^{2}0 \parallel 0 \parallel XER_{so}$.
				323

stwu	RS, D(RA)	Store word (RS) in memory at EA = (RA|0) + EXTS(D). Update the base address, (RA) $\leftarrow$ EA.		324
stwux	RS, RA, RB	Store word (RS) in memory at EA = (RA|0) + (RB). Update the base address, (RA) $\leftarrow$ EA.		325
stwx	RS, RA, RB	Store word (RS) in memory at EA = (RA|0) + (RB).		326

B.6 Arithmetic and Logical Instructions

Table B-7 lists the arithmetic and logical instructions. Arithmetic operations are performed on integer or ordinal operands stored in registers. Instructions using two operands are defined in a three-operand format, where the operation is performed on the operands stored in two registers, and the result is placed in a third register. Instructions using one operand are defined in a two-operand format, where the operation is performed on the operand in one register, and the result is placed in another register. Several instructions have immediate formats, in which one operand is coded as part of the instruction itself. Most arithmetic and logical instructions can optionally set the Condition Register (CR) based on the outcome of the instruction.

Table B-7. Arithmetic and Logical Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
add	RT, RA, RB	Add (RA) to (RB). Place result in RT.		
add.			CR[CR0]	
addo			XER[SO, OV]	
addo.			CR[CR0] XER[SO, OV]	

Table B-7. Arithmetic and Logical Instructions (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
addc	RT, RA, RB	Add (RA) to (RB). Place result in RT. Place carry-out in XER[CA].		161
addc.			CR[CR0]	
addco			XER[SO, OV]	
addco.			CR[CR0] XER[SO, OV]	
adde	RT, RA, RB	Add XER[CA], (RA), (RB). Place result in RT. Place carry-out in XER[CA].		162
adde.			CR[CR0]	
addeo			XER[SO, OV]	
addeo.			CR[CR0] XER[SO, OV]	
addi	RT, RA, IM	Add EXTS(IM) to (RA|0). Place result in RT.		163
addic	RT, RA, IM	Add EXTS(IM) to (RA|0). Place result in RT. Place carry-out in XER[CA].		
addic.	RT, RA, IM	Add EXTS(IM) to (RA|0). Place result in RT. Place carry-out in XER[CA].	CR[CR0]	
addis	RT, RA, IM	Add (IM $\|\|$ $^{16}0$) to (RA|0). Place result in RT.		
addme	RT, RA	Add XER[CA], (RA), (-1). Place result in RT. Place carry-out in XER[CA].		164
addme.			CR[CR0]	166
addmeo			XER[SO, OV]	166
addmeo.			CR[CR0] XER[SO, OV]	167
addze	RT, RA	Add XER[CA] to (RA). Place result in RT. Place carry-out in XER[CA].		168
addze.			CR[CR0]	
addzeo			XER[SO, OV]	
addzeo.			CR[CR0] XER[SO, OV]	
and	RA, RS, RB	AND (RS) with (RB). Place result in RA.		170
and.			CR[CR0]	
andc	RA, RS, RB	AND (RS) with ¬(RB). Place result in RA.		171
andc.			CR[CR0]	
andi.	RA, RS, IM	AND (RS) with ($^{16}0$ $\|\|$ IM). Place result in RA.	CR[CR0]	172
andis.	RA, RS, IM	AND (RS) with (IM $\|\|$ $^{16}0$). Place result in RA.	CR[CR0]	173
cntlzw	RA, RS	Count leading zeros in RS. Place result in RA.		192
cntlzw.			CR[CR0]	

Table B-7. Arithmetic and Logical Instructions (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
divw	RT, RA, RB	Divide (RA) by (RB), signed. Place result in RT.		213
divw.			CR[CR0]	
divwo			XER[SO, OV]	
divwo.			CR[CR0] XER[SO, OV]	
divwu	RT, RA, RB	Divide (RA) by (RB), unsigned. Place result in RT.		214
divwu.			CR[CR0]	
divwuo			XER[SO, OV]	
divwuo.			CR[CR0] XER[SO, OV]	
eqv	RA, RS, RB	Equivalence of (RS) with $\overline{(RB)}$. $(RA) \leftarrow \neg((RS) \oplus (RB))$		216
eqv.			CR[CR0]	
extsb	RA, RS	Extend the sign of byte $(RS)_{24:31}$. Place the result in RA.		217
extsb.			CR[CR0]	
extsh	RA, RS	Extend the sign of halfword $(RS)_{16:31}$. Place the result in RA.		218
extsh.			CR[CR0]	
mulhw	RT, RA, RB	Multiply (RA) and (RB), signed. Place hi-order result in RT. $prod_{0:63} \leftarrow (RA) \times (RB)$ (signed). $(RT) \leftarrow prod_{0:31.}$		280
mulhw.			CR[CR0]	
mulhwu	RT, RA, RB	Multiply (RA) and (RB), unsigned. Place hi-order result in RT. $prod_{0:63} \leftarrow (RA) \times (RB)$ (unsigned). $(RT) \leftarrow prod_{0:31.}$		281
mulhwu.			CR[CR0]	
mulli	RT, RA, IM	Multiply (RA) and IM, signed. Place lo-order result in RT. $prod_{0:47} \leftarrow (RA) \times IM$ (signed) $(RT) \leftarrow prod_{16:47}$		282
mullw	RT, RA, RB	Multiply (RA) and (RB), signed. Place lo-order result in RT. $prod_{0:63} \leftarrow (RA) \times (RB)$ (signed). $(RT) \leftarrow prod_{32:63.}$		283
mullw.			CR[CR0]	
mullwo			XER[SO, OV]	
mullwo.			CR[CR0] XER[SO, OV]	
nand	RA, RS, RB	NAND (RS) with (RB). Place result in RA.		284
nand.			CR[CR0]	
neg	RT, RA	Negative (two’s complement) of RA. $(RT) \leftarrow \neg(RA) + 1$		285
neg.			CR[CR0]	
nego			XER[SO, OV]	
nego.			CR[CR0] XER[SO, OV]	
nor	RA, RS, RB	NOR (RS) with (RB). Place result in RA.		292
nor.			CR[CR0]	
or	RA, RS, RB	OR (RS) with (RB). Place result in RA.		293
or.			CR[CR0]	

Table B-7. Arithmetic and Logical Instructions (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
orc	RA, RS, RB	OR (RS) with ¬(RB). Place result in RA.		294
orc.			CR[CR0]	
ori	RA, RS, IM	OR (RS) with ($^{16}0 \parallel$ IM). Place result in RA.		295
oris	RA, RS, IM	OR (RS) with (IM $\parallel {}^{16}0$). Place result in RA.		296
subf	RT, RA, RB	Subtract (RA) from (RB). (RT) ← ¬(RA) + (RB) + 1.		327
subf.			CR[CR0]	
subfo			XER[SO, OV]	
subfo.			CR[CR0] XER[SO, OV]	
subfc	RT, RA, RB	Subtract (RA) from (RB). (RT) ← ¬(RA) + (RB) + 1. Place carry-out in XER[CA].		328
subfc.			CR[CR0]	
subfco			XER[SO, OV]	
subfco.			CR[CR0] XER[SO, OV]	
subfe	RT, RA, RB	Subtract (RA) from (RB) with carry-in. (RT) ← ¬(RA) + (RB) + XER[CA]. Place carry-out in XER[CA].		329
subfe.			CR[CR0]	
subfeo			XER[SO, OV]	
subfeo.			CR[CR0] XER[SO, OV]	
subfic	RT, RA, IM	Subtract (RA) from EXTS(IM). (RT) ← ¬(RA) + EXTS(IM) + 1. Place carry-out in XER[CA].		330
subfme	RT, RA, RB	Subtract (RA) from (–1) with carry-in. (RT) ← ¬(RA) + (–1) + XER[CA]. Place carry-out in XER[CA].		331
subfme.			CR[CR0]	
subfmeo			XER[SO, OV]	
subfmeo.			CR[CR0] XER[SO, OV]	
subfze	RT, RA, RB	Subtract (RA) from zero with carry-in. (RT) ← ¬(RA) + XER[CA]. Place carry-out in XER[CA].		332
subfze.			CR[CR0]	
subfzeo			XER[SO, OV]	
subfzeo.			CR[CR0] XER[SO, OV]	
xor	RA, RS, RB	XOR (RS) with (RB). Place result in RA.		349
xor.			CR[CR0]	
xori	RA, RS, IM	XOR (RS) with ($^{16}0 \parallel$ IM). Place result in RA.		350
xoris	RA, RS, IM	XOR (RS) with (IM $\parallel {}^{16}0$). Place result in RA.		351

B.7 Condition Register Logical Instructions

CR logical instructions combine the results of several comparisons without incurring the overhead of conditional branching. These instructions can significantly improve code performance if multiple conditions are tested before making a branch decision. Table B-8 summarizes the CR logical instructions.

Table B-8. Condition Register Logical Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
crand	BT, BA, BB	AND bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		193
crandc	BT, BA, BB	AND bit (CR_{BA}) with $\neg$(CR_{BB}). Place result in CR_{BT}.		194
creqv	BT, BA, BB	Equivalence of bit CR_{BA} with CR_{BB}. $CR_{BT} \leftarrow \neg(CR_{BA} \oplus CR_{BB})$		195
crnand	BT, BA, BB	NAND bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		196
crnor	BT, BA, BB	NOR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		197
cror	BT, BA, BB	OR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		198
crorc	BT, BA, BB	OR bit (CR_{BA}) with $\neg$ (CR_{BB}). Place result in CR_{BT}.		199
crxor	BT, BA, BB	XOR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		200
mcrf	BF, BFA	Move CR field, (CR[CRn]) $\leftarrow$ (CR[CRm]) where m $\leftarrow$ BFA and n $\leftarrow$ BF.		263

B.8 Branch Instructions

The architecture provides conditional and unconditional branches to any storage location. The conditional branch instructions test condition codes set previously and branch accordingly. Conditional branch instructions may decrement and test the Count Register (CTR) as part of determination of the branch condition and may save the return address in the Link Register (LR). The target address for a branch may be a displacement from the current instruction address (CIA), or may be contained in the LR or CTR, or may be an absolute address.

Table B-9. Branch Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
b	target	Branch unconditional relative. $LI \leftarrow (target - CIA)_{6:29}$ $NIA \leftarrow CIA + EXTS(LI \parallel {}^{2}0)$		174
ba		Branch unconditional absolute. $LI \leftarrow target_{6:29}$ $NIA \leftarrow EXTS(LI \parallel {}^{2}0)$		
bl		Branch unconditional relative. $LI \leftarrow (target - CIA)_{6:29}$ $NIA \leftarrow CIA + EXTS(LI \parallel {}^{2}0)$	$(LR) \leftarrow CIA + 4.$	
bla		Branch unconditional absolute. $LI \leftarrow target_{6:29}$ $NIA \leftarrow EXTS(LI \parallel {}^{2}0)$	$(LR) \leftarrow CIA + 4.$	
bc	BO, BI, target	Branch conditional relative. $BD \leftarrow (target - CIA)_{16:29}$ $NIA \leftarrow CIA + EXTS(BD \parallel {}^{2}0)$	CTR if $BO_2 = 0$.	175
bca		Branch conditional absolute. $BD \leftarrow target_{16:29}$ $NIA \leftarrow EXTS(BD \parallel {}^{2}0)$	CTR if $BO_2 = 0$.	
bcl		Branch conditional relative. $BD \leftarrow (target - CIA)_{16:29}$ $NIA \leftarrow CIA + EXTS(BD \parallel {}^{2}0)$	CTR if $BO_2 = 0$. $(LR) \leftarrow CIA + 4.$	
bcla		Branch conditional absolute. $BD \leftarrow target_{16:29}$ $NIA \leftarrow EXTS(BD \parallel {}^{2}0)$	CTR if $BO_2 = 0$. $(LR) \leftarrow CIA + 4.$	
bcctr	BO, BI	Branch conditional to address in CTR. Using (CTR) at exit from instruction, $NIA \leftarrow CTR_{0:29} \parallel {}^{2}0.$	CTR if $BO_2 = 0$.	181
bcctrl			CTR if $BO_2 = 0$. $(LR) \leftarrow CIA + 4.$	
bclr	BO, BI	Branch conditional to address in LR. Using (LR) at entry to instruction, $NIA \leftarrow LR_{0:29} \parallel {}^{2}0.$	CTR if $BO_2 = 0$.	184
bclrl			CTR if $BO_2 = 0$. $(LR) \leftarrow CIA + 4.$	

B.9 Comparison Instructions

Comparison instructions perform arithmetic and logical comparisons between two operands and set one of the eight condition code register fields based on the outcome of the comparison. Table B-10 shows the comparison instructions supported by the PPC405.

Table B-10. Comparison Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
cmp	BF, 0, RA, RB	Compare (RA) to (RB), signed. Results in CR[CRn], where $n = BF$.		188
cmpi	BF, 0, RA, IM	Compare (RA) to EXTS(IM), signed. Results in CR[CRn], where $n = BF$.		189
cmpl	BF, 0, RA, RB	Compare (RA) to (RB), unsigned. Results in CR[CRn], where $n = BF$.		190

Table B-10. Comparison Instructions (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
cmpli	BF, 0, RA, IM	Compare (RA) to ($^{16}0$ || IM), unsigned. Results in CR[CRn], where n = BF.		191

B.10 Rotate and Shift Instructions

Rotate and shift instructions rotate and shift operands which are stored in the general purpose registers. Rotate instructions can also mask rotated operands. Table B-11 shows the PPC405 rotate and shift instructions.

Table B-11. Rotate and Shift Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
rlwimi	RA, RS, SH, MB, ME	Rotate left word immediate, then insert according to mask. r $\leftarrow$ ROTL((RS), SH) m $\leftarrow$ MASK(MB, ME) (RA) $\leftarrow$ (r $\wedge$ m) $\vee$ ((RA) $\wedge$ $\neg$m)		299
rlwimi.			CR[CR0]	
rlwinm	RA, RS, SH, MB, ME	Rotate left word immediate, then AND with mask. r $\leftarrow$ ROTL((RS), SH) m $\leftarrow$ MASK(MB, ME) (RA) $\leftarrow$ (r $\wedge$ m)		300
rlwinm.			CR[CR0]	
rlwnm	RA, RS, RB, MB, ME	Rotate left word, then AND with mask. r $\leftarrow$ ROTL((RS), $(RB)_{27:31}$) m $\leftarrow$ MASK(MB, ME) (RA) $\leftarrow$ (r $\wedge$ m)		302
rlwnm.			CR[CR0]	
slw	RA, RS, RB	Shift left (RS) by $(RB)_{27:31}$. n $\leftarrow$ $(RB)_{27:31}$. r $\leftarrow$ ROTL((RS), n). if $(RB)_{26}$ = 0 then m $\leftarrow$ MASK(0, 31 – n) else m $\leftarrow$ $^{32}0$. (RA) $\leftarrow$ r $\wedge$ m.		304
slw.			CR[CR0]	
sraw	RA, RS, RB	Shift right algebraic (RS) by $(RB)_{27:31}$. n $\leftarrow$ $(RB)_{27:31}$. r $\leftarrow$ ROTL((RS), 32 – n). if $(RB)_{26}$ = 0 then m $\leftarrow$ MASK(n, 31) else m $\leftarrow$ $^{32}0$. s $\leftarrow$ $(RS)_0$. (RA) $\leftarrow$ (r $\wedge$ m) $\vee$ (^{32}s $\wedge$ $\neg$m). XER[CA] $\leftarrow$ s $\wedge$ ((r $\wedge$ $\neg$m) $\neq$ 0).		305
sraw.			CR[CR0]	
srawi	RA, RS, SH	Shift right algebraic (RS) by SH. n $\leftarrow$ SH. r $\leftarrow$ ROTL((RS), 32 – n). m $\leftarrow$ MASK(n, 31). s $\leftarrow$ $(RS)_0$. (RA) $\leftarrow$ (r $\wedge$ m) $\vee$ (^{32}s $\wedge$ $\neg$m). XER[CA] $\leftarrow$ s $\wedge$ ((r $\wedge$ $\neg$m)$\neq$0).		306
srawi.			CR[CR0]	
srw	RA, RS, RB	Shift right (RS) by $(RB)_{27:31}$. n $\leftarrow$ $(RB)_{27:31}$. r $\leftarrow$ ROTL((RS), 32 – n). if $(RB)_{26}$ = 0 then m $\leftarrow$ MASK(n, 31) else m $\leftarrow$ $^{32}0$. (RA) $\leftarrow$ r $\wedge$ m.		307
srw.			CR[CR0]	

B.11 Cache Control Instructions

Cache control instructions allow the user to indirectly control the contents of the data and instruction caches. The user may fill, flush, invalidate and zero blocks (16-byte lines) in the data cache. The user may also invalidate congruence classes in both caches and invalidate individual lines in the instruction cache.

Table B-12. Cache Control Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
dcba	RA, RB	Speculatively establish the data cache block which contains the EA (RA|0) + (RB).		201
dcbf	RA, RB	Flush (store, then invalidate) the data cache block which contains the EA (RA|0) + (RB).		203
dcbi	RA, RB	Invalidate the data cache block which contains the EA (RA|0) + (RB).		204
dcbst	RA, RB	Store the data cache block which contains the EA (RA|0) + (RB).		205
dcbt	RA, RB	Load the data cache block which contains the EA (RA|0) + (RB).		206
dcbtst	RA,RB	Load the data cache block which contains the EA (RA|0) + (RB).		207
dcbz	RA, RB	Zero the data cache block which contains the EA (RA|0) + (RB).		208
dccci	RA, RB	Invalidate the data cache congruence class associated with the EA (RA|0) + (RB).		210
dcread	RT, RA, RB	Read either tag or data information from the data cache congruence class associated with the EA (RA|0) + (RB). Place the results in RT.		211
icbi	RA, RB	Invalidate the instruction cache block which contains the EA (RA|0) + (RB).		219
icbt	RA, RB	Load the instruction cache block which contains the EA (RA|0) + (RB).		220
iccci	RA, RB	Invalidate instruction cache.		221
icread	RA, RB	Read either tag or data information from the instruction cache congruence class associated with the EA (RA|0) + (RB). Place the results in ICDBDR.		221

B.12 Interrupt Control Instructions

The interrupt control instructions allow the user to move data between general purpose registers and the machine state register, return from interrupts and enable or disable maskable external interrupts. Table B-13 shows the interrupt control instruction set.

Table B-13. Interrupt Control Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
mfmsr	RT	Move from MSR to RT, (RT) $\leftarrow$ (MSR).		265
mtmsr	RS	Move to MSR from RS, (MSR) $\leftarrow$ (RS).		271
rfci		Return from critical interrupt (PC) $\leftarrow$ (SRR2). (MSR) $\leftarrow$ (SRR3).		297

Table B-13. Interrupt Control Instructions (Continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
rfi		Return from interrupt. (PC) $\leftarrow$ (SRR0). (MSR) $\leftarrow$ (SRR1).		298
wrtee	RS	Write value of RS_{16} to the External Enable bit (MSR[EE]).		347
wrteei	E	Write value of E to the External Enable bit (MSR[EE]).		348

B.13 TLB Management Instructions

The TLB management instructions read and write entries of the TLB array in the MMU, search the TLB array for an entry which will translate a given address, invalidate all TLB entries, and synchronize TLB updates with other processors.

Table B-14. TLB Management Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
tlbia		All of the entries in the TLB are invalidated and become unavailable for translation by clearing the valid (V) bit in the TLBHI portion of each TLB entry. The rest of the fields in the TLB entries are unmodified.		334
tlbre	RT, RA,WS	If WS = 0: Load TLBHI portion of the selected TLB entry into RT. Load the PID register with the contents of the TID field of the selected TLB entry. (RT) $\leftarrow$ TLBHI[(RA)] (PID) $\leftarrow$ $TLB[(RA)]_{TID}$ If WS = 1: Load TLBLO portion of the selected TLB entry into RT. (RT) $\leftarrow$ TLBLO[(RA)]		335
tlbsx	RT,RA,RB	Search the TLB array for a valid entry which translates the EA EA = (RA|0) + (RB). If found, (RT) $\leftarrow$ Index of TLB entry. If not found, (RT) Undefined.		337
tlbsx.		If found, (RT) $\leftarrow$ Index of TLB entry. $CR[CR0]_{EQ}$ $\leftarrow$ 1. If not found, (RT) Undefined. $CR[CR0]_{EQ}$ $\leftarrow$ 1.	$CR[CR0]_{LT,GT,SO}$	
tlbsync		**tlbsync** does not complete until all previous TLB-update instructions executed by this processor have been received and completed by all other processors. For the PPC405, **tlbsync** is a no-op.		338
tlbwe	RS, RA,WS	If WS = 0: Write TLBHI portion of the selected TLB entry from RS. Write the TID field of the selected TLB entry from the PID register. TLBHI[(RA)] $\leftarrow$ (RS) $TLB[(RA)]_{TID}$ $\leftarrow$ $(PID)_{24:31}$ If WS = 1: Write TLBLO portion of the selected TLB entry from RS. TLBLO[(RA)] $\leftarrow$ (RS)		339

B.14 Processor Management Instructions

The processor management instructions move data between GPRs and SPRs and DCRs in the PPC405; these instructions also provide traps, system calls and synchronization controls.

Table B-15. Processor Management Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
eieio		Storage synchronization. All loads and stores that precede the **eieio** instruction complete before any loads and stores that follow the instruction access main storage. Implemented as **sync**, which is more restrictive.		215
isync		Synchronize execution context by flushing the prefetch queue.		224
mcrxr	BF	Move XER[0:3] into field CRn, where n←BF. CR[CRn] ← (XER[SO, OV, CA]). (XER[SO, OV, CA]) ← $^{3}0$.		262
mfcr	RT	Move from CR to RT, (RT) ← (CR).		263
mfdcr	RT, DCRN	Move from DCR to RT, (RT) ← (DCR(DCRN)).		264
mfspr	RT, SPRN	Move from SPR to RT, (RT) ← (SPR(SPRN)).		265
mtcrf	FXM, RS	Move some or all of the contents of RS into CR as specified by FXM field, mask $\leftarrow {}^{4}(FXM_0) \parallel {}^{4}(FXM_1) \parallel ... \parallel {}^{4}(FXM_6) \parallel {}^{4}(FXM_7)$. $(CR) \leftarrow ((RS) \wedge mask) \vee (CR) \wedge \neg mask)$.		269
mtdcr	DCRN, RS	Move to DCR from RS, (DCR(DCRN)) ← (RS).		270
mtspr	SPRN, RS	Move to SPR from RS, (SPR(SPRN)) ← (RS).		271
sc		System call exception is generated. (SRR1) ← (MSR) (SRR0) ← (PC) PC ← $EVPR_{0:15}$ || 0x0C00 (MSR[WE, PR, EE, PE, DR, IR]) ← 0		303
sync		Synchronization. All instructions that precede **sync** complete before any instructions that follow **sync** begin. When **sync** completes, all storage accesses initiated before **sync** will have completed.		333
tw	TO, RA, RB	Trap exception is generated if, comparing (RA) with (RB), any condition specified by TO is true.		341
twi	TO, RA, IM	Trap exception is generated if, comparing (RA) with EXTS(IM), any condition specified by TO is true.		344

Appendix C. Code Optimization and Instruction Timings

The code optimization guidelines in "Code Optimization Guidelines" and the information describing instruction timings in *Instruction Timings* on page 431 can help compiler, system, and application programmers produce high-performance code and determine accurate execution times.

C.1 Code Optimization Guidelines

The following guidelines can help to reduce program execution times.

C.1.1 Condition Register Bits for Boolean Variables

Compilers can use Condition Register (CR) bits to store boolean variables, where 0 and 1 represent False and True values, respectively. This generally improves performance, compared to using General Purpose Registers (GPRs) to store boolean variables. Most common operations on boolean variables can be accomplished using the CR Logical instructions.

C.1.2 CR Logical Instruction for Compound Branches

For example, consider the following pseudocode:

```
if (Var28 || Var29 || Var30 || Var 31) branch to target
```

Var28–Var31 are boolean variables, maintained as bits in the CR[CR7] field ($CR_{28:31}$). The value 1 represents True; 0 represents False.

This could be coded with branches as:

```
bt      28, target
bt      29, target
bt      30, target
bt      31, target
```

Generally faster, functionally equivalent code, using CR Logical instructions, follows:

```
crcr    2, 28, 29
cror    2, 2, 30
cror    2, 2, 31
bt      2, target
```

C.1.3 Cache Usage

Code and data can be organized, based on the size and structure of the instruction and data cache arrays, to minimize cache misses.

In the cache arrays, any two addresses in which $A_{m:26}$ (the index) are the same, but which differ in $A_{0:m-1}$ (the tag), are called congruent. (This describes a two-way set-associative cache.) $A_{27:31}$ define the 32 bytes in a cache line, the smallest object that can be brought into the cache. Only two congruent lines can be in the cache simultaneously; accessing a third congruent line causes the removal from the cache of one of the two lines previously there

Table C-1 illustrates the value of *m* and the index size for the various cache array sizes.

Moving new code and data into the cache arrays occurs at the speed of external memory. Much faster execution is possible when all code and data is available in the cache. Organizing code to uniformly use $A_{m:26}$ minimizes the use of congruent addresses.

C.1.4 CR Dependencies

For CR-setting arithmetic, compare, CR-logical, and logical instructions, and the CR-setting **mcrf**, **mcrxr**, and **mtcrf** instructions, put two instructions between the CR-setting instruction and a Branch instruction that uses a bit in the CR field set by the CR-setting instruction.

C.1.5 Branch Prediction

Use the Y-bit in branch instructions to force proper branch prediction when there is a more likely prediction than the standard prediction. See *Branch Prediction* on page 52 for a more information about branch prediction.

C.1.6 Alignment

For speed, align all accesses on the appropriate operand-size boundary. For example, load/store word operands should be word-aligned, and so on. Hardware does not trap unaligned accesses; instead, two accesses are performed for a load or store of an unaligned operand that crosses a word boundary. Unaligned accesses that do not cross word boundaries are performed in one access.

Align branch targets that are unlikely to be hit by “fall-through” code on cache line boundaries (such as the address of functions such as **strcpy**), to minimize the number of unused instructions in cache line fills.

C.2 Instruction Timings

The following timing descriptions consider only “first order” effects of cache misses in the ICU (instruction-side) and DCU (data-side) arrays.

The timing descriptions *do not* provide complete descriptions of the performance penalty associated with cache misses; the timing descriptions do not consider bus contention between the instruction-side and the data-side, or the time associated with performing line fills or flushes. Unless specifically stated otherwise, the number of cycles apply to systems having zero-wait memory access.

C.2.1 General Rules

Instructions execute in order.

All instructions, assuming cache hits, execute in one cycle, except:

- Divide instructions execute in 35 clock cycles.
- Branches execute in one or three clock cycles, as described in “Branches.”
- MAC and multiply instructions execute in one to five cycles as described in “Multiplies.”
- Aligned load/store instructions that hit in the cache execute in one clock cycle/word. See “Alignment” for information on execution timings for unaligned load/stores.
- In isolation, a data cache control instruction takes two cycles in the processor pipeline. However, subsequent DCU accesses are stalled until a cache control instruction finishes accessing the data cache array.

Note: Note that subsequent DCU accesses do not remain stalled while transfers associated with previous data cache control instructions continue on the PLB.

C.2.2 Branches

Branch instructions are decoded in prefetch buffer 0 (PFB0) and the decode stage of the instruction pipeline. Branch targets, whether the branch is known or predicted taken, can be fetched from the PFB0 and DCD stages. Incorrectly predicted branches can be corrected from the DCD or EXE (execute) stages of the pipeline.

Branches can be known taken or known not taken, or can have address or condition dependencies. Branches having address dependencies are never predicted taken. The directions of conditional branches having no address dependencies are statically predicted.

Conditional branches may depend on the results of an instruction that is changing the CR or the CTR.

Address dependencies can occur when:

- A **bclr** instruction that is known taken, or unresolved, follows (immediately, or separated by only one instruction) a link updating instruction (**mtlr** or a branch and link).
- A **bcctr** instruction that is known taken, or unresolved, follows (immediately, or separated by only one instruction) a counter updating instruction (**mtctr** or a branch that decrements the counter).

Instruction timings for branch instructions follow:

- A branch known not taken (BKNT) executes in one clock cycle. By definition a BKNT does not have address or condition dependencies.
- A branch known taken (BKT) by definition has no condition dependencies, but can have address dependencies.A BKT without address dependencies can execute in one clock cycle if it is first decoded from the PFB0 stage, or in two clock cycles if it is first decoded in the DCD stage. A BKT having address dependencies can execute in two clock cycles if there is one instruction between the branch and the address dependency, or in three clock cycles if there are no instructions between the branch and address dependency.
- A branch predicted not taken (BPNT), which must have condition dependencies, executes in one clock cycle if the prediction is correct. If the prediction is incorrect, the branch can take two or three cycles. If there was one instruction between the branch and the instruction causing the condition dependency, the branch executes in two cycles. If there were no instructions between the branch and the instruction causing the condition dependency, the branch executes in three clock cycles.
- A branch that is correctly predicted taken (BPT), which must have condition dependencies, executes in one clock cycle, if it is first decoded from the PFB0 stage, or two clock cycles if it is first decoded in the DCD stage. If the prediction is incorrect, the branch can take two or three cycles. If there is one instruction between the branch and the instruction causing the condition dependency, the branch executes in two cycles. If there are no instructions between the branch and the instruction causing the condition dependency, the branch executes in three clock cycles.

C.2.3 Multiplies

For multiply instructions having two word operands, hardware internal to the core automatically detects smaller operand sizes (by examining sign bit extension) to reduce the number of cycles necessary to complete the multiplication.

The PPC405 also supports multiply accumulate (MAC) instructions and multiply instructions having halfword operands.

Word and halfword multiply instructions are pipelined in the execution unit and use the same multiplication hardware. Because these instructions are pipelined in the execution stage they have latency and reissue rate cycle numbers. Under conditions to be described, a second multiply or MAC instruction can begin execution before the first multiply or MAC instruction completes. When these conditions are met, the reissue rate cycle numbers should be used; otherwise, the latency cycle numbers should be used. (A MAC or multiply instruction can follow another MAC or a multiply and still meet the conditions that support the use of the reissue rate cycle numbers.

Use reissue rate cycle numbers for multiply or MAC instructions that are followed by another multiply or MAC instruction, and do not have an operand dependency from a previous multiply or MAC instruction. However, one operand dependency is allowed for reissue rate cycle numbers. Internal forwarding logic allows the accumulate value of a first MAC instruction to be used as the accumulate value of a second MAC instruction without affecting the reissue rate.

Use latency cycle numbers for multiply or MAC instructions that are not followed by another multiply or MAC, or that have an operand dependency from a previous multiply or MAC instruction. However, accumulate-only dependencies between adjacent MAC instructions use reissue rate cycle numbers.

An operand dependency exists when a second multiply or MAC instruction depends on the result of a first multiply or MAC instruction.

Table C-1 summarizes the multiply and MAC instruction timings. In the table, the syntax “[**o**]” indicates that the instruction has an “o” form that updates XER[SO,OV], and a “non-o” form. The syntax “[**.**]” indicates that the instruction has a “record” form that updates CR[CR0], and a “non-record” form.

Table C-1. Multiply and MAC Instruction Timing

Operation	Reissue Rate Cycles	Latency Cycles
MAC		
MAC and negative MAC instructions	1	2
Halfword x Halfword		
mullhw[**.**], **mullhwu**[**.**], **mulhhw**[**.**], **mulhhwu**[**.**], **mulchw**[**.**], **mulchwu**[**.**]	1	2
mulli[**.**], **mullw**[**o**][**.**], **mulhw**[**.**], **mulhwu**[**.**]	2	3
Halfword x Word		
mulli[**.**], **mullw**[**o**][**.**], **mulhw**[**.**], **mulhwu**[**.**]	2	3
Word × Word		
mullw[**o**][**.**], **mulhw**[**.**], **mulhwu**[**.**]	4	5

C.2.4 Scalar Load Instructions

Generally, the PPC405 executes cacheable load instructions that hit in the data cache array or line fill buffer, or non cacheable load instructions that hit in the line fill buffer (when enabled), in one cycle. However, the pipelined nature of load instructions can even cause loads that hit in the cache or line fill buffer to appear to take extra cycles under some conditions.

If a load is followed by an instruction that uses the load target as an operand, a load-use dependency exists. When the load target is returned, it is forwarded to the operand register of the “using” instruction. This forwarding results in an additional cycle of latency to a load immediately followed by a “using” instruction, causing the load to appear to execute in two cycles.

Because the PPC405 can execute instructions that follow load misses if no load-use dependency exists, the load and the “using” instruction should be separated by two “non-using” instructions when possible. If only one instruction can be placed between the load and the “using” instruction, the load appears to execute in two cycles.

C.2.5 Scalar Store Instructions

Cacheable stores that miss in the DCU, and non cacheable stores, are queued in the data cache so that the store appears to execute in a single cycle if operand-aligned. Under certain conditions, the DCU can pipeline up to three store instructions. (See *Cache Operations* on page 69 for more information.) **stwcx.** instructions that do not cause alignment errors execute in two cycles.

C.2.6 Alignment in Scalar Load and Store Instructions

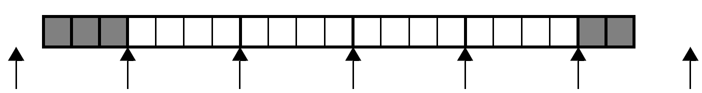
The PPC405 requires an extra cycle to execute scalar loads and stores having unaligned big or little endian data (except for **lwarx** and **stwcx.**, which require word-aligned operands). If the target data is not operand aligned, and the sum of the least two significant bits of the effective address (EA) and the byte count is greater than four, the PPC405 decomposes a load or store scalar into two load or store operations. That is, the PPC405 never presents the DCU with a request for a transfer that crosses a word boundary. For example, a **lwz** with an EA of 0b11 causes the PPC405 to decompose the **lwz** into two load operations. The first load operation is for a byte at the starting effective address; the second load operation is for three bytes, starting at the next word address.

C.2.7 String and Multiple Instructions

Calculating execution times for string and multiple instructions (**lmw** and **stmw**) instructions requires an understanding of data alignment, and of the behavior of the string instructions with respect to alignment.

In the following example, the string contains 21 bytes. The first three bytes do not begin on a word boundary, and the final two bytes do not end on a word boundary. The PPC405 handles any unaligned leading bytes as a special case, then moves as many bytes as aligned words as possible, and finally handles any unaligned trailing bytes as a special case.

In the following example, arrows indicate word boundaries (the address is an exact multiple of four); shaded boxes represent unaligned bytes.



The execution time of the string instruction is the sum of the:

1. Cycles required to handle unaligned leading bytes; if any, add one clock cycle.

   In the example, there are unaligned leading bytes; this transfer adds one clock cycle.

2. Cycles required to handle the number of word-aligned transfers required. Assuming data cache hits, each word-aligned transfer requires one clock cycle.

   In the example, there are four aligned words; this transfer requires four clock cycles.

3. Cycles required to handle unaligned trailing bytes; if any, add one clock cycle.

   In the example, there are unaligned trailing bytes; this transfer adds one clock cycle.

A string instruction operating on the example 21-byte string requires six clock cycles.

C.2.8 Loads and Store Misses

Cacheable stores that miss in the DCU, and non cacheable stores, are queued internally in the DCU so that the store instruction appears to execute in one cycle. Under certain conditions, the DCU can pipeline up to three store instructions. (See the *Cache Operations* on page 69 for more information.)

Because the PPC405 can execute instructions that follow load misses if no load-use dependency exists, the load and the "using" instruction should be separated by "non-using" instructions whenever possible. The number of load miss penalty cycles incurred by a load that misses in the DCU or DCU line fill buffer is reduced by one cycle for every non-use instruction following the load. When the number of non-use instructions following the load is equal to or greater than the number of cycles that it takes to obtain the load data, the load instruction appears to execute in a single cycle. The number of cycles that it takes to obtain load data when it misses in the data cache and line fill buffer depends on whether operand forwarding is enabled or disabled and the system memory timing.

C.2.9 Instruction Cache Misses

Refer to *Instruction Processing* on page 49 for detailed information about the instruction queue and instruction fetching. *Table C-2* illustrates instruction cache penalties for cacheable and non cacheable fetches that miss in the ICU array and line fill buffer.

Table C-2. Instruction Cache Miss Penalties

Type of ICU Request	Miss Penalty Cycles
Sequential	3
Branch Taken from DCD	5
Branch Taken from PFB0	4

Table C-2 assumes that:

- The PPC405 and processor local bus (PLB) run at the same frequency
- The PLB returns an address acknowledge during the first cycle in which the DCU asserts the PLB request
- The target instruction is returned in the cycle following the address acknowledge cycle

The penalty cycles shown for sequential ICU requests assume that the DCD stage and pre-fetch queue are filled with single-cycle non branching instructions or BKNT branch instructions. The penalty cycles for the remaining two rows are for taken branches from DCD and PFB0, respectively.

Index

A

B

C

D

E

F

G

H, I, J, K

L

M

N

O

P

R

S

T

U, V, W

X

Z

Revision Log

Revision Level	Date	Contents of Modification
1.00	Jan. 24, 2007	Initial creation of separate 405 processor UM.
1.01	Feb. 19, 2007	Add bit definitions to CCR0 register for 405EZ chip.
1.02	Sept. 10, 2007	Change clock source for 405EZ to CPU. Correct AMCC phone numbers.