86190

PPC465 PPC465 Processor Complex User's Manual Rev 1.31 Oct 1, 2012

PPC465 PPC465 Processor Complex User's Manual Rev 1.31 Oct 1, 2012 Rev 1

Part Number 465

Revision 1.31 – October 1, 2012

PPC465

PPC465 Processor Complex User’s Manual

Production

PPC465 Processor Complex

User’s Manual

Printed in the United States of America, October 2012.

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

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

http://www.apm.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 AppliedMicro™ PowerPC™ 465 32-bit RISC processor cores. This manual covers both the L1 and L2 cache functions of the PPC465 embedded core.

The PPC465 RISC processor features:

- Book-E Enhanced PowerPC Architecture™
- JTAG support for board level testing
- Extensive development tool support

Who Should Use This Book

This book is for system hardware and software developers, and for application developers who need to understand the PPC465 contained in the AppliedMicro embedded processors.

The following APM86XXX processor(s) contain the PPC465 core.

- APM86290 and APM86190
- APM86392 and APM86391
- APM86791, APM86771, and APM86491
- APM86692 and APM86691

How to Use This Book

This book describes the PPC465 device architecture (including instructions and registers), programming model, interrupts, timer facilities and debugger facilities.

The book is organized as follows:

This book also contains the following appendixes:

To help readers find material in these sections, 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.
$\|\|$	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 $0 \leq r \leq 31$.
(GPR(r))	The contents of GPR r, where $0 \leq r \leq 31$.

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 $\geq$ 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 PPC465 processor core contained in the APM86XXX series processors.

Processor Core	**AppliedMicro Embedded Processors**
PPC465	APM86290 and APM86190
	APM86392 and APM86391
	APM86791, APM86771, and APM86491
	APM86692 and APM86691

This section describes:

- PPC465 Processor Core Features
- PPC465 Processor as a PowerPC Implementation
- PPC465 Organization
- PPC465 Processor Core Interfaces

1.1 PPC465 Processor Core Features

The PPC465 is a Book-E Enhanced Power Architecture processor cores with the following features:

- High performance, dual-issue, superscalar 32-bit RISC CPU
    - Seven stage, highly-pipelined micro-architecture
    - Dual instruction fetch, decode, and out-of-order issue
    - Dynamic branch prediction utilizing a Branch History Table (BHT)
    - Reduced branch latency using Branch Target Address Cache (BTAC)
    - Four independent pipelines
        - Combined complex integer, system, and branch pipeline
        - Simple integer pipeline
        - Load/store pipeline
        - Floating-point unit which is connected in parallel with the other pipelines via the APU interface.
    - Single-cycle multiply
    - Single-cycle multiply-accumulate (new DSP instruction set extensions)
    - 32x32-bit general purpose register (GPR) files
    - Hardware support for all CPU misaligned accesses
    - Full support for both big- and little-endian byte order
    - Extensive power management designed into core for maximum performance/power efficiency
- L1 Cache Features
    - 32KB instruction and data cache arrays
    - Single-cycle access
    - 32-byte (eight word) line size
    - Highly associative (64-way)
    - Write-back and write-through operation
    - Control over whether stores will allocate or write-through on cache miss

- Load/store queues and multiple line fill/flush buffers
- Non-blocking with up to four outstanding load misses
- Cache line locking supported
- Caches can be partitioned to provide separate regions for “transient” instructions and data
- Target word first data access and forwarding
- Cache tags and data are parity protected against soft errors.

1.2 L2 Cache features

- 256K bytes of storage space
- RAM and TAG: ECC protected
- Coherency supported with snooping MESI protocol
- Separate L2 caching-inhibited for data and instruction space support
- WIMG support
- 4-way set associative cache model with LRU replacement
- Cache line locking supported
- 128-byte cache line
- Fixed Address Mode support configurable
- L2 debug features supported
- Array access frequency based Power management support

- Memory management unit
  - Separate instruction and data shadow TLBs
  - 64-entry, fully associative unified TLB array
  - Variable page sizes (1KB–1GB), simultaneously resident in TLB
  - Four-bit extended real address for 36-bit (64GB) addressability
  - Flexible TLB management with software page table search
  - Storage attribute controls for write-through, caching inhibited, guarded, and byte order (endianness)
  - Four user-definable storage attribute controls
  - TLB tags and data are parity protected against soft errors.
- Debug facilities
  - Extensive hardware debug facilities incorporated into the IEEE 1149.1 JTAG port
    - Multiple instruction and data address break points (including range)
    - Data value compare
    - Single-step, branch, trap, and other debug events
  - Non-invasive real-time software trace interface
- Timer facilities
  - 64-bit time base
  - Decrementer with auto-reload capability
  - Fixed interval timer (FIT)

- Watchdog timer with critical interrupt and/or auto-reset
- Multiple core Interfaces
  - PLB interfaces
  - Independent separate Master and Slave 128-bit read and write data interfaces
  - Separate snoop bus with the Master bus
  - Multiple CPU:L2/PLB frequency ratios supported (N:1, N:2)
  - Decoupled address and data bus tenures for high bandwidth performance
  - Tagged out of order data return on-chip memory integration capability over the PLB interface
- Auxiliary Processor Unit (APU) Port for communication with the Floating Point Unit (FPU)
  - Provides functional extensions to the processor pipelines, including GPR file operations
  - 128-bit load/store interface (direct access between FPU and the primary data cache)

1.3 The PPC465 Processor as a PowerPC Implementation

The PPC465 implements the full, 32-bit fixed-point subset of the Book-E Enhanced PowerPC Architecture. Although it fully complies with these architectural specifications, the 64-bit operations of the architecture are not supported. Within the RISC core, the 64-bit operations and the floating-point operations are trapped, and the floating point operations can be emulated using software.

See Appendix A of the Book-E Enhanced PowerPC Architecture specification for more information on 32-bit subset implementations of the architecture.

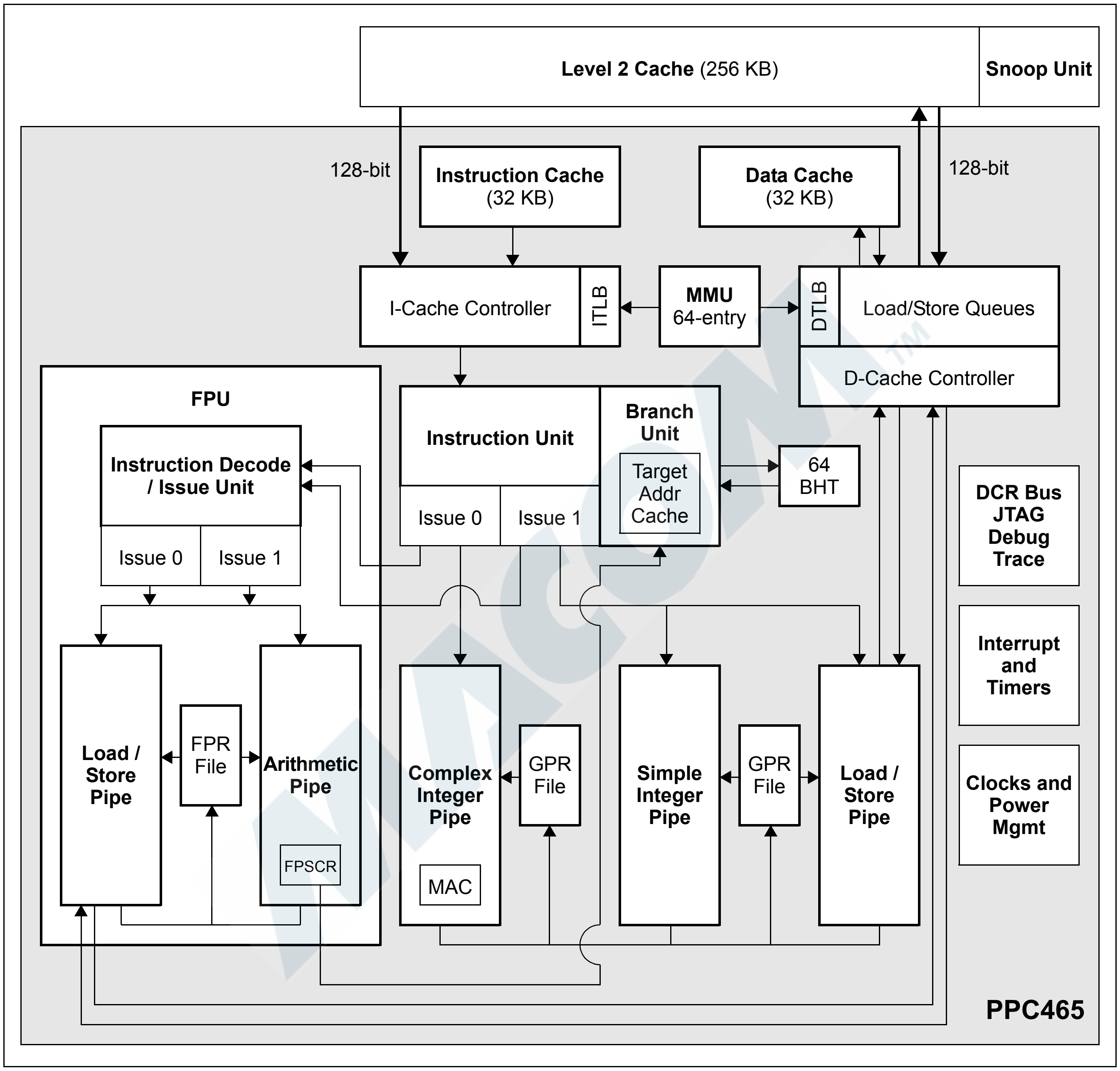
The PPC465 also provides a number of optimizations and extensions to the lower layers of the Book-E Enhanced PowerPC Architecture. Some of the specific implementation features of the PPC465 are simply extensions of the Book-E Enhanced PowerPC Architecture. These features are included to enhance performance, integrate functionality, and reduce system complexity in embedded control applications.

Note: This document differs from the Book-E architecture specification in the use of bit numbering for architected registers. Specifically, Book-E defines the full, 64-bit instruction set architecture, and thus all registers are shown as having bit numbers from 0 to 63, with bit 63 being the least significant. On the other hand, this document describes the PPC465, which is a 32-bit subset implementation of the architecture. Accordingly, all architected registers are described as being 32 bits in length, with the bits numbered from 0 to 31, and with bit 31 being the least significant. Therefore, when this document makes reference to register bit numbers from 0 to 31, they actually correspond to bits 32 to 63 of the same register in the Book-E architecture specification.

1.4 PPC465 Organization

As shown in Figure 1-1, the PPC465 processor includes a memory management unit (MMU); a floating point unit (FPU); separate instruction and data cache units; an interface to a separate Level 2 cache; JTAG, debug, and tracelogic; along with timer facilities.

Figure 1-1. PPC465 Core Block Diagram



1.4.1 Superscalar Instruction Unit

The instruction unit of the PPC465 fetches, decodes, and issues two instructions per cycle to any combination of the three execution pipelines and/or the APU interface. The instruction unit includes a branch unit which provides dynamic branch prediction using a branch history table (BHT), as well as a branch target address cache (BTAC). These mechanisms greatly improve the branch prediction accuracy and reduce the latency of taken branches, such that the target of a branch can usually be executed immediately after the branch itself, with no penalty.

1.4.2 Execution Pipelines

The PPC465 CPU core contains five execution pipelines: complex integer, simple integer, integer load/store, floating point arithmetic, and floating point load/store.

The three integer pipelines consist of four stages and can access the nine-ported (six read, three write) GPR file. In order to improve performance and avoid contention for the GPR file, there are two identical copies of it. One is dedicated to the complex integer pipeline, while the other is shared by the simple integer and the integer load/store pipelines.

- The complex integer pipeline handles all arithmetic, logical, branch, and system management instructions (such as interrupt and TLB management, move to/from system registers, and so on). This pipeline also handles multiply and divide operations, and 24 DSP instructions that perform a variety of multiply-accumulate operations. The complex integer pipeline multiply unit can perform 32-bit × 32-bit multiply operations with single-cycle throughput and three-cycle latency;16-bit × 32-bit multiply operations have only two-cycle latency. Divide operations take 33 cycles.
- The simple integer pipeline can handle most arithmetic and logical operations which do not update the Condition Register (CR).
- The load/store pipeline handles all load, store, and cache management instructions. All misaligned operations are handled in hardware, with no penalty on any operation which is contained within an aligned 16-byte region. The load/store pipeline supports all operations to both big endian and little endian data regions.

The FPU incorporates a dual-issue instruction decode and issue unit and a five-stage arithmetic pipeline working in parallel with a four-stage load/store pipeline. The FPU also contains a Floating Point Register (FPR) file that interfaces to both floating point pipelines. There are thirty-two 64-bit FPRs.

- The two floating point pipelines execute all floating point instructions.

See *Appendix B* on page 653.

1.4.3 Level 1 Instruction and Data Cache Controllers

The PPC465 has separate instruction and data cache controllers and 32KB arrays, which allow concurrent access and minimize pipeline stalls. Both cache controllers have 32-byte lines, and both are highly-associative, with 64-way set-associativity. Both caches support parity checking on the tags and data in the memory arrays, to protect against soft errors. If a parity error is detected, the CPU will cause a machine check exception.

The PowerPC instruction set provides a rich set of cache management instructions for software-enforced coherency. The PPC465 core implementation also provides special debug instructions that can directly read the tag and data arrays.

The PPC465L2AC6 Level-2 cache core provides a unified Level-2 cache, and contains a function that manages transfers between PPC465 CPU core caches (I-side: IU, D-side: DCU) and the Processor Local Bus (PLB). The storage capacity of the L2 cache is 256 K bytes. The L2C provides support for hardware memory coherence in multi-processor systems via the MESI protocol.

The L2 cache controllers connect to the PLB to implement cache coherent symmetric multi-processor systems.

1.4.3.1 Instruction Cache Controller (ICC)

The ICC delivers two instructions per cycle to the instruction unit of the PPC465 CPU core. The ICC also handles the execution of the PowerPC instruction cache management instructions for coherency.

The ICC supports cache line locking, at 16-line granularity. In addition, the notion of a “transient” portion of the cache is supported, in which the cache can be configured such that only a limited portion is used for instruction cache lines from memory pages that are designated by a storage attribute from the MMU as being transient in nature. Such memory pages would contain code which is unlikely to be reused once the processor moves on to the next series of instruction lines, and thus performance may be improved by preventing each series of instruction lines from overwriting all of the “regular” code in the instruction cache.

1.4.3.2 Data Cache Controller (DCC)

The DCC handles all load and store data accesses, as well as the PowerPC data cache management instructions. All misaligned integer accesses are handled in hardware, with those accesses that are contained within a halfline (16 bytes) being handled as a single request. Load and store accesses which cross a 16-byte boundary are broken into two separate accesses by the hardware.

The DCC interfaces to the APU port to provide direct load/store access to the data cache for APU load and store operations. Such APU load and store instructions can access up to 16 bytes (one quadword) in a single cycle.

The data cache can be operated in a store-in (copy-back) or write-through manner, according to the writethrough storage attribute specified for the memory page by the MMU. The DCC also supports both “storewithallocate” and “store-without-allocate” operations, such that store operations that miss in the data cache can either “allocate” the line in the cache by reading it in and storing the new data into the cache, or alternatively bypassing the cache on a miss and simply storing the data to memory. This characteristic can also be specified on a page-by-page basis by a storage attribute in the MMU.

The DCC also supports cache line locking and “transient” data, in the same manner as the ICC.

The DCC provides extensive load, store, and flush queues, such that up to three outstanding line fills and up to four outstanding load misses can be pending, and the DCC can continue servicing subsequent load and store hits in an out-of-order fashion. Store-gathering can also be performed on caching inhibited, writethrough, and “without-allocate” store operations, for up to 16 contiguous bytes. Finally, each cache line has four separate “dirty” bits (one per doubleword), so that the amount of data flushed on cache line replacement can be minimized.

1.4.4 Level 2 Cache (L2C)

The L2C is a unified cache of 256 K bytes, that handles 128-bit reads of CPU instruction cache interface, separate 128-bit read and write data cache interfaces, and separate and independent 128-bit Master and Slave reads and writes interfaces of PLB. The L2C supports cache line locking similar to CPU cache line locking but based on its cache line which is 128-byte, Fixed-Address-Mode for local memory shadowing, and Book-E architecture WIMG. In addition, the L2C supports the Memory coherency for data using snooping MESI protocols.

To improve its reliability and maintain its operability in a system, it is provided with a single bit error correction and double bits error detection ECC for both TAG and data RAM. There are also debug facilities provided to ease trouble shooting of L2C.

1.4.5 Memory Management Unit (MMU)

The PPC465 supports a flat, 36-bit (64 GB) real (physical) address space. This 36-bit real address is generated by the MMU, as part of the translation process from the 32-bit effective address, which is calculated by the processor core as an instruction fetch or load/store address.

The MMU provides address translation, access protection, 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 32-bit effective address space into the 36-bit real address space
- Page level read, write, and execute access control
- Storage attributes for cache policy, byte order (endianness), and speculative memory access
- Software control of page replacement strategy

The translation lookaside buffer (TLB) is the primary hardware resource involved in the control of translation, protection, and storage attributes. It consists of 64 entries, each specifying the various attributes of a given page of the address space. The TLB is fully-associative; the entry for a given page can be placed anywhere in the TLB. The TLB tag and data memory arrays are parity protected against soft errors; if a parity error is detected, the CPU will cause a machine check exception.

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 for managing TLB entries. These instructions are privileged and the processor must be in supervisor state in order for them to be executed.

The first step in the address translation process is to expand the effective address into a virtual address. This is done by taking the 32-bit effective address and appending to it an 8-bit Process ID (PID), as well as a 1-bit “address space” identifier (AS). The PID value is provided by the PID register. The AS identifier is provided by the Machine State Register (MSR, which contains separate bits for the instruction fetch address space (MSR[IS]) and the data access address space (MSR[DS]). Together, the 32-bit effective address, the 8-bit PID, and the 1-bit AS form a 41-bit virtual address. This 41-bit virtual address is then translated into the 36-bit real address using the TLB.

The MMU divides the address space (whether effective, virtual, or real) into pages. Nine page sizes (1 KB, 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 16 MB, 256 MB, and 1 GB) are simultaneously supported, such that at any given time the TLB can contain entries for any combination of page sizes. In order for an address translation to occur, a valid entry for the page containing the virtual address must be in the TLB. An attempt to access an address for which no TLB entry exists causes an Instruction (for fetches) or Data (for load/store accesses) TLB Error exception.

To improve performance, both the instruction cache and the data cache maintain separate “shadow” TLBs. The instruction shadow TLB (ITLB) contains four entries, while the data shadow TLB (DTLB) contains eight. These shadow arrays minimize TLB contention between instruction fetch and data load/store operations. The instruction fetch and data access mechanisms only access the main 64-entry unified TLB when a miss occurs in the respective shadow TLB. The penalty for a miss in either of the shadow TLBs is three cycles. Hardware manages the replacement and invalidation of both the ITLB and DTLB; no system software action is required. Each TLB entry provides separate user state and supervisor state read, write, and execute permission controls for the memory page associated with the entry. If software attempts to access a page for which it does not have the necessary permission, an Instruction (for fetches) or Data (for load/store accesses) Storage exception will occur.

Each TLB entry also provides a collection of storage attributes for the associated page. These attributes control cache policy (such as cachability and write-through as opposed to copy-back behavior), byte order (big endian as opposed to little endian), and enabling of speculative access for the page. In addition, a set of four, user-definable storage attributes are provided. These attributes can be used to control various system level behaviors. They can also be configured to control whether data cache lines are allocated upon a store miss, and whether accesses to a given page should use the “normal” or “transient” portions of the instruction or data cache.

See *Memory Management* on page 219.

1.4.6 Interrupts and Exceptions

An *interrupt* is the action in which the processor saves its old context (Machine State Register (MSR) and next instruction address) and begins execution at a pre-determined interrupt-handler address, with a modified MSR. *Exceptions* are the events that may cause the processor to take an interrupt, if the corresponding interrupt type is enabled.

Exceptions may be generated by the execution of instructions, or by signals from devices external to the PPC465, the internal timer facilities, debug events, or error conditions.

See *Interrupts and Exceptions* on page 247.

1.4.7 Timers

The PPC465 contains a Time Base and three timers: a Decrementer (DEC), a Fixed Interval Timer (FIT), and a Watchdog Timer. The Time Base is a 64-bit counter which is incremented at a frequency either equal to the processor core clock rate or as controlled by a separate, asynchronous timer clock input to the core. No interrupt is generated as a result of the Time Base wrapping back to zero.

The DEC is a 32-bit register that is decremented at the same rate at which the Time Base is incremented. The user loads the DEC register with a value to create the desired interval. When the register is decremented to zero, a number of actions occur: The DEC stops decrementing, a status bit is set in the Timer Status Register (TSR), and a Decrementer exception is reported to the interrupt mechanism of the PPC465. Optionally, the DEC can be programmed to reload automatically the value contained in the Decrementer Auto-Reload register (DECAR), after which the DEC resumes decrementing. The Timer Control Register (TCR) contains the interrupt enable for the Decrementer interrupt.

The FIT generates periodic interrupts based on the transition of a selected bit from the Time Base. Users can select one of four intervals for the FIT period by setting a control field in the TCR to select the appropriate bit from the Time Base. When the selected Time Base bit changes from 0 to 1, a status bit is set in the TSR and a FIT exception is reported to the interrupt mechanism of the PPC465. The FIT interrupt enable is contained in the TCR.

Similar to the FIT, the Watchdog Timer also generates a periodic interrupt based on the transition of a selected bit from the Time Base. Users can select one of four intervals for the watchdog period, again by setting a control field in the TCR to select the appropriate bit from the Time Base. Upon the first change from 0 to 1 of the selected Time Base bit, a status bit is set in the TSR and a Watchdog Timer exception is reported to the interrupt mechanism of the PPC465. The Watchdog Timer can also be configured to initiate a hardware reset if a second transition of the selected Time Base bit occurs prior to the first Watchdog exception being serviced. This capability provides an extra measure of recoverability from potential system lock-ups.

See *Timer Facilities* on page 307.

1.4.8 Debug Facilities

The PPC465 CPU core supports four debug modes: internal, external, real-time-trace, and debug wait. Each mode supports a different type of debug tool used in embedded systems development. Internal debug mode supports software-based ROM monitors, and external debug mode supports a hardware emulator type of debug. Real-time-trace mode uses the debug facilities to indicate events within a trace of processor execution in real time. Debug wait mode enables the processor to continue to service real-time critical interrupts while instruction execution is otherwise stopped for hardware debug. The debug modes are controlled by Debug Control Register 0 (DBCR0) and the setting of bits in the Machine State Register (MSR).

Internal debug mode supports accessing architected processor resources, setting hardware and software breakpoints, and monitoring processor status. In internal debug mode, debug events can generate debug exceptions, 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—running on the processor—along with an external communications path to debug software problems. This mode is used while the processor continues executing instructions and enables debugging of problems in application or operating system code. Access to debugger software executing in the processor while in internal debug mode is through a communications port on the processor board, such as a serial port or ethernet connection.

External debug mode supports stopping, starting, and single-stepping the processor, accessing architected processor resources, setting hardware and software breakpoints, and monitoring processor status. In external debug mode, debug events can architecturally “freeze” the processor. While the processor is frozen, normal instruction execution stops, and the architected processor resources can be accessed and altered using a debug tool attached through the JTAG port. This mode is useful for debugging hardware and low-level control software problems.

L2 cache Debug modes are controlled by L2 Debug Control Register, L2DBCR, and described in L2 Debug Facilities chapter. There are five debug registers for data, L2DBDR0 to 4, three for TAG, L2DBTR) to 2, and one status register, L2DBSR. All L2 debug registers are accessed as DCRs.

See *Debug Facilities* on page 315

1.5 PPC465 Processor Core Interfaces

The PPC465 has the following interfaces for interconnect and debug:

- Processor local bus (PLB)
- Device configuration register (DCR) interface
- Auxiliary processor unit (APU) port (used by optional FPU)
- JTAG, debug, and trace ports
- Interrupt interface
- Clock and power management interface

Several of these interfaces are described briefly in the following sections.

1.5.1 Processor Local Bus (PLB)

There are three independent 128-bit PLB interfaces to the PPC465 core. Each of these interfaces includes a 36-bit address bus and a 128-bit data bus. One PLB interface supports instruction cache reads, while the other two support data cache reads and writes, respectively.

Each of the PLB interfaces supports connection to a PLB subsystem of either 32, 64, or 128 bits. The instruction and data cache controllers handle any dynamic data bus resizing which is required when the subsystem data width is less than the 128 bits of the PPC465 core PLB interfaces.

The data cache PLB interfaces make requests for 32-byte lines, as well as for 1 – 15 bytes within a 16-byte (quadword) aligned region.

The instruction cache controller makes 32-byte line read requests, and also presents quadword burst read requests for up to three 32-byte lines (six quadwords), as part of its speculative line fill mechanism.

Each of the PLB interfaces fully supports the address pipelining capabilities of the PLB, and in fact can go beyond the pipeline depth and minimum latency which the PLB supports. Specifically, each interface supports up to three pipelined request/acknowledge sequences prior to performing the data transfers associated with the first request. For the data cache, if each of the requests must themselves be broken into three separate transactions (for example, for a misaligned doubleword request to a 32-bit PLB slave), then the interface actually supports up to nine outstanding request/acknowledge sequences prior to the first data transfer. Furthermore, each PLB interface tolerates a zero-cycle latency between the request and the address and data acknowledge (that is, the request, address acknowledge, and data acknowledge may all occur in the same cycle).

1.5.2 Device Control Register (DCR) Interface

The DCR interface provides a mechanism for the PPC465 core to setup other on-chip facilities. For example, programmable resources in an external bus interface unit may be configured for usage with various memory devices according to their transfer characteristics and address assignments. DCRs are accessed through the use of the PowerPC mfdcr and mtdcr instructions.

The interface is interlocked with control signals such that it may be connected to peripheral units that are clocked at different frequencies from the processor core.

The DCR interface also allows the PPC465 core to communicate with peripheral devices without using the PLB interface, thereby avoiding the impact to the primary system bus bandwidth, and without additional segmentation of the usable address map.

1.5.3 Auxiliary Processor Unit (APU) Interface

The APU interface provides the PPC465 processor with the flexibility to attach a tightly-coupled Floating-Point Unit (FPU).

APM86XXX Processor(s) with a FPU:

- APM86290 and APM86190
- APM86392 and APM86391
- APM86282 and APM86281
- APM86791 and APM86491

1.5.4 JTAG Port

The PPC465 JTAG port is enhanced to support the attachment of a debug tool. Through the JTAG port, and using the debug facilities designed into the PPC465 core, a debug workstation can single-step the processor and interrogate the internal processor state to facilitate hardware and software debugging. The enhancements comply with the IEEE 1149.1 specification for vendor-specific extensions, and are compatible with standard JTAG hardware for boundaryscan system testing.

2. Processor Complex

The PPC465 Processor Complex includes five direct DCRs for accessing basic configuration information via software. All PPC465s in the PPC465 processor complex have access to these registers. This section documents the definition of these DCRs, and the direct DCR address mapping for the registers. Four read-only registers provide PPC465 processor complex configuration information and one read-write register provides control of some functions in the PPC465 processor complex.

2.1 Configuration and Control Register Summary

Table 2-1. Configuration and Control Register Summary

Mnemonic	Register	Direct DCR #	Access	Privileged	Page
AMCCID	PPC465 complex identifier	0x180	RO	Yes	37
465ID	Processor identifier	0x181	RO	Yes	38
REVID	Revision ID	0x182	RO	Yes	38
CPU_Config	CPU core configuration	0x183	RO	Yes	38
IP_Control	PPC465 Complex Control	0x184	RW	Yes	39

All PPC465s can read the five PPC465 Complex Configuration DCRs at the DCR numbers indicated in the figure above using the mfdcr instruction. All five PPC read/write DCRs (0x184) can be written using the mtdcr instructions.

2.2 Processor Complex Configuration Registers

The PPC465 Processor Complex provides the following read-only Configuration DCRs as a mechanism for software running on any of the PPC465s to sample the static configuration signals applied to the PPC465 processor complex by the SOC at assembly time. The table below shows the format of these four read only DCRs.

2.2.1 Processor Complex Manufacturer ID Register (AMCCID) Direct DCR 0x180

Table 2-2. Processor Complex Manufacturer ID Register (AMCCID) Direct DCR 0x180

Bits	Mnemonic	Description
0:31	AMCCID	“AMCC” identifier text in 8 bit ASCII

2.2.2 Processor Complex ID Register (STREAKID) Direct DCR 0x181

Table 2-3. Processor Complex ID Register (STREAKID) Direct DCR 0x181

Bits	Mnemonic	Description
0:31	STREAKID	"STRK" identifier text in 8 bit ASCII

2.2.3 Processor Complex Revision ID Register (REVID) Direct DCR 0x182 (See PVR register)

Table 2-4. Processor Complex Revision ID Register (REVID) Direct DCR 0x182 (See PVR register)

Bits	Mnemonic	Description
0:25	Rev_ID	Mask
26:28	Major_step	Major revision stepping
29:31	Minor_step	Minor revision stepping

2.2.4 Processor Complex Configuration Register (CPU_CONFIG) Direct DCR 0x183

Table 2-5. Processor Complex Configuration Register (CPU_CONFIG) Direct DCR 0x183

Bits	Mnemonic	Description
0	CPU0_PRESENT	CPU 0 present bit
1	FPU0_PRESENT	FPU 0 present bit.
2:3	L2SIZE_0	L2 cache size. 0 : 0 KB (not present) 1 : 256 KB 2: Reserved 3 : Reserved
4:7	Reserved	Reserved
8	CPU1_PRESENT	CPU 1 present bit.
9	FPU1_PRESENT	FPU 0 present bit.
10:11	L2SIZE_1	L2 cache size. 0 : 0 KB (not present) 1: 256 KB 2: Reserved 3 : Reserved
12:31	Reserved	Reserved

2.3 Processor Complex Control DCR

The PPC465 processor complex provides the following Read/Write Control DCR which contains bits to control the processor complex. These bits address the STWCX store gather issue and the Snooping issue associated with the PPC465 CPU Wait state. This register is located in direct DCR address space at DCR address 0x184.

2.3.1 PPC465 Complex Control Register (IP_CONTROL) Direct DCR 0x184

Table 2-6. PPC465 Complex Control Register (IP_CONTROL) Direct DCR 0x184

Bits	Mnemonic	Description	Notes
0	ENABLE_SNOOP_INHIBIT	Snoop control 0 = allow snooping (default) 1 = inhibit snooping	When set, the assertion of L2CR[SNPME] is blocked if the CPU is in WAIT state.
1	STWCX_GATHER_INHIBIT	STWCX gather control 0 = allow STWCX gather (default) 1 = inhibit STWCX gather	When set, STWCX stores cannot be gathered with other stores. Must always be set to 1.
2:31	Reserved	Reserved	

3. Programming Model

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

- Storage addressing (including data types and byte ordering), starting on page 41
- Registers, starting on page 48
- Instruction classes, starting on page 53
- Instruction set, starting on page 56
- Branch processing, starting on page 63
- Integer processing, starting on page 69
- Processor control, starting on page 72
- User and supervisor state, starting on page 77
- Speculative access, starting on page 78
- Synchronization, starting on page 79

3.1 Storage Addressing

As a 32-bit implementation of the Book-E Enhanced PowerPC Architecture, the PPC465 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 219 for more information on the translation process).

The PPC465 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.

3.1.1 Storage Operands

Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Data storage operands accessed by the integer load/store instructions may be bytes, halfwords, words, or—for load/store multiple and string instructions—a sequence of words or bytes, respectively. The address of a storage operand is the address of its first byte (that is, of its lowest-numbered byte). Byte ordering can be either big endian or little endian, as controlled by the endian storage attribute (see *Byte Ordering* on page 44; also see *Endian (E)* on page 234 for more information on the endian storage attribute).

Operand length is implicit for each scalar storage access instruction type (that is, each storage access instruction type other than the load/store multiple and string instructions). The operand of such a scalar storage access instruction has a “natural” alignment boundary equal to the operand length. In other words, the ‘natural’ address of an operand is an integral multiple of the operand length. A storage operand is said to be *aligned* if it is aligned at its natural boundary: otherwise it is said to be *unaligned*.

Data storage operands for storage access instructions have the following characteristics.

Table 3-1. Data Operand Definitions

Storage Access Instruction Type	Operand Length	Addr[28:31] if aligned
Byte (or String)	8 bits	0bxxxx
Halfword	2 bytes	0bxxx0
Word (or Multiple)	4 bytes	0bxx00
Double word (FPU only)	8 bytes	0bx000
Note: An “x” in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.		

The alignment of the operand effective address of some storage access instructions may affect performance, and in some cases may cause an Alignment exception to occur. For such storage access instructions, the best performance is obtained when the storage operands are aligned. *Table 3-2* summarizes the effects of alignment on those storage access instruction types for which such effects exist. If an instruction type is not shown in the table, then there are no alignment effects for that instruction type.

Table 3-2. Alignment Effects for Storage Access Instructions

Storage Access Instruction Type	Alignment Effects
Integer load/store halfword	Broken into two byte accesses if crosses 16-byte boundary (EA[28:31] = 0b1111); otherwise no effect
Integer load/store word	Broken into two accesses if crosses 16-byte boundary (EA[28:31] > 0b1100); otherwise no effect
Integer load/store multiple or string	Broken into a series of 4-byte accesses until the last byte is accessed or a 16-byte boundary is reached, whichever occurs first. If bytes remain past a 16-byte boundary, resume accessing 4 bytes at a time until the last byte is accessed or the next 16-byte boundary is reached, whichever occurs first; repeat.

Cache management instructions access *cache block* operands, and for the PPC465 the cache block size is 32 bytes. However, the effective addresses calculated by cache management instructions are not required to be aligned on cache block boundaries. Instead, the architecture specifies that the associated low-order effective address bits (bits 27:31 for PPC465) are ignored during the execution of these instructions.

Similarly, the TLB management instructions access *page* operands, and—as determined by the page size—the associated low-order effective address bits are ignored during the execution of these instructions.

Instruction storage operands, on the other hand, are always four bytes long, and the effective addresses calculated by Branch instructions are therefore always word-aligned.

3.1.2 Effective Address Calculation

For a storage access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address of 2^{32}–1 (that is, the storage operand itself crosses the maximum address boundary), the result of the operation is undefined, as specified by the architecture. The PPC465 performs the operation as if the storage operand wrapped around from the maximum effective address to effective address 0. Software, however, should not depend upon this behavior, so that it may be ported to other implementations that do not handle this scenario in the same fashion. Accordingly, software should ensure that no data storage operands cross the maximum address boundary.

Note that since instructions are words and since the effective addresses of instructions are always implicitly on word boundaries, it is not possible for an instruction storage operand to cross any word boundary, including the maximum address boundary.

Effective address arithmetic, which calculates the starting address for storage operands, wraps around from the maximum address to address 0, for all effective address computations except next sequential instruction fetching. See *Instruction Storage Addressing Modes* on page 43 for more information on next sequential instruction fetching at the maximum address boundary.

3.1.2.1 Data Storage Addressing Modes

There are two data storage addressing modes supported by the PPC465:

- Base + displacement (D-mode) addressing mode:

  The 16-bit D field is sign-extended and added to the contents of the GPR designated by RA or to zero if RA = 0; the low-order 32 bits of the sum form the effective address of the data storage operand.

- Base + index (X-mode) addressing mode:

  The contents of the GPR designated by RB (or the value 0 for **lswi** and **stswi**) are added to the contents of the GPR designated by RA, or to 0 if RA = 0; the low-order 32 bits of the sum form the effective address of the data storage operand.

3.1.2.2 Instruction Storage Addressing Modes

There are four instruction storage addressing modes supported by the PPC465:

- I-form branch instructions (unconditional):

  The 24-bit LI field is concatenated on the right with 0b00, sign-extended, and then added to either the address of the branch instruction if AA=0, or to 0 if AA=1; the low-order 32 bits of the sum form the effective address of the next instruction.

- Taken B-form branch instructions:

  The 14-bit BD field is concatenated on the right with 0b00, sign-extended, and then added to either the address of the branch instruction if AA=0, or to 0 if AA=1; the low-order 32 bits of the sum form the effective address of the next instruction.

- Taken XL-form branch instructions:

  The contents of bits 0:29 of the Link Register (LR) or the Count Register (CTR) are concatenated on the right with 0b00 to form the 32-bit effective address of the next instruction.

- Next sequential instruction fetching (including non-taken branch instructions):

  The value 4 is added to the address of the current instruction to form the 32-bit effective address of the next instruction. If the address of the current instruction is 0xFFFFFFFC, the PPC465 wraps the next sequential instruction address back to address 0. This behavior is not required by the architecture, which specifies that the next sequential instruction address is undefined under these circumstances. Therefore, software should not depend upon this behavior, so that it may be ported to other implementations that do not handle this scenario in the same fashion. Accordingly, if software wishes to execute across this maximum address boundary and wrap back to address 0, it should place an unconditional branch at the boundary, with a displacement of 4.

  In addition to the above four instruction storage addressing modes, the following behavior applies to branch instructions:

- Any branch instruction with LK=1:

  The value 4 is added to the address of the current instruction and the low-order 32 bits of the result are placed into the LR. As for the similar scenario for next sequential instruction fetching, if the address of the branch instruction is 0xFFFFFFFC, the result placed into the LR is architecturally undefined, although once again the

PPC465 wraps the LR update value back to address 0. Again, however, software should not depend on this behavior, in order that it may be ported to implementations which do not handle this scenario in the same fashion.

3.1.3 Byte Ordering

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 order arise.

For a machine in which the smallest addressable unit of storage is the 64-bit doubleword, there is no question of the ordering of bytes within doublewords. All transfers of individual scalars between registers and storage are of doublewords, 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 Book-E Enhanced PowerPC Architecture, as for most current computer architectures, the smallest addressable unit of storage is the 8-bit byte. Many 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 occupies four consecutive byte addresses. It thus becomes meaningful to discuss the order of the byte addresses with respect to the value of the scalar: 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 ("left-most") 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. IBM RISC System/6000, IBM System/390, and Motorola 680x0 are examples of computer architectures using this byte ordering.

- The ordering that assigns the lowest address to the lowest-order ("right-most") 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. The Intel x86 is an example of a processor architecture using this byte ordering.

PowerPC Book-E supports both big endian and little endian byte ordering, for both instruction and data storage accesses. Which byte ordering is used is controlled on a memory page basis by the endian (E) storage attribute, which is a field within the TLB entry for the page. The endian storage attribute is set to 0 for a big endian page, and is set to 1 for a little endian page. See *Memory Management* on page 219 for more information on memory pages, the TLB, and storage attributes, including the endian storage attribute.

3.1.3.1 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 below 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.

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). The shaded cells correspond to padded bytes.

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

Little Endian Mapping

Structure *s* is shown mapped little endian.

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

3.1.3.2 Instruction Byte Ordering

PowerPC Book-E defines instructions as aligned words (four bytes) in memory. As such, instructions in a big endian program image are arranged with the most-significant byte (MSB) of the instruction word at the lowest-numbered address.

Consider the big endian mapping of instruction *p* at address 0x00, where, for example, *p* = add r7, r7, r4:

MSB			LSB
0x00	0x01	0x02	0x03

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

LSB			MSB
0x00	0x01	0x02	0x03

By the definition of PowerPC Book-E bit numbering, the most-significant byte of an instruction is the byte containing bits 0:7 of the instruction. As depicted in the instruction format diagrams (see *Instruction Formats* on page 344), this most-significant byte is the one which contains the primary opcode field (bits 0:5). Due to this difference in byte orderings, the processor must perform whatever byte reversal is required (depending on the particular byte ordering in use) in order to correctly deliver the opcode field to the instruction decoder. In the PPC465, this reversal is performed between the memory interface and the instruction cache, according to the value of the endian storage attribute for each memory page, such that the bytes in the instruction cache are always correctly arranged for delivery directly to the instruction decoder.

If the endian storage attribute for a memory page is reprogrammed from one byte ordering to the other, the contents of the memory page must be reloaded with program and data structures that are in the appropriate byte ordering. Furthermore, anytime the contents of instruction memory change, the instruction cache must be made coherent with the updates by invalidating the instruction cache and refetching the updated memory contents with the new byte ordering.

3.1.3.3 Data Byte Ordering

Unlike instruction fetches, data accesses cannot be byte-reversed between memory and the data cache. Data byte ordering in memory depends upon the data type (byte, halfword, word, and so on) of a specific data item. It is only when moving a data item of a specific type from or to an architected register (as directed by the execution of a particular storage access instruction) that it becomes known what kind of byte reversal may be required due to the byte ordering of the memory page containing the data item. Therefore, byte reversal during load or store accesses is performed between data cache (or memory, on a data cache miss, for example) and the load register target or store register source, depending on the specific type of load or store instruction (that is, byte, halfword, word, and so on).

Comparing the big endian and little endian mappings of structure *s*, as shown in *Structure Mapping Examples* on page 45, the differences between the byte locations of any data item in the structure depends upon the size of the particular data item. For example (again referring to the big endian and little endian mappings of structure *s*):

- 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.

The size of the data item being loaded or stored must be known before the processor can decide whether, and if so, how to reorder the bytes when moving them between a register and the data cache (or memory).

- For byte loads and stores, including strings, no reordering of bytes occurs, regardless of byte ordering.
- For halfword loads and stores, bytes are reversed within the halfword, for one byte order with respect to the other.
- For word loads and stores (including load/store multiple), bytes are reversed within the word, for one byte order with respect to the other.
- For doubleword loads and stores (AP loads/stores only), bytes are reversed within the doubleword, for one byte order with respect to the other.
- For quadword loads and stores (AP loads/stores only), bytes are reversed within the quadword, for one byte order with respect to the other.

Note that this mechanism applies independent of the alignment of data. In other words, when loading a multi-byte data operand with a scalar load instruction, bytes are accessed from the data cache (or memory) starting with the byte at the calculated effective address and continuing with consecutively higher-numbered bytes until the required number of bytes have been retrieved. Then, the bytes are arranged such that either the byte from the highest-numbered address (for big endian storage regions) or the lowest-numbered address (for little endian storage regions) is placed into the least-significant byte of the register. The rest of the register is filled in corresponding order with the rest of the accessed bytes. An analogous procedure is followed for scalar store instructions.

For load/store multiple instructions, each group of four bytes is transferred between memory and the register according to the procedure for a scalar load word instruction.

For load/store string instructions, the most-significant byte of the first register is transferred to or from memory at the starting (lowest-numbered) effective address, regardless of byte ordering. Subsequent register bytes (from most-significant to least-significant, and then moving into the next register, starting with the most-significant byte, and so on) are transferred to or from memory at sequentially higher-numbered addresses. This behavior for byte strings ensures that if two strings are loaded into registers and then compared, the first bytes of the strings are treated as most significant with respect to the comparison.

3.1.3.4 Byte-Reverse Instructions

PowerPC Book-E defines load/store byte-reverse instructions which can access storage which is specified as being of one byte ordering in the same manner that a regular (that is, non-byte-reverse) load/store instruction would access storage which is specified as being of the opposite byte ordering. In other words, a load/store byte-reverse instruction to a big endian memory page transfers data between the data cache (or memory) and the register in the same manner that a normal load/store would transfer the data to or from a little endian memory page. Similarly, a load/store byte-reverse instruction to a little endian memory page transfers data between the data cache (or memory) and the register in the same manner that a normal load/store would transfer the data to or from a big endian memory page.

The function of the load/store byte-reverse instructions is useful when a particular memory page contains a combination of data with both big endian and little endian byte ordering. In such an environment, the Endian storage attribute for the memory page would be set according to the predominant byte ordering for the page, and the normal load/store instructions would be used to access data operands which used this predominant byte ordering. Conversely, the load/store byte-reverse instructions would be used to access the data operands which were of the other (less prevalent) byte ordering.

Software compilers cannot typically make general use of the load/store byte-reverse instructions, so they are ordinarily used only in special, hand-coded device drivers.

3.2 Registers

This section provides an overview of the register categories and types provided by the PPC465. Detailed descriptions of each of the registers are provided within the chapters covering the functions with which they are associated (for example, the cache control and cache debug registers are described in *Level 1 Cache* on page 125). An alphabetical summary of all registers is provided in *Register Summary* on page 599

All registers in the PPC465 are 32 bits wide, although certain bits in some registers are *reserved* and thus not necessarily implemented. For all registers with fields marked as reserved, these reserved fields should be written as 0 and read as *undefined*. The recommended coding practice is to perform the initial write to a register with reserved fields set to 0, 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.

All of the registers are grouped into categories according to the processor functions with which they are associated. In addition, each register is classified as being of a particular *type*, as characterized by the specific instructions which are used to read and write registers of that type. Finally, most of the registers contained within the PPC465 are defined by the Book-E Enhanced PowerPC Architecture, although some registers are implementation-specific and unique to the PPC465.

Figure 3-1 illustrates the PPC465 registers contained in the user programming model, that is, those registers to which access is non-privileged and which are available to both user and supervisor programs.

Figure 3-1. User Programming Model Registers

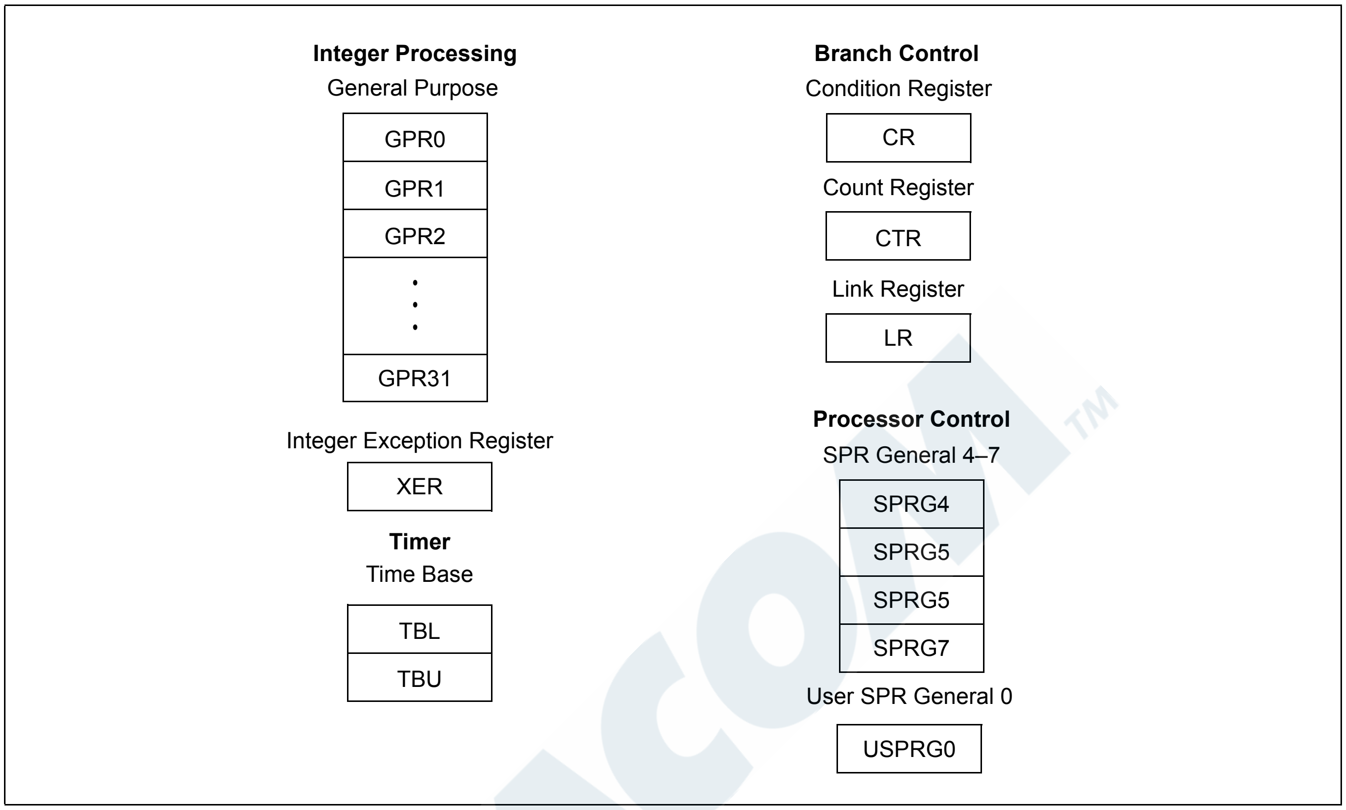


Figure 3-2 on page 50 illustrates the PPC465 registers contained in the supervisor programming model, to which access is privileged and which are available to supervisor programs only. See *User and Supervisor Modes* on page 77 for more information on privileged instructions and register access, and the user and supervisor programming models.

Figure 3-2. Supervisor Programming Model Registers

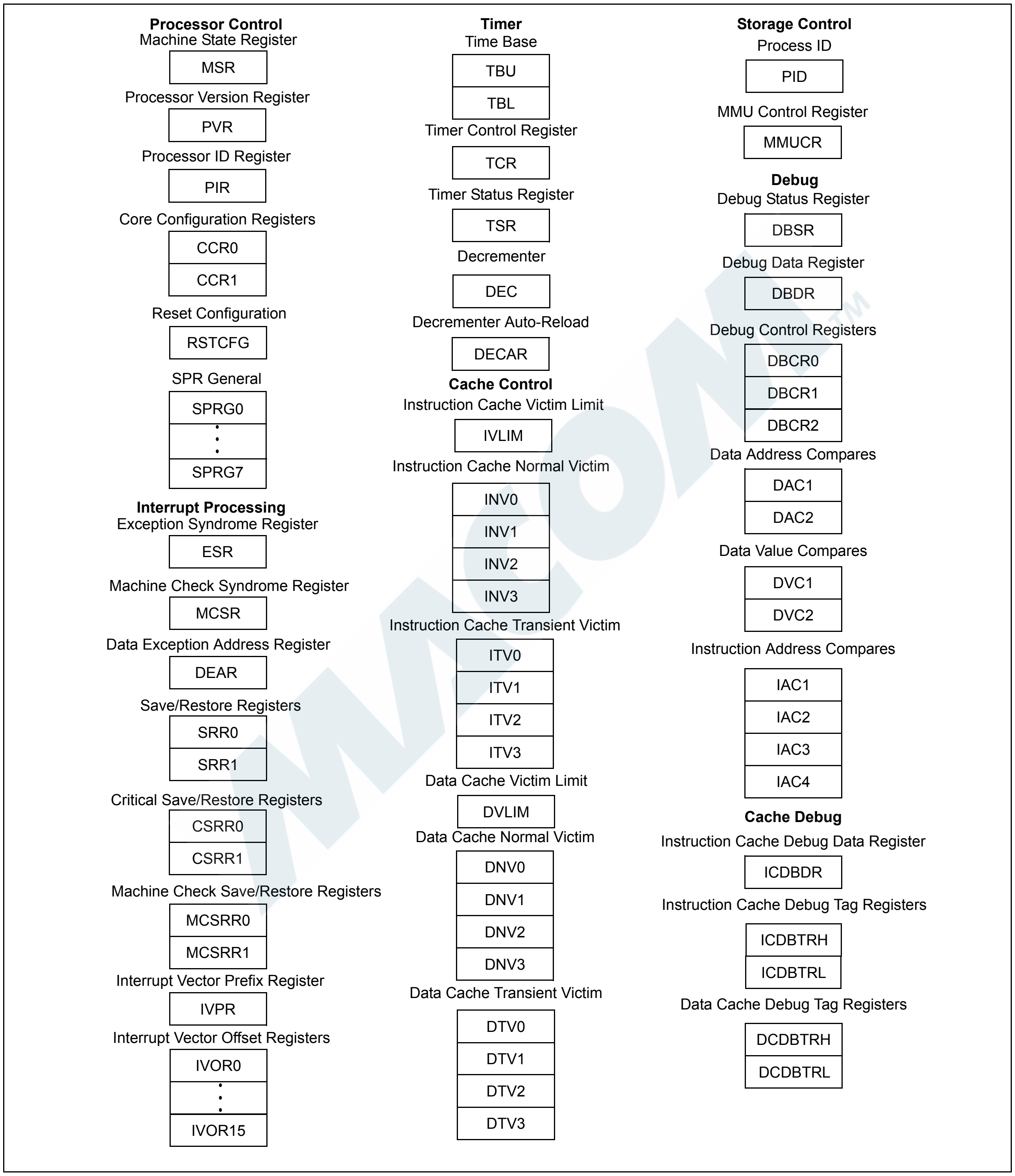


Table 3-3 lists each register category and the registers that belong to each category, along with their types and a cross-reference to the section of this document which describes them more fully. Registers that are not part of PowerPC Book-E, and are thus specific to the PPC465, are shown in italics in *Table 3-3*. Unless otherwise indicated, all registers have read/write access.

Table 3-3. Register Categories

Register Category	Register(s)	Model and Access	Type	Page
Branch Control	CR	User	CR	66
	CTR	User	SPR	66
	LR	User	SPR	66
Cache Control	*DNV0–DNV3*	Supervisor	SPR	126
	DTV0–DTV3	Supervisor	SPR	126
	DVLIM	Supervisor	SPR	128
	INV0–INV3	Supervisor	SPR	126
	ITV0–ITV3	Supervisor	SPR	126
	IVLIM	Supervisor	SPR	128
Cache Debug	*DCDBTRH, DCDBTRL*	Supervisor, read-only	SPR	151
	ICDBDR, ICDBTRH, ICDBTRL	Supervisor, read-only	SPR	136
Debug	DAC1–DAC2	Supervisor	SPR	340
	DBCR0–DBCR2	Supervisor	SPR	335
	DBDR	Supervisor	SPR	341
	DBSR	Supervisor	SPR	339
	DVC1–DVC2	Supervisor	SPR	340
	IAC1–IAC4	Supervisor	SPR	340
Device Control	Implemented outside core	Supervisor	DCR	53
	Implemented outside core	User	DCR	53
Integer Processing	GPR0–GPR31	User	GPR	69
	XER	User	SPR	70
Interrupt Processing	CSRR0–CSRR1	Supervisor	SPR	255
	DEAR	Supervisor	SPR	256
	ESR	Supervisor	SPR	258
	IVOR0–IVOR15	Supervisor	SPR	257
	IVPR	Supervisor	SPR	258
	MCSR	Supervisor	SPR	260
	MCSRR0-MCSRR1	Supervisor	SPR	255
	SRR0–SRR1	Supervisor	SPR	254

Table 3-3. Register Categories (continued)

Register Category	Register(s)	Model and Access	Type	Page
Processor Control	*CCR0*	Supervisor	SPR	135
	CCR1	Supervisor	SPR	135
	MSR	Supervisor	MSR	253
	PIR, PVR	Supervisor, read-only	SPR	73
	RSTCFG	Supervisor, read-only	SPR	76
	SPRG0–SPRG3	Supervisor	SPR	72
	SPRG4–SPRG7	User, read-only; Supervisor	SPR	72
	USPRG0	User	SPR	72
Storage Control	*MMUCR*	Supervisor	SPR	236
	PID	Supervisor	SPR	239
Timer	DEC	Supervisor	SPR	309
	DECAR	Supervisor, write-only	SPR	309
	TBL, TBU	User read, Supervisor write	SPR	308
	TCR	Supervisor	SPR	312
	TSR	Supervisor	SPR	313

3.2.1 Register Types

There are five register types contained within and/or supported by the PPC465. Each register type is characterized by the instructions which are used to read and write the registers of that type. The following subsections provide an overview of each of the register types and the instructions associated with them.

3.2.1.1 General Purpose Registers

The PPC465 contains 32 integer general purpose registers (GPRs); each contains 32 bits. Data from the data cache or memory can be loaded into GPRs using integer load instructions; the contents of GPRs can be stored to the data cache or memory using integer store instructions. Most of the integer instructions reference GPRs. The GPRs are also used as targets and sources for most of the instructions which read and write the other register types.

Integer Processing on page 69 provides more information on integer operations and the use of GPRs.

3.2.1.2 Special Purpose Registers

Special Purpose Registers (SPRs) are directly accessed using the **mtspr** and **mfspr** instructions. In addition, certain SPRs may be updated as a side-effect of the execution of various instructions. For example, the Integer Exception Register (XER) (see *Integer Exception Register (XER)* on page 70) is an SPR which is updated with arithmetic status (such as carry and overflow) upon execution of certain forms of integer arithmetic instructions.

SPRs control the use of the debug facilities, timers, interrupts, memory management, caches, and other architected processor resources. *Table 15-1* on page 599 shows the mnemonic, name, and number for each SPR, in order by SPR number. Each of the SPRs is described in more detail within the section or chapter covering the function with which it is associated. See *Table 3-3* on page 51 for a cross-reference to the associated document section for each register.

3.2.1.3 Condition Register

The Condition Register (CR) is a 32-bit register of its own unique type and is divided up into eight, independent 4-bit fields (CR0–CR7). The CR may be used to record certain conditional results of various arithmetic and logical operations. Subsequently, conditional branch instructions may designate a bit of the CR as one of the branch conditions (see *Branch Processing* on page 63). Instructions are also provided for performing logical bit operations and for moving fields within the CR.

See *Condition Register (CR)* on page 66 for more information on the various instructions which can update the CR.

3.2.1.4 Machine State Register

The Machine State Register (MSR) is a register of its own unique type that controls important chip functions, such as the enabling or disabling of various interrupt types.

The MSR can be written from a GPR using the **mtmsr** instruction. The contents of the MSR can be read into a GPR using the **mfmsr** instruction. The MSR[EE] bit can be set or cleared atomically using the **wrtee** or **wrteei** instructions. The MSR contents are also automatically saved, altered, and restored by the interrupt-handling mechanism. See *Machine State Register (MSR)* on page 253 for more detailed information on the MSR and the function of each of its bits.

3.2.1.5 Device Control Registers

Device Control Registers (DCRs) are on-chip registers that exist architecturally and physically outside the PPC465, and thus are not specified by the Book-E Enhanced PowerPC Architecture, nor by this user's manual for the PPC465. Rather, PowerPC Book-E simply defines the existence of the DCR address space and the instructions that access the DCRs, and does not define any particular DCRs. DCRs may be used to control various on-chip system functions, such as the operation of on-chip buses, peripherals, and certain processor core 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 to PPC465 has added additional instructions, such as **mtdcrx** (move to device control register indexed), **mtdcrux** (move to device control register user-mode indexed), **mfdcrx** (move from device control register indexed), and **mfdcrux** (move from device control register user-mode indexed).

The DCR access instructions, such as mtdcr, mfdcr, mtdcrx and mfdcrx, are privileged and executed by the privileged users only. The hardware provides access to those 'privileged' DCRs. However, mtdcrux and mfdcrux instructions can be executed by either privileged users or normal users and access those non-privileged DCRs implemented by hardware.

3.2.1.6 Memory Mapped Registers

Some registers associated with on-chip peripherals are memory-mapped input/output (MMIO) registers. Such registers are mapped into the system memory space and are accessed using load/store instructions that contain the register addresses.

3.3 Instruction Classes

PowerPC Book-E architecture defines all instructions as falling into exactly one of the following four classes, as determined by the primary opcode (and the extended opcode, if any):

1. Defined
2. Allocated
3. Preserved

4. Reserved (illegal or nop)

3.3.1 Defined Instruction Class

This class of instructions consists of all the instructions defined in PowerPC Book-E. In general, defined instructions are guaranteed to be supported within a PowerPC Book-E system as specified by the architecture, either within the processor implementation itself or within emulation software supported by the system operating software.

One exception to this is that, for implementations (such as the PPC465) that only provide the 32-bit subset of PowerPC Book-E, it is not expected (and likely not even possible) that emulation of the 64-bit behavior of the defined instructions will be provided by the system.

As defined by PowerPC Book-E, any attempt to execute a defined instruction will:

- Cause an Illegal Instruction exception type Program interrupt, if the instruction is not recognized by the implementation; or
- Cause an Unimplemented Instruction exception type Program interrupt, if the instruction is recognized by the implementation and is not a floating-point instruction, but is not supported by the implementation; or
- Cause a Floating-Point Unavailable interrupt if the instruction is recognized as a floating-point instruction, but floating-point processing is disabled; or
- Cause an Unimplemented Instruction exception type Program interrupt, if the instruction is recognized as a floating-point instruction and floating-point processing is enabled, but the instruction is not supported by the implementation; or
- Perform the actions described in the rest of this document, if the instruction is recognized and supported by the implementation. The architected behavior may cause other exceptions.

The PPC465 recognizes and fully supports all of the instructions in the defined class, with a few exceptions. First, because the PPC465 is a 32-bit implementation, those operations which are defined specifically for 64-bit operation are not supported at all, and will always cause an Illegal Instruction exception type Program interrupt.

Second, instructions that are defined for floating-point processing may be implemented within an auxiliary processor and attached to the core using the AP interface. If no such auxiliary processor is attached, attempting to execute any floating-point instructions will cause an Illegal Instruction exception type Program interrupt. If an auxiliary processor which supports the floating-point instructions *is* attached, the behavior of these instructions is as defined above and as determined by the implementation details of the floating-point auxiliary processor.

Finally, there are two other defined instructions which are not supported within the PPC465. One is a TLB management instruction (**tlbiva**, TLB Invalidate Virtual Address) that is specifically intended for coherent multiprocessor systems. The other is **mfapidi** (Move From Auxiliary Processor ID Indirect), which is a special instruction intended to assist with identification of the auxiliary processors which may be attached to a particular processor implementation. Since the PPC465 does not support **mfapidi**, the means of identifying the auxiliary processors in a PPC465-based system are implementation-dependent. Execution of either **tlbiva** or **mfapidi** will cause an Illegal Instruction exception type Program interrupt.

3.3.2 Allocated Instruction Class

This class of instructions contains a set of primary opcodes, as well as extended opcodes for certain primary opcodes. The specific opcodes are listed in *Appendix A.3* on page 643.

Allocated instructions are provided for purposes that are outside the scope of PowerPC Book-E, and are for implementation-dependent and application-specific use, including use within auxiliary processors.

PowerPC Book-E declares that any attempt to execute an allocated instruction results in one of the following effects:

- Causes an Illegal Instruction exception type Program interrupt, if the instruction is not recognized by the implementation
- Causes an Auxiliary Processor Unavailable interrupt if the instruction is recognized by the implementation, but allocated instruction processing is disabled
- Causes an Unimplemented Instruction exception type Program interrupt, if the instruction is recognized and allocated instruction processing is enabled, but the instruction is not supported by the implementation
- Perform the actions described for the particular implementation of the allocated instruction. The implementation-dependent behavior may cause other exceptions.

In addition to supporting the defined instructions of PowerPC Book-E, the PPC465 also implements a number of instructions which use the allocated instruction opcodes, and thus are not part of the PowerPC Book-E architecture. *Table 3-21 on page 63* identifies the allocated instructions that are implemented within the PPC465. All of these instructions are always enabled and supported, and thus they always perform the functions defined for them within this document, and never cause Illegal Instruction, Auxiliary Processor Unavailable, nor Unimplemented Instruction exceptions.

The PPC465 also supports the use of *any* of the allocated opcodes by an attached auxiliary processor, except for those allocated opcodes which have been implemented within the PPC465, as mentioned above. Also, there is one other allocated opcode (primary opcode 31, secondary opcode 262) that has been implemented within the PPC465 and is thus not available for use by an attached auxiliary processor. This is the opcode which was used on previous PowerPC 400 Series embedded controllers for the **icbt** (Instruction Cache Block Touch) instruction. The **icbt** instruction is now part of the defined instruction class for PowerPC Book-E, and uses a new opcode (primary opcode 31, secondary opcode 22). The PPC465 implements the new defined opcode, but also continues to support the previous opcode, in order to support legacy software written for earlier PowerPC 400 Series implementations. The **icbt** instruction description in *Instruction Set* on page 343 only identifies the defined opcode, although *Appendix A Instruction Summary* on page 609 includes both the defined and the allocated opcode in the table which lists all the instructions by opcode. In order to ensure portability between the PPC465 and future PowerPC Book-E implementations, software should take care to only use the defined opcode for **icbt**, and avoid usage of the previous opcode which is now in the allocated class.

3.3.3 Preserved Instruction Class

The preserved instruction class is provided to support backward compatibility with the PowerPC Architecture, and/or earlier versions of the PowerPC Book-E architecture. This instruction class includes opcodes which were defined for these previous architectures, but which are no longer defined for PowerPC Book-E.

Any attempt to execute a preserved instruction results in one of the following effects:

- Performs the actions described in the previous version of the architecture, if the instruction is recognized by the implementation
- Causes an Illegal Instruction exception type Program interrupt, if the instruction is not recognized by the implementation.

The only preserved instruction recognized and supported by the PPC465 is the **mftb** (Move From Time Base) opcode. This instruction was used in the PowerPC Architecture to read the Time Base Upper (TBU) and Time Base Lower (TBL) registers. PowerPC Book-E architecture instead defines TBU and TBL as Special Purpose Registers (SPRs), and thus the **mfspr** (Move From Special Purpose Register) instruction is used to read them. In order to enable legacy time base management software to be run on the PPC465, the processor core also supports the preserved opcode of **mftb**. However, the **mftb** instruction is not included in the various sections of this document that describe the implemented instructions, and software should take care to use the currently architected mechanism of **mfspr** to read the time base registers, in order to guarantee portability between the PPC465 and future implementations of PowerPC Book-E.

On the other hand, *Appendix A Instruction Summary* on page 609 does identify the **mftb** instruction as an implemented preserved opcode in the table which lists all the instructions by opcode.

3.3.4 Reserved Instruction Class

This class of instructions consists of all instruction primary opcodes (and associated extended opcodes, if applicable) which do not belong to either the defined, allocated, or preserved instruction classes.

Reserved instructions are available for future versions of PowerPC Book-E architecture. That is, future versions of PowerPC Book-E may define any of these instructions to perform new functions or make them available for implementation-dependent use as allocated instructions. There are two types of reserved instructions: reserved-illegal and reserved-nop.

Any attempt to execute a reserved-illegal instruction will cause an Illegal Instruction exception type Program interrupt. Reserved-illegal instructions are, therefore, available for future extensions to PowerPC Book-E that would affect architected state. Such extensions might include new forms of integer or floating-point arithmetic instructions, or new forms of load or store instructions that affect architected registers or the contents of memory.

Any attempt to execute a reserved-nop instruction, on the other hand, either has no effect (that is, is treated as a no-operation instruction), or causes an Illegal Instruction exception type Program interrupt. Because implementations are typically expected to treat reserved-nop instructions as true no-ops, these instruction opcodes are thus available for future extensions to PowerPC Book-E which have no effect on architected state. Such extensions might include performance-enhancing hints, such as new forms of cache touch instructions. Software would be able to take advantage of the functionality offered by the new instructions, and still remain backwards-compatible with implementations of previous versions of PowerPC Book-E.

The PPC465 implements all of the reserved-nop instruction opcodes as true no-ops. The specific reserved-nop opcodes are listed in *Appendix A.5* on page 644

3.4 Implemented Instruction Set Summary

This section provides an overview of the various types and categories of instructions implemented within the PPC465. In addition, *Instruction Set* on page 343 provides a complete alphabetical listing of every implemented instruction, including its register transfer language (RTL) and a detailed description of its operation. Also, *Appendix A Instruction Summary* on page 609 lists each implemented instruction alphabetically (and by opcode) along with a short-form description and its extended mnemonic(s).

Table 3-4 summarizes the PPC465 instruction set by category. Instructions within each category are described in subsequent sections.

Table 3-4. Instruction Categories

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

3.4.1 Integer Instructions

Integer instructions transfer data between memory and the GPRs, and perform various operations on the GPRs. This category of instructions is further divided into seven sub-categories, described below.

3.4.1.1 Integer Storage Access Instructions

Integer storage access instructions load and store data between memory and the GPRs. These instructions operate on bytes, halfwords, and words. Integer storage access instructions also support loading and storing multiple registers, character strings, and byte-reversed data, and loading data with sign-extension.

Table 3-5 shows the integer storage access instructions in the PPC465. In the table, the syntax “[**u**]” indicates that the instruction has both an “update” form (in which the RA addressing register is updated with the calculated address) and a “non-update” form. Similarly, the syntax “[**x**]” indicates that the instruction has both an “indexed” form (in which the address is formed 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 (specified as part of the instruction) to the contents of GPR RA. See the detailed instruction descriptions in *Instruction Set* on page 343.

Table 3-5. Integer Storage Access 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**

3.4.1.2 Integer Arithmetic Instructions

Arithmetic operations are performed on integer or ordinal operands stored in registers. 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 registers. The result is placed in a third register. Instructions that perform operations on one operand are defined in a two-operand format; the operation is performed on the operand in a register and the result is placed in another register. Several instructions also have immediate formats in which one of the source operands is a field in the instruction.

Most integer arithmetic instructions have versions that can update CR[CR0] and/or XER[SO OV] (Summary Overflow, Overflow), based on the result of the instruction. Some integer arithmetic instructions also update XER[CA] (Carry) implicitly. See *Integer Processing* on page 69 for more information on how these instructions update the CR and/or the XER.

Table 3-6 lists the integer arithmetic instructions in the PPC465. In the table, the syntax “[**o**]” indicates that the instruction has both an “o” form (which updates the XER[SO,OV] fields) and a “non-o” form. Similarly, the syntax “[**.**]” indicates that the instruction has both a “record” form (which updates CR[CR0]) and a “non-record” form.

Table 3-6. Integer 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**][**.**]

3.4.1.3 Integer Logical Instructions

Table 3-7 lists the integer logical instructions in the PPC465. See *Integer Arithmetic Instructions* on page 58 for an explanation of the "**[.]**" syntax.

Table 3-7. Integer 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**[.]

3.4.1.4 Integer Compare Instructions

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

Table 3-8 lists the integer compare instructions in the PPC465.

Table 3-8. Integer Compare Instructions

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

3.4.1.5 Integer Trap Instructions

Table 3-9 lists the integer trap instructions in the PPC465.

Table 3-9. Integer Trap Instructions

Trap
tw **twi**

3.4.1.6 Integer Rotate Instructions

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

Table 3-10 lists the rotate instructions in the PPC465. See *Integer Arithmetic Instructions* on page 58 for an explanation of the "[.]" syntax.

Table 3-10. Integer Rotate Instructions

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

3.4.1.7 Integer Shift Instructions

Table 3-11 lists the integer shift instructions in the PPC465. Note that the shift right algebraic instructions implicitly update the XER[CA] field. See *Integer Arithmetic Instructions* on page 58 for an explanation of the “[**.**]” syntax.

Table 3-11. Integer Shift Instructions

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

3.4.1.8 Integer Select Instruction

Table 3-12 lists the integer select instruction in the PPC465. The RA operand is 0 if the RA field of the instruction is 0, or is the contents of GPR[RA] otherwise.

Table 3-12. Integer Select Instruction

Integer Select
isel

3.4.2 Branch Instructions

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

See *Branch Processing* on page 63 for more information on branch operations.

Table 3-13 lists the branch instructions in the PPC465. In the table, the syntax “[**l**]” indicates that the instruction has both a “link update” form (which updates LR with the address of the instruction after the branch) and a “non-link update” form. Similarly, the syntax “[**a**]” indicates that the instruction has both 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 specified immediate field to the address of the branch instruction).

Table 3-13. Branch Instructions

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

3.4.3 Processor Control Instructions

Processor control instructions manipulate system registers, perform system software linkage, and synchronize processor operations. The instructions in these three sub-categories of processor control instructions are described below.

3.4.3.1 Condition Register Logical Instructions

These instructions perform logical operations on a specified pair of bits in the CR, placing the result in another specified bit. The benefit of these instructions is that they can logically combine the results of several comparison operations without incurring the overhead of conditional branching between each one. Software performance can significantly improve if multiple conditions are tested at once as part of a branch decision.

Table 3-14 lists the condition register logical instructions in the PPC465.

Table 3-14. Condition Register Logical Instructions

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

3.4.3.2 Register Management Instructions

These instructions move data between the GPRs and control registers in the PPC465.

Table 3-15 lists the register management instructions in the PPC465.

Table 3-15. Register Management Instructions

CR	DCR	MSR	SPR
mcrf **mcrxr** **mfcr** **mtcrf**	**mfdcr** **mfdcrx** **mfdcrux** **mtdcr** **mtdcrx** **mtdcrux**	**mfmsr** **mtmsr** **wrtee** **wrteei**	**mfspr** **mtspr**

3.4.3.3 System Linkage Instructions

These instructions invoke supervisor software level for system services, and return from interrupts.

Table 3-16 lists the system linkage instructions in the PPC465.

Table 3-16. System Linkage Instructions

rfi
rfci
rfmci
sc

3.4.3.4 Processor Synchronization Instruction

The processor synchronization instruction, **isync**, forces the processor to complete all instructions preceding the **isync** before allowing any context changes as a result of any instructions that follow the **isync**. Additionally, all instructions that follow the **isync** will execute within the context established by the completion of all the instructions that precede the **isync**. See *Synchronization* on page 79 for more information on the synchronizing effect of **isync**.

Table 3-17 shows the processor synchronization instruction in the PPC465.

Table 3-17. Processor Synchronization Instruction

isync

3.4.4 Storage Control Instructions

These instructions manage the instruction and data caches and the TLB of the PPC465. Instructions are also provided to synchronize and order storage accesses. The instructions in these three sub-categories of storage control instructions are described below.

3.4.4.1 Cache Management Instructions

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

Table 3-18 lists the cache management instructions in the PPC465.

Table 3-18. Cache Management Instructions

Data Cache	Instruction Cache
dcba **dcbf** **dcbi** **dcbst** **dcbt** **dcbtst** **dcbz**	**icbi** **icbt**

3.4.4.2 TLB Management Instructions

The TLB management instructions read and write entries of the TLB array, and search the TLB array for an entry which will translate a given virtual address. There is also an instruction for synchronizing TLB updates with other processors, but since the PPC465 is intended for use in uni-processor environments, this instruction performs no operation on the PPC465.

Table 3-19 lists the TLB management instructions in the PPC465. See *Integer Arithmetic Instructions* on page 58 for an explanation of the “[**.**]” syntax.

Table 3-19. TLB Management Instructions

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

3.4.4.3 Storage Synchronization Instructions

The storage synchronization instructions allow software to enforce ordering amongst the storage accesses caused by load and store instructions, which by default are “weakly-ordered” by the processor. “Weakly-ordered” means that the processor is architecturally permitted to perform loads and stores generally out-of-order with respect to their sequence within the instruction stream, with some exceptions. However, if a storage synchronization instruction is executed, then all storage accesses prompted by instructions preceding the synchronizing instruction must be performed before any storage accesses prompted by instructions which come after the synchronizing instruction. See *Synchronization* on page 79 for more information on storage synchronization.

Table 3-17 shows the storage synchronization instructions in the PPC465.

Table 3-20. Storage Synchronization Instructions

msync **mbar**

3.4.5 Allocated Instructions

These instructions are not part of the PowerPC Book-E architecture, but they are included as part of the PPC465. Architecturally, they are considered allocated instructions, as they use opcodes which are within the allocated class of instructions, which the PowerPC Book-E architecture identifies as being available for implementation-dependent and/or application-specific purposes. However, all of the allocated instructions which are implemented within the PPC465 are "standard" for the family of PowerPC embedded controllers, and are not unique to the PPC465.

The allocated instructions implemented within the PPC465 are divided into four sub-categories, and are shown in *Table 3-21*. See *Integer Arithmetic Instructions* on page 58 for an explanation of the "[**.**]" and "[**o**]" syntax.

Table 3-21. Allocated Instructions

Arithmetic			Logical	Cache Management	Cache Debug
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**[.]	**dlmzb**[.]	**dccci** **iccci**	**dcread** **icread**

3.5 Branch Processing

The four branch instructions provided by PPC465 are summarized in *Table 3.4.2* on page 60. In addition, each of these instructions is described in detail in *Instruction Set* on page 343. The following sections provide additional information on branch addressing, instruction fields, prediction, and registers.

3.5.1 Branch Addressing

The branch instruction (**b**[**l**][**a**]) specifies the displacement of the branch target address as a 26-bit value (the 24-bit LI field right-extended with 0b00). This displacement is regarded as a signed 26-bit number covering an address range of ±32MB. Similarly, the branch conditional instruction (**bc**[**l**][**a**]) specifies the displacement as a 16-bit value (the 14-bit BD field right-extended with 0b00). This displacement covers an address range of ±32KB.

For the relative form of the branch and branch conditional instructions (**b**[**l**] and **bc**[**l**], with instruction field AA = 0), the target address is the address of the branch instruction itself (the Current Instruction Address, or CIA) plus the signed displacement. This address calculation is defined to "wrap around" from the maximum effective address (0xFFFFFFFF) to 0x0000 0000, and vice-versa.

For the absolute form of the branch and branch conditional instructions (**ba[l]** and **bca[l]**, with instruction field AA = 1), the target address is the sign-extended displacement. This means that with absolute forms of the branch and branch conditional instructions, the branch target can be within the first or last 32MB or 32KB of the address space, respectively.

The other two branch instructions, **bclr** (branch conditional to LR) and **bcctr** (branch conditional to CTR), do not use absolute nor relative addressing. Instead, they use *indirect* addressing, in which the target of the branch is specified indirectly as the contents of the LR or CTR.

3.5.2 Branch Instruction BI Field

Conditional branch instructions can optionally test one bit of the CR, as indicated by instruction field BO[0] (see BO field description below). The value of instruction field BI specifies the CR bit to be tested (0-31). The BI field is ignored if BO[0] = 1. The branch (**b[l][a]**) instruction is by definition unconditional, and hence does not have a BI instruction field. Instead, the position of this field is part of the LI displacement field.

3.5.3 Branch Instruction BO Field

The BO field specifies the condition under which a conditional branch is taken, and whether the branch decrements the CTR. The branch (**b[l][a]**) instruction is by definition unconditional, and hence does not have a BO instruction field. Instead, the position of this field is part of the LI displacement field.

Conditional branch instructions can optionally 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 the CR condition option is selected, the condition is satisfied (branch can occur) if the CR bit selected by the BI instruction field matches BO[1].

Conditional branch instructions can also optionally decrement the CTR by one, and test whether the decremented value is 0. This option is selected when BO[2] = 0; if BO[2] = 1, the CTR is not decremented and does not participate in the branch condition test. If CTR decrement option is selected, BO[3] specifies the condition that must be satisfied to allow the branch to be taken. If BO[3] = 0, CTR ≠ 0 is required for the branch to occur. If BO[3] = 1, CTR = 0 is required for the branch to occur.

Table 3-22 summarizes the usage of the bits of the BO field. BO[4] is further discussed in *Branch Prediction* on page 65

Table 3-22. BO Field Definition

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 If BO[0] = 0, test for CR[BI] = 0. 1 If BO[0] = 0, test for CR[BI] = 1.
BO[2]	CTR Decrement and Test Control 0 Decrement CTR by one and test whether the decremented CTR satisfies the condition specified by BO[3]. 1 Do not decrement CTR, do not test CTR.
BO[3]	CTR Test Value 0 If BO[2] = 0, test for decremented CTR ≠ 0. 1 If BO[2] = 0, test for decremented CTR = 0.
BO[4]	Branch Prediction Reversal 0 Apply standard branch prediction. 1 Reverse the standard branch prediction.

Table 3-23 lists specific BO field contents, and the resulting actions; *z* represents a mandatory value of zero, and *y* is a branch prediction option discussed in *Branch Prediction* on page 65

Table 3-23. BO Field Examples

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.

3.5.4 Branch Prediction

Conditional branches might be taken or not taken; if taken, instruction fetching is re-directed to the target address. If the branch is not taken, instruction fetching simply falls through to the next sequential instruction. The PPC465 attempts to predict whether or not a branch is taken before all information necessary to determine the branch direction is available. This action is called *branch prediction*. The processor core can then prefetch instructions down the predicted path. If the prediction is correct, performance is improved because the branch target instruction is available immediately, instead of having to wait until the branch conditions are resolved. If the prediction is incorrect, then the prefetched instructions (which were fetched from addresses down the “wrong” path of the branch) must be discarded, and new instructions fetched from the correct path.

The PPC465 combines the static prediction mechanism defined by PowerPC Book-E, together with a dynamic branch prediction mechanism, in order to provide correct branch prediction as often as possible. The dynamic branch prediction mechanism is an implementation optimization, and is not part of the architecture, nor is it visible to the programming model. *Appendix B Instruction Execution Performance and Code Optimizations* on page 653 provides additional information on the dynamic branch prediction mechanism.

The static branch prediction mechanism enables software to designate the “preferred” branch prediction via bits in the instruction encoding. The “default” static branch prediction for conditional branches is as follows:

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

where *s* is bit 16 of the instruction (the sign bit of the displacement for all **bc** forms, and zero for all **bclr** and **bcctr** forms). In other words, conditional branches are predicted taken if they are “unconditional” (i.e., they do not test the CR nor the CTR decrement, and are always taken), or if their branch displacement is “negative” (i.e., the branch is branching “backwards” from the current instruction address). The standard prediction for this case derives from considering the relative form of **bc**, often used at the end of loops to control the number of times that a loop is executed. The branch is taken each 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. Because this situation is most common, a branch is taken if *s* = 1.

If branch displacements are positive, *s* = 0, then the branch is predicted not taken. Also, if the branch instruction is any form of **bclr** or **bcctr** except the “unconditional” form, 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 *Branch Addressing* on page 63, if *s* = 1, the branch target is in high memory. If *s* = 0, the branch target 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 standard prediction is taken, and for the low memory case the standard prediction is not taken.

Another bit in the BO field allows software further control over branch prediction. Specifically, BO[4] is the *prediction reversal bit*. If BO[4] = 0, the default prediction is applied. If BO[4] = 1, the reverse of the default prediction is applied. For the cases in *Table 3-23* where BO[4] = *y*, software can reverse the default prediction by setting *y* to 1. 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, as BO[4] is designated as *z* for this case, meaning the bit must be set to 0 or the instruction form is invalid.

3.5.5 Branch Control Registers

There are three registers in the PPC465 which are associated with branch processing, and they are described in the following sections.

3.5.5.1 Link Register (LR)

The LR is written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**. The LR can also be updated by the "link update" form of branch instructions (instruction field LK = 1). Such branch instructions load the LR with the address of the instruction following the branch instruction (4 + address of the branch instruction). Thus, the LR contents can be used as a return address for a subroutine that was entered using a link update form of branch. The **bclr** instruction uses the LR in this fashion, enabling indirect branching to any address.

When being used as a return address by a **bclr** instruction, bits 30:31 of the LR are ignored, since all instruction addresses are on word boundaries.
Access to the LR is non-privileged.

Figure 3-3. Link Register (LR)

0:31		Link Register contents	Target address of **bclr** instruction

3.5.5.2 Count Register (CTR)

The CTR is written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**. The CTR contents can be used as a loop count that gets decremented and tested by conditional branch instructions that specify count decrement as one of their branch conditions (instruction field BO[2] = 0). Alternatively, the CTR contents can specify a target address for the **bcctr** instruction, enabling indirect branching to any address.

Access to the CTR is non-privileged.

Figure 3-4. Count Register (CTR)

0:31		Count	Used as count for branch conditional with decrement instructions, or as target address for **bcctr** instructions

3.5.5.3 Condition Register (CR)

The CR is used to record certain information ("conditions") related to the results of the various instructions which are enabled to update the CR. A bit in the CR may also be selected to be tested as part of the condition of a conditional branch instruction.

The CR is organized into eight 4-bit fields (CR0–CR7), as shown in *Figure 3-5*. *Table 3-24* lists the instructions which update the CR.

Access to the CR is non-privileged.

Figure 3-5. 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	

Table 3-24. CR Updating Instructions

Integer						Processor Control	Storage Control	
Storage Access	**Arithmetic**	**Logical**	**Compare**	**Rotate**	**Shift**	**CR-Logical and Register Management**	**TLB Mgmt.**	**Logical**
stwcx.	**add.**[o] **addc.**[o] **adde.**[o] **addic.** **addme.**[o] **addze.**[o] **subf.**[o] **subfc.**[o] **subfe.**[o] **subfme.**[o] **subfze.**[o] **mulhw.** **mulhwu.** **mullw.**[o] **divw.**[o] **divwu.**[o] **neg.**[o]	**and.** **andi.** **andis.** **andc.** **nand.** **or.** **orc.** **nor.** **xor.** **eqv.** **extsb.** **extsh.** **cntlzw.**	**cmp** **cmpi** **cmpl** **cmpli**	**rlwimi.** **rlwinm.** **rlwnm.**	**slw.** **srw.** **sraw.** **srawi.**	**crand** **crandc** **creqv** **crnand** **crnor** **cror** **crorc** **crxor** **mcrf** **mcrxr** **mtcrf**	**tlbsx.**	**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.** **dlmzb.**

Instruction Set on page 343, provides detailed information on how each of these instructions updates the CR. To summarize, the CR can be accessed in any of the following ways:

- **mfcr** reads the CR into a GPR. Note that this instruction does not *update* the CR and is therefore not listed in *Table 3-24*.
- Conditional branch instructions can designate a CR bit to be used as a branch condition. Note that these instructions do not *update* the CR and are therefore not listed in *Table 3-24*.
- **mtcrf** sets specified CR fields by writing to the CR from a GPR, under control of a mask field specified as part of the instruction.
- **mcrf** updates a specified CR field by copying another specified CR field into it.
- **mcrxr** copies certain bits of the XER into a specified CR field, and clears the corresponding XER bits.
- Integer compare instructions update a specified CR field.
- CR-logical instructions update a specified CR bit with the result of any one of eight logical operations on a specified pair of CR bits.
- Certain forms of various integer instructions (the "." forms) implicitly update CR[CR0], as do certain forms of the auxiliary processor instructions implemented within the PPC465.
- Auxiliary processor instructions may in general update a specified CR field in an implementation-specified manner. In addition, if an auxiliary processor implements the floating-point operations specified by PowerPC Book-E, then those instructions update the CR in the manner defined by the architecture. See *Book E: PowerPC Architecture Enhanced for Embedded Applications* for details.

CR[CR0] Implicit Update By Integer Instructions

Most of the CR-updating instructions listed in *Table 3-24* implicitly update the CR0 field. These are the various "dot-form" instructions, indicated by a "**.**" in the instruction mnemonic. Most of these instructions update CR[CR0] according to an arithmetic comparison of 0 with the 32-bit result which the instruction writes to the GPR file. That is, after performing the operation defined for the instruction, the 32-bit result which is written to the GPR file is compared to 0 using a signed comparison, independent of whether the actual operation being performed by the instruction is considered "signed" or not. For example, logical instructions such as **and.**, **or.**, and **nor.** update CR[CR0] according to this signed comparison to 0, even though the result of such a logical operation is not typically interpreted as a signed value. For each of these dot-form instructions, the individual bits in CR[CR0] are updated as follows:

CR[CR0]$_0$ — LT	Less than 0; set if the most-significant bit of the 32-bit result is 1.
CR[CR0]$_1$ — GT	Greater than 0; set if the 32-bit result is non-zero and the most-significant bit of the result is 0.
CR[CR0]$_2$ — EQ	Equal to 0; set if the 32-bit result is 0.
CR[CR0]$_3$ — SO	Summary overflow; a copy of XER[SO] at the completion of the instruction (including any XER[SO] update being performed the instruction itself.

Note that if an arithmetic overflow occurs, the "sign" of an instruction result indicated in CR[CR0] 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]$_0$ is set, even though the infinitely precise result of the add is positive.

Similarly, 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]$_2$ is set, indicating a result of 0, but the infinitely precise result is negative.

CR[CR0]$_3$ is a copy of XER[SO] at the completion of the instruction, whether or not the instruction which is updating CR[CR0] is also updating XER[SO]. Note that if an instruction causes an arithmetic overflow but is not of the form which actually updates XER[SO], then the value placed in CR[CR0]$_3$ does not reflect the arithmetic overflow which occurred on the instruction (it is merely a copy of the value of XER[SO] which was already in the XER before the execution of the instruction updating CR[CR0]).

There are a few dot-form instructions which do not update CR[CR0] in the fashion described above. These instructions are: **stwcx.**, **tlbsx.**, and **dlmzb.** See the instruction descriptions in *Instruction Set* on page 343 for details on how these instructions update CR[CR0].

CR Update By Integer Compare Instructions

Integer compare instructions update a specified CR field with the result of a comparison of two 32-bit numbers, the first of which is from a GPR and the second of which is either an immediate value or from another GPR. There are two types of integer compare instructions, *arithmetic* and *logical*, and they are distinguished by the interpretation given to the 32-bit numbers being compared. For *arithmetic* compares, the numbers are considered to be signed, whereas for *logical* compares, the numbers are considered to be unsigned. As an example, consider the comparison of 0 with 0xFFFFFFFF. In an *arithmetic* compare, 0 is larger; in a *logical* compare, 0xFFFFFFFF is larger.

A compare instruction can direct its result to any CR field. The BF field (bits 6:8) of the instruction specifies the CR field to be updated. After a compare, the specified CR field is interpreted as follows:

CR[(BF)]$_0$ — LT	The first operand is less than the second operand.
CR[(BF)]$_1$ — GT	The first operand is greater than the second operand.
CR[(BF)]$_2$ — EQ	The first operand is equal to the second operand.
CR[(BF)]$_3$ — SO	Summary overflow; a copy of XER[SO].

3.6 Integer Processing

Integer processing includes loading and storing data between memory and GPRs, as well as performing various operations on the values in GPRs and other registers (the categories of integer instructions are summarized in *Table 3-4* on page 57). The sections which follow describe the registers which are used for integer processing, and how they are updated by various instructions. In addition, *Condition Register (CR)* on page 66 provides more information on the CR updates caused by integer instructions. Finally, *Instruction Set* on page 343 also provides details on the various register updates performed by integer instructions.

3.6.1 General Purpose Registers (GPRs)

The PPC465 contains 32 GPRs. The contents of these registers can be transferred to and from memory using integer storage access instructions. Operations are performed on GPRs by most other instructions.

Access to the GPRs is non-privileged.

Figure 3-6. General Purpose Registers (R0-R31)

0:31		General Purpose Register data	

3.6.2 Integer Exception Register (XER)

The XER records overflow and carry indications from integer arithmetic and shift instructions. It also provides a byte count for string indexed integer storage access instructions (**lswx** and **stswx**). Note that the term *exception* in the name of this register does not refer to exceptions as they relate to interrupts, but rather to the *arithmetic* exceptions of carry and overflow.

Figure 3-7 illustrates the fields of the XER, while *Table 3-25* and *Table 3-26* list the instructions which update XER[SO,OV] and the XER[CA] fields, respectively. The sections which follow the figure and tables describe the fields of the XER in more detail.

Access to the XER is non-privileged.

Figure 3-7. Integer Exception Register (XER)

0	SO	Summary Overflow 0 No overflow has occurred. 1 Overflow has occurred.	Can be *set* by **mtspr** or by integer or auxiliary processor instructions with the [**o**] option; can be *reset* by **mtspr** or by **mcrxr**.
1	OV	Overflow 0 No overflow has occurred. 1 Overflow has occurred.	Can be *set* by **mtspr** or by integer or allocated instructions with the [**o**] option; can be *reset* by **mtspr**, by **mcrxr**, or by integer or allocated instructions with the [**o**] option.
2	CA	Carry 0 Carry has not occurred. 1 Carry has occurred.	Can be *set* by **mtspr** or by certain integer arithmetic and shift instructions; can be *reset* by **mtspr**, by **mcrxr**, or by certain integer arithmetic and shift instructions.
3:24		Reserved	
25:31	TBC	Transfer Byte Count	Used as a byte count by **lswx** and **stswx**; written by **dlmzb**[.] and by **mtspr.**

Table 3-25. XER[SO,OV] Updating Instructions

Integer Arithmetic							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**[.] **macchwso**[.] **macchwsuo**[.] **macchwuo**[.] **machhwo**[.] **machhwso**[.] **machhwsuo**[.] **machhwuo**[.] **maclhwo**[.] **maclhwso**[.] **maclhwsuo**[.] **maclhwuo**[.]	**nmacchwo**[.] **nmacchwso**[.] **nmachhwo**[.] **nmachhwso**[.] **nmaclhwo**[.] **nmaclhwso**[.]	**mtspr** **mcrxr**

Table 3-26. 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**

3.6.2.1 Summary Overflow (SO) Field

This field is set to 1 when an instruction is executed that causes XER[OV] to be set to 1, except for the case of **mtspr**(XER), which writes XER[SO,OV] with the values in $(RS)_{0:1}$, respectively. Once set, XER[SO] is not reset until either an **mtspr**(XER) is executed with data that explicitly writes 0 to XER[SO], or until an **mcrxr** instruction is executed. The **mcrxr** instruction sets XER[SO] (as well as XER[OV,CA]) to 0 after copying all three fields into $CR[CR0]_{0:2}$ (and setting $CR[CR0]_3$ to 0).

Given this behavior, XER[SO] does not necessarily indicate that an overflow occurred on the most recent integer arithmetic operation, but rather that one occurred at some time subsequent to the last clearing of XER[SO] by **mtspr**(XER) or **mcrxr**.

XER[SO] is read (along with the rest of the XER) into a GPR by **mfspr**(XER). In addition, various integer instructions copy XER[SO] into $CR[CR0]_3$ (see *Condition Register (CR)* on page 66).

3.6.2.2 Overflow (OV) Field

This field is updated by certain integer arithmetic instructions to indicate whether the infinitely precise result of the operation can be represented in 32 bits. For those integer arithmetic instructions that update XER[OV] and produce *signed* results, XER[OV] = 1 if the result is greater than $2^{31} – 1$ or less than $–2^{31}$; otherwise, XER[OV] = 0. For those integer arithmetic instructions that update XER[OV] and produce *unsigned* results (certain integer divide instructions and multiply-accumulate instructions), XER[OV] = 1 if the result is greater than $2^{32}–1$; otherwise, XER[OV] = 0. See the instruction descriptions in *Instruction Set* on page 343 for more details on the conditions under which the integer divide instructions set XER[OV] to 1.

The **mtspr**(XER) and **mcrxr** instructions also update XER[OV]. Specifically, **mcrxr** sets XER[OV] (and XER[SO,CA]) to 0 after copying all three fields into $CR[CR0]_{0:2}$ (and setting $CR[CR0]_3$ to 0), while **mtspr**(XER) writes XER[OV] with the value in $(RS)_1$.

XER[OV] is read (along with the rest of the XER) into a GPR by **mfspr**(XER).

3.6.2.3 Carry (CA) Field

This field is updated by certain integer arithmetic instructions (the “carrying” and “extended” versions of add and subtract) to indicate whether or not there is a carry-out of the most-significant bit of the 32-bit result. XER[CA] = 1 indicates a carry. The integer shift right algebraic instructions update XER[CA] to indicate whether or not any 1-bits were shifted out of the least significant bit of the result, if the source operand was negative (see the instruction descriptions in *Instruction Set* on page 343 for more details).

The **mtspr**(XER) and **mcrxr** instructions also update XER[CA]. Specifically, **mcrxr** sets XER[CA] (as well as XER[SO,OV]) to 0 after copying all three fields into $CR[CR0]_{0:2}$ (and setting $CR[CR0]_3$ to 0), while **mtspr**(XER) writes XER[CA] with the value in $(RS)_2$.

XER[CA] is read (along with the rest of the XER) into a GPR by **mfspr**(XER). In addition, the “extended” versions of the add and subtract integer arithmetic instructions use XER[CA] as a source operand for their arithmetic operations.

Transfer Byte Count (TBC) Field

The TBC field is used by the string indexed integer storage access instructions (**lswx** and **stswx**) as a byte count. The TBC field is updated by the **dlmzb[.]** instruction with a value indicating the number of bytes up to and including the zero byte detected by the instruction (see the instruction description for **dlmzb** in *Instruction Set* on page 343 for more details). The TBC field is also written by **mtspr**(XER) with the value in $(RS)_{25:31}$.

XER[TBC] is read (along with the rest of the XER) into a GPR by **mfspr**(XER).

3.7 Processor Control

The PPC465 provides several registers for general processor control and status. These include:

- Machine State Register (MSR)

  Controls interrupts and other processor functions
- Special Purpose Registers General (SPRGs)

  SPRs for general purpose software use
- Processor Version Register (PVR)

  Indicates the specific implementation of a processor
- Processor Identification Register (PIR)

  Indicates the specific instance of a processor in a multi-processor system
- Core Configuration Register 0 (CCR0)

  Controls specific processor functions, guaranteed touch operation
- Reset Configuration (RSTCFG)

  Reports the values of certain fields of the TLB as supplied at reset
- Device Control Register Immediate Prefix Register (DCRIPR)

  The Device Control Register Immediate Prefix Register (DCRIPR) provides the upper order 22 bits of the DCR address to be used by the mtdcr and mfdcr. This SPR has hex address 0x37B, can be read and written, and is privileged.Except for the MSR, each of these registers is described in more detail in the following sections. The MSR is described in more detail in *Interrupts and Exceptions* on page 247.

3.7.1 Special Purpose Registers General (USPRG0, SPRG0:SPRG7)

USPRG0 and SPRG0:SPRG7 are provided for general purpose, system-dependent software use. One common system usage of these registers is as temporary storage locations. For example, a routine 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 SPRG4:SPRG7 is non-privileged for read but privileged for write, and hence different SPR numbers are used for reading than for writing.

Access to SPRG0:SPRG3 is privileged for both read and write.

Figure 3-8. Special Purpose Registers General (USPRG0, SPRG0:SPRG7)

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

3.7.2 Processor Version Register (PVR)

The PVR is a read-only register typically used to identify a specific processor core and chip implementation. Software can read the PVR to determine processor core and chip hardware features. The PVR can be read into a GPR using **mfspr** instruction.

Refer to the chip data sheet for the PVR value for a particular chip.

Access to the PVR is privileged.

Figure 3-9. Processor Version Register (PVR)

0:31		Processor Version	Refer to the chip data sheet for the PVR value for a particular chip.

3.7.3 Processor Identification Register (PIR)

The PIR is a read-only register that uniquely identifies a specific instance of a processor core, within a multi-processor configuration, enabling software to determine exactly which processor it is running on. This capability is important for operating system software within multiprocessor configurations. The PIR can be read into a GPR using **mfspr**.

Because the PPC465 is a uniprocessor, PIR[PIN] = 0b0000.

Access to the PIR is privileged.

Figure 3-10. Processor Identification Register (PIR)

0:27	▨	Reserved	
28:31	PIN	Processor Identification Number (PIN)	

3.7.4 Core Configuration Register 0 (CCR0)

The CCR0 controls a number of special chip functions, including data cache and auxiliary processor operation, speculative instruction fetching, trace, and the operation of the cache block touch instructions. The CCR0 is written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**. *Figure 3-11* illustrates the fields of the CCR0, and gives a brief description of their functions. A cross reference after the bit-field description indicates the section of this document which describes each field in more detail.

Access to the CCR0 is privileged.

Figure 3-11. Core Configuration Register 0 (CCR0)

0		Reserved	
1	PRE	Parity Recovery Enable 0 Semi-recoverable parity mode enabled for data cache 1 Fully recoverable parity mode enabled for data cache	Must be set to 1 to guarantee full recovery from MMU and data cache parity errors.
2:3		Reserved	
4	CRPE	Cache Read Parity Enable 0 Disable parity information reads 1 Enable parity information reads	When enabled, execution of **icread**, **dcread**, or **tlbre** loads parity information into the ICDBTRH, DCDBTRL, or target GPR, respectively.
5:9		Reserved	
10	DSTG	Disable Store Gathering 0 Enabled; stores to contiguous addresses may be gathered into a single transfer 1 Disabled; all stores to memory will be performed independently	See *Store Gathering* on page 142.
11	DAPUIB	Disable APU/FPU Instruction Broadcast 0 Enabled. 1 Disabled; instructions not broadcast to APU/FPU for decoding	This mechanism is provided as a means of reducing power consumption when an auxiliary processor is not attached and/or is not being used. See *Reset and Initialization* in the chip user's manual. Note: APU broadcast must be enabled to use the FPU.
12:15		Reserved	
16	DTB	Disable Trace Broadcast 0 Enabled. 1 Disabled; no trace information is broadcast.	This mechanism is provided as a means of reducing power consumption when instruction tracing is not needed. See *Reset and Initialization* in the chip user's manual.
17	GICBT	Guaranteed Instruction Cache Block Touch 0 **icbt** may be abandoned without having filled cache line if instruction pipeline stalls. 1 **icbt** is guaranteed to fill cache line even if instruction pipeline stalls.	See *icbt Operation* on page 135.
18	GDCBT	Guaranteed Data Cache Block Touch 0 **dcbt/dcbtst** may be abandoned without having filled cache line if load/store pipeline stalls. 1 **dcbt/dcbtst** are guaranteed to fill cache line even if load/store pipeline stalls.	See *Data Cache Control and Debug* on page 146.
19:22		Reserved	
23	FLSTA	Force Load/Store Alignment 0 No Alignment exception on integer storage access instructions, regardless of alignment 1 An alignment exception occurs on integer storage access instructions if data address is not on an operand boundary.	See *Load and Store Alignment* on page 140.
24:31		Reserved	

3.7.5 Core Configuration Register 1 (CCR1)

Bits 0:19 of CCR1 can cause all possible parity error exceptions to verify correct machine check exception handler operation. Other CCR1 bits can force a full-line data cache flush, or select a CPU timer clock input other than CPUClock. The CCR1 is written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**. *Figure 3-12* illustrates the fields of the CCR1, and gives a brief description of their functions. Access to the CCR1 is privileged.

Figure 3-12. Core Configuration Register 1 (CCR1)

0:7	ICDPEI	Instruction Cache Data Parity Error Insert 0 record even parity (normal) 1 record odd parity (simulate parity error)	Controls inversion of parity bits recorded when the instruction cache is filled. Each of the 8 bits corresponds to one of the instruction words in the line.
8:9	ICTPEI	Instruction Cache Tag Parity Error Insert 0 record even parity (normal) 1 record odd parity (simulate parity error)	Controls inversion of parity bits recorded for the tag field in the instruction cache.
10:11	DCTPEI	Data Cache Tag Parity Error Insert 0 record even parity (normal) 1 record odd parity (simulate parity error)	Controls inversion of parity bits recorded for the tag field in the data cache.
12	DCDPEI	Data Cache Data Parity Error Insert 0 record even parity (normal) 1 record odd parity (simulate parity error)	Controls inversion of parity bits recorded for the data field in the data cache.
13	DCUPEI	Data Cache U-bit Parity Error Insert 0 record even parity (normal) 1 record odd parity (simulate parity error)	Controls inversion of parity bit recorded for the U fields in the data cache.
14	DCMPEI	Data Cache Modified-bit Parity Error Insert 0 record even parity (normal) 1 record odd parity (simulate parity error)	Controls inversion of parity bits recorded for the modified (dirty) field in the data cache.
15	FCOM	Force Cache Operation Miss 0 normal operation 1 cache ops appear to miss the cache	Force **icbt**, **dcbt**, **dcbtst**, **dcbst**, **dcbf**, **dcbi**, and **dcbz** to appear to miss the caches. The intended use is with **icbt** and **dcbt** only, which will fill a duplicate line and allow testing of multi-hit parity errors. See *Section 6.2.3.7 Simulating Instruction Cache Parity Errors for Software Testing* on page 138 and *Figure 6.3.4.4* on page 155.
16:19	MMUPEI	Memory Management Unit Parity Error Insert 0 record even parity (normal) 1 record odd parity (simulate parity error)	Controls inversion of parity bits recorded for the tag field in the MMU.
20	FFF	Force Full-line Flush 0 flush only as much data as necessary. 1 always flush entire cache lines	When flushing 32-byte (8-word) lines from the data cache, normal operation is to write nothing, a double word, quad word, or the entire 8-word block to the memory as required by the dirty bits. This bit ensures that none or all dirty bits are set so that either nothing or the entire 8-word block is written to memory when flushing a line from the data cache. Refer to *Section 6.3.1.4 Line Flush Operations* on page 143.
21:23		Reserved	
24	TCS	Timer Clock Select 0 CPU timer advances by one at each rising edge of the CPU input clock (CPUCoreClk). 1 CPU timer advances by one for each rising edge of the CPU timer clock (TmrClk).	When TCS = 1,TmrClk input can toggle at up to half of the CPU clock frequency. Note: TmrClk is an external input signal. Not all designs have TmrClk input.
25	L2COBE	L2 Cache OP Broadcast Enable 0 L2 Cache OP disabled 1 L2 Cache OP enabled	The execution of cache ops in the CPU core signal the L2 via the I-side read, and D-side read and write interfaces to signal the L2C/PLB logic.
26:31		Reserved	

3.7.6 Reset Configuration (RSTCFG)

The read-only RSTCFG register reports the values of certain fields of TLB as supplied at reset.

Access to RSTCFG is privileged.

Figure 3-13. Reset Configuration (RSTCFG)

0:15		Reserved	
16	U0	U0 Storage Attribute 0 U0 storage attribute is disabled 1 U0 storage attribute is enabled	U0 has no effect in the PPC465.
17	U1	U1 Storage Attribute 0 Memory page contains normal instructions and data 1 Memory page contains transient instructions or data	
18	U2	U2 Storage Attribute 0 A storage miss does not cause a line to be allocated in the data cache 1 A storage miss causes a line to be allocated in the data cache	
19	U3	U3 Storage Attribute 0 U3 storage attribute is disabled 1 U3 storage attribute is enabled	U3 has no effect in the PPC465.
20:23		Reserved	
24	E	E Storage Attribute 0 Accesses to the page are big endian. 1 Accesses to the page are little endian.	
25:27		Reserved	
28:31	ERPN	Extended Real Page Number	The ERPN is configured to point to the boot device on reset. The ERPN when booting from the EBC (NOR flash) or NDFC (NAND flash) is 0b0100. The reset vector is 0x4FFFFFFFC.

3.7.7 Device Control Register Immediate Prefix Register (DCRIPR)

The DCRIPR provides the upper order 22 bits of a DCR address to be used by the **mfdcr**, **mfdcr[u]x**, **mtdcr**, and **mtdcr[u]x** instructions. This SPR has hex address 0x37B, can be read and written, and is privileged. It is implementation dependent; it is not part of the Book-E Architecture specification. This register is specific to the PPC440H6, PPC464 and PPC465 processor cores.

The DCRIPR must be set to 0 when configured. All DCR devices on these parts use10-bit addressing. Since DCRIPR defaults to 0 after reset, software is not required to initialize this register to 0.

Figure 3-14. Device Control Register Immediate Prefix Register (DCRIPR)		
0:21	UOA	Upper order address Implementation-specific. Use for the upper order address bits for DCR address.
22:31	Reserved	Reserved

3.8 User and Supervisor Modes

PowerPC Book-E architecture defines two operating “states” or “modes,” supervisor (privileged), and user (non-privileged). Which mode the processor is operating in 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 an historical alternative name for user mode, which is “problem state.” Hence the value 1 in the field indicates “problem state,” and not “privileged” as one might expect.

3.8.1 Privileged Instructions

The following instructions are privileged and cannot be executed in user mode:

Table 3-27. Privileged Instructions

dcbi	
dccci	
dcread	
iccci	
icread	
mfdcr	
mfdcrx	
mfdcrux	
mfmsr	
mfspr	For any SPR Number with SPRN_5 = 1. See *Privileged SPRs* on page 78.
mtdcr	
mtdcrx	
mtdcrux	

Table 3-27. Privileged Instructions (continued)

mtmsr	
mtspr	For any SPR Number with $SPRN_5$ = 1. See *Privileged SPRs* on page 78.
rfci	
rfi	
rfmci	
tlbre	
tlbsx	
tlbsync	
tlbwe	
wrtee	
wrteei	

3.8.2 Privileged SPRs

Most SPRs are privileged. The only defined non-privileged SPRs are the LR, CTR, XER, USPRG0, SPRG4–7 (read access only), TBU (read access only), and TBL (read access only). The PPC465 also treats all SPR numbers with a 1 in bit 5 of the SPRN field as privileged, whether the particular SPR number is defined or not. Thus the processor core causes a Privileged Instruction exception type Program interrupt on any attempt to access such an SPR number while in user mode. In addition, the processor core causes an Illegal Instruction exception type Program interrupt on any attempt to access while in user mode an undefined SPR number with a 0 in $SPRN_5$. On the other hand, the result of attempting to access an undefined SPR number in supervisor mode is undefined, regardless of the value in $SPRN_5$.

3.8.3 Privileged/Non-Privileged DCRs

In order to support the new mtdcr[u]x and mfdcr[u]x instructions, the DCR interface adds an one output pin for the privileged/non-privileged indicator. Note that privileged signal indicates which type of opcode caused the DCR operation to appear on the DCR interface, and is not directly related to the MSR.PR bit. Privileged (a.k.a. supervisor-mode) code may execute any of the six DCR opcodes, and hence may produce DCR operations on the interface with either value indicated on the privileged signal. Non-privileged (user-mode) code only generates DCR traffic with a non-privileged indication on the interface. If user-mode code attempts to execute a privileged opcode, an exception is signalled due to the privilege violation.

3.9 Speculative Accesses

The PowerPC Book-E 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 that is not required by the sequential execution model (SEM).

For example, the PPC465 speculatively prefetches instructions down the predicted path of a conditional branch; if the branch is later determined to not go in the predicted direction, the fetching of the instructions from the predicted path is not required by the SEM and thus is speculative. Similarly, the PPC465 executes load instructions out-of-order, and may read data from memory for a load instruction that is past an undetermined branch.

Sometimes speculative accesses are inappropriate, however. For example, attempting to access data at addresses to which I/O devices are mapped can cause problems. If the I/O device is a serial port, reading it speculatively could cause data to be lost.

The architecture provides two mechanisms for protecting against errant accesses to such “non-well-behaved” memory addresses. The first is the guarded (G) storage attribute, and protects against speculative data accesses. The second is the execute permission mechanism, and protects against speculative instruction fetches. Both of these mechanisms are described in *Memory Management* on page 219

3.10 Synchronization

The PPC465 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.

3.10.1 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 look aside 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**, **rfci**, and **rfmci** 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 *Interrupts and Exceptions* on page 247).
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 80).

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 PPC465, this can only occur with Data Machine Check exceptions, and not Instruction Machine Check exceptions.

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

1. Consider the following self-modifying code instruction sequence:

   ```
   stw XYZ   Store to caching inhibited address XYZ
   isync
   XYZ       fetch and execute the instruction at address XYZ
   ```

   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. Now consider the required self-modifying code sequence:

stw	Write new instruction to data cache
dcbst	Push the new instruction from the data cache to memory
msync	Guarantee that **dcbst** completes before subsequent instructions begin
icbi	Invalidate old copy of instruction in instruction cache
msync	Guarantee that **icbi** completes before subsequent instructions begin
isync	Force context synchronization, discarded instructions and re-fetch, fetch of stored instruction guaranteed to get new value

3. This example illustrates the use of **isync** with context changes to the debug facilities

mtdbcr0	Enable the instruction address compare (IAC) debug 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** is necessary to guarantee that the IAC event is recognized on the execution of this instruction; without the **isync**, the XYZ instruction may be prefetched and dispatched to execution before recognizing that the IAC event has been enabled.

4. The last example is the use of isync to access DCRs with mtdcr or mfdcr instructions based on DCRIPR register:

mtspr DCRIPR	set up DCRIPR value for DCRN
isync	ensures new DCRN by context synchronization
mtdcr	access new DCR with new value

3.10.2 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. An example of a scenario requiring execution synchronization would be just before the execution of a TLB-updating instructions (such as **tlbwe**). An execution synchronizing instruction should be executed to guarantee that all preceding storage access instructions have performed their address translations before executing **tlbwe** to invalidate an entry which might be used by those preceding instructions.

There are four execution synchronizing instructions: **mtmsr, wrtee, wrteei,** and **msync**. Of course, all context synchronizing instruction are also implicitly execution synchronizing, since context synchronization is a superset of execution synchronization.

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].

3.10.3 Storage Ordering and Synchronization

Storage synchronization enforces ordering between storage access instructions executed by the PPC465. There are two storage synchronizing instructions: **msync** and **mbar**. PowerPC Book-E architecture defines different ordering requirements for these two instructions, but the PPC465 implements them in an identical fashion. Architecturally, **msync** is the “stronger” of the two, and is also execution synchronizing, whereas **mbar** is not.

The instruction **mbar** acts as a “barrier” between all storage access instructions executed before the **mbar** and all those executed after the **mbar**. That is, **mbar** ensures that all of the storage accesses initiated by instructions before the **mbar** are performed with respect to the memory subsystem before any of the accesses initiated by instructions after the **mbar**. However, **mbar** does not prevent subsequent instructions from executing (nor even from completing) before the completion of the storage accesses initiated by instructions before the **mbar**.

msync, on the other hand, does guarantee that all preceding storage accesses have actually been performed with respect to the memory subsystem before the execution of any instruction after the **msync**. Note that this requirement goes beyond the requirements of mere execution synchronization, in that execution synchronization doesn’t require the completion of preceding storage accesses.

The following two examples illustrate the distinctive use of **mbar** vs. **msync**.

stw	Store data to an I/O device
msync	Wait for store to actually complete
mtdcr	Reconfigure the I/O device

In this example, the **mtdcr** is reconfiguring the I/O device in a manner which would cause the preceding store instruction to fail, were the **mtdcr** to change the device before the completion of the store. Since **mtdcr** is not a storage access instruction, the use of **mbar** instead of **msync** would not guarantee that the store is performed before letting the **mtdcr** reconfigure the device. It only guarantees that subsequent storage accesses are not performed to memory or any device before the earlier store.

Now consider this next example:

stb X	Store data to an I/O device at address X, causing a status bit at address Y to be reset
mbar	Guarantee preceding store is performed to the device before any subsequent storage accesses are performed
lbz Y	Load status from the I/O device at address Y

Here, **mbar** is appropriate instead of **msync**, because all that is required is that the store to the I/O device happens before the load does, but not that other instructions subsequent to the **mbar** won’t get executed before the store.

4. FPU Programming Model

The PPC465 has a built-in super scalar FPU that supports both single- and double-precision operations, and offers single cycle through put on most instructions.

Features include:
- Five stages with 2 MFlops/MHz
- Hardware support for IEEE 754
- Single- and double-precision
- Single-cycle throughput on most instructions
- Thirty-two 64-bit floating point registers

The programming model of the PPC465-S FPU describes how the following features and operations appear to programmers:

- Storage addressing (including storage operands, effective address calculation, and data storage addressing modes), starting on page 83
- Floating-point exceptions, starting on page 85
- Floating-point registers, starting on page 86
- Floating-point data formats, starting on page 89
- Floating-point execution models, starting on page 96
- Floating-point instructions, starting on page 99

The Book-E Enhanced PowerPC Architecture (referred to as Book-E) specifies that the floating-point unit (FPU) implements a floating-point system as defined in ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic* (referred to as IEEE 754), but the architecture requires software support to conform fully with the standard. IEEE 754 defines certain required “operations” (addition, subtraction, and so on); the term “floating-point operation” is used to refer to one of these required operations, or to the operation performed by one of the *Multiply-Add* or *Reciprocal Estimate* instructions. All floating-point operations conform to the IEEE standard, unless software sets the IEEE Mode (NI) bit to 1 in the Floating-Point Status and Control Register (FPSCR). When FPSCR[NI] = 1, floating-point operations do not necessarily conform to the IEEE standard.

Important: Before using the FPU, it must be enabled and configured in the processor MSR and CCR0 registers. Specifically, set MSR[FP] = 1 and CCR0[DAPUIB] = 0. Also, MSR[FE0, FE1] must be set as desired. Refer to the MSR and CCR0 registers in the *PPC440H6 Processor User’s Manual* for details.

4.1 Storage Addressing

The PPC465-S FPU accesses storage in the same uniform 32-bit (4GB) effective address (EA) space as the PPC465-S processor. Effective addresses are expanded into virtual addresses and then translated to 33-bit (8GB) real addresses by the memory management unit (MMU) of the processor.

The PPC465-S FPU generates an effective address whenever it executes a *Load/Store* instruction.

4.1.1 Storage Operands

Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

The data storage operands accessed by the PPC465-S FPU load/store instructions can be words (4 bytes, or 32 bits) or double words (8 bytes, or 64 bits). The address of a storage operand is the address of its first byte (that is, of its lowest-numbered byte). Byte ordering can be either big endian or little endian, as controlled by the endian (E) storage attribute.

Operand length is implicit for each scalar storage access instruction. The operand of such a scalar storage access instruction has a “natural” alignment boundary equal to the operand length. In other words, the “natural” address of an operand is an integral multiple of the operand length. A storage operand is said to be *aligned* if it is aligned at its natural boundary; otherwise, it is said to be *unaligned*.

Data storage operands for storage access instructions have the following characteristics.

Table 4-1. Data Operand Definitions

Storage Access Instruction Type	Operand Length	$A_{28:31}$ if aligned
Word	4 bytes	0bxx00
Doubleword	8 bytes	0bx000
Note: An “x” in an address bit position indicates that the bit can be 0 or 1 regardless of the state of other bits in the address.		

The alignment of the operand effective address of some storage access instructions can affect performance, and in some cases can cause an Alignment exception to occur. For such storage access instructions, the best performance is obtained when the storage operands are naturally aligned. *Table 4-2* summarizes the effects of alignment on those storage access instruction types for which such effects exist. If an instruction type is not shown in the table, then there are no alignment effects for that instruction type.

Table 4-2. Alignment Effects for Storage Access Instructions

Storage Access Instruction Type	Alignment Effect
FP Load/Store Word	Alignment exception if the storage crosses a 16-byte boundary ($EA_{28:31}$ = 0b1100); otherwise, no effect
FP Load/Store Doubleword	Alignment exception if the storage crosses 16-byte boundary ($EA_{28:31}$ > 0b1000); otherwise no effect

Instruction storage operands, on the other hand, are a word, and the effective addresses calculated by branch instructions are therefore always word-aligned.

4.1.2 Effective Address Calculation

For a storage access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address of $2^{32} – 1$ (that is, the storage operand itself crosses the maximum address boundary), the result of the operation is undefined, as specified by the architecture. The processor performs the operation as if the storage operand wrapped around from the maximum effective address to effective address 0. Software, however, should not depend upon this behavior, so that can be ported to other implementations that do not handle such accesses in the same manner. Software should ensure that no data storage operands cross the maximum address boundary.

Note that because instructions are words, and because the effective addresses of instructions are always implicitly on word boundaries, an instruction storage operand cannot cross any word boundary, including the maximum address boundary.

Effective address arithmetic, which calculates the starting address for storage operands, wraps around from the maximum address to address 0, for all effective address computations except next sequential instruction fetching.

4.1.3 Data Storage Addressing Modes

The PPC465-S FPU supports the following data storage addressing modes.

- Base + displacement (D-mode) addressing mode:

  The 16-bit D field is sign-extended to 32 bits and added to the contents of the GPR designated by RA, or to zero if RA = 0. The low-order 32 bits of the sum form the effective address of the data storage operand.
- Base + index (X-mode) addressing mode:

  The contents of the GPR designated by RB (or the value 0 for **lswi** and **stswi**) are added to the contents of the GPR designated by RA, or to zero if RA = 0; the low-order 32 bits of the sum form the effective address of the data storage operand.

4.2 Floating-Point Exceptions

Each floating-point exception, and each category of Invalid Operation Exception, is associated with an exception bit in the FPSCR. The following floating-point exceptions are detected by the processor; the associated FPSCR fields are listed with each exception and Invalid Operation exception category:

Table 4-3. Invalid Operation Exception Categories

Category	FPSCR Field
SNaN	VXSNAN
Infinity – Infinity	VXISI
Infinity ÷ Infinity	VXIDI
Zero ÷ Zero	VXZDZ
Infinity × Zero	VXIMZ
Invalid Compare	VXVC
Software Request	VXSOFT
Invalid Square Root	VXSQRT
Invalid Integer Convert	VXCVI

- Invalid Operation Exception (VX)
- Zero Divide Exception (ZX)
- Overflow Exception (OX)
- Underflow Exception (UX)
- Inexact Exception (XI)

Each floating-point exception also has a corresponding enable bit in the FPSCR. See *Floating-Point Status and Control Register Instructions* on page 106 for descriptions of these exception and enable bits, and *Floating Point Unit Interrupts and Exceptions* on page 293 for a detailed discussion of floating-point exceptions, including the effects of the FPSCR enable bits.

4.3 Floating-Point Registers

This section provides an overview of the register types implemented in the PPC465-S FPU. Detailed descriptions of the floating-point registers are provided within the chapters covering the functions with which they are associated.

Certain bits in some registers are *reserved* and thus not necessarily implemented. For all registers with fields marked as reserved, these reserved fields should be written as 0 and read as *undefined*. The recommended coding practice is to perform the initial write to a register with reserved fields set to 0, 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.

Each register is classified as being of a particular *type*, as characterized by the specific instructions used to read and write registers of that type. The registers contained within the PPC465-S are defined by Book-E, except for Device Control Registers (DCRs) that are implementation-specific and unique to the PPC465-S FPU.

4.3.1 Register Types

The PPC465-S floating point unit provides three types of registers, Floating Point Registers (FPRs), the FPSCR, and DCRs. Each type is characterized by the instructions used to read and write the registers. The following subsections provide an overview of each register type and the instructions associated with them.

4.3.1.1 Floating-Point Registers (FPR0:31)

The PPC465-S FPU provides 32 Floating-Point Registers (FPRs), each 64 bits wide. In any cycle, the FPR file can read the operands for a store instruction and an arithmetic instruction, or write the data from a load instruction and the result of an arithmetic instruction.

Figure 4-1. Floating-Point Registers (FPR0:31)

0:63	FPRD	Floating-Point Register data	

The FPRs are numbered FPR0:FPR31. The floating-point instruction formats provide 5-bit fields to specify the FPRs used as operands in the execution of the associated instructions.

Each FPR contains 64 bits that support the floating-point double format. All instructions that interpret the contents of an FPR as a floating-point value uses the floating-point double format for this interpretation.

The computational instructions, and the *Move* and *Select* instructions, operate on data located in FPRs and, with the exception of the *Compare* instructions, place the result value into a FPR and optionally place status information into the Condition Register (CR).

Load and store double instructions are provided that transfer 64 bits of data between storage and the FPRs with no conversion. *Load Single* instructions transfer and convert floating-point values in floating-point single format from storage to the same value in floating-point double format in the FPRs. Store single instructions are provided to transfer and convert floating-point values in floating-point double format from the FPRs to the same value in floating-point single format in storage.

Some floating-point instructions update the FPSCR and CR explicitly. Some of these instructions move data to and from an FPR to the FPSCR, or from the FPSCR to an FPR.

The computational instructions and the *Select* instruction accept values from the FPRs in double format. For single-precision arithmetic instructions, all input values must be representable in single format;. if not, the result placed into the target FPR, and the setting of status bits in the FPSCR are undefined.

4.3.1.2 Floating-Point Status and Control Register (FPSCR)

The FPSCR controls the handling of floating-point exceptions and records status resulting from the floating-point operations. See *Floating-Point Status and Control Register* on page 303 for a more detailed description of the FPSCR.

Figure 4-2. Floating-Point Status and Control Register (FPSCR)

0	FX	Floating-Point Exception Summary 0 No FPSCR exception bits changed from 0 to 1. 1 At least one FPSCR exception bit changed from 0 to 1.	All floating-point instructions, except **mtfsfi** and **mtfsf**, implicitly set this field to 1 if the instruction causes any floating-point exception bits in the FPSCR to change from 0 to 1. **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0,** and **mtfsb1** can alter this field explicitly.
1	FEX	Floating-Point Enabled Exception Summary	The OR of all the floating-point exception fields masked by their respective enable fields. **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, and **mtfsb1** cannot alter this field explicitly.
2	VX	Floating-Point Invalid Operation Exception Summary	The OR of all the Invalid Operation exception fields. **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, and **mtfsb1** cannot alter this field explicitly.
3	OX	Floating-Point Overflow Exception 0 A Floating-Point Overflow exception did not occur. 1 A Floating-Point Overflow exception occurred.	See *Overflow Exception* on page 299
4	UX	Floating-Point Underflow Exception 0 A Floating-Point Underflow exception did not occur. 1 A Floating-Point Underflow exception occurred.	See *Underflow Exception* on page 300
5	ZX	Floating-Point Zero Divide Exception 0 A Floating-Point Zero Divide exception did not occur. 1 A Floating-Point Zero Divide exception occurred.	See *Zero Divide Exception* on page 299
6	IX	Floating-Point Inexact Exception 0 A Floating-Point Inexact exception did not occur. 1 A Floating-Point Inexact exception occurred.	This field is a sticky version of FPSCR[FI] The following rules describe how a given instruction sets this field. If the instruction affects FPSCR[FI], the new value of this field is obtained by ORing the old value of this field with the new value of FPSCR[FI]. If the instruction does not affect FPSCR[FI], the value of this field is unchanged.
7	VXSNAN	Floating-Point Invalid Operation Exception (SNaN) 0 A Floating-Point Invalid Operation exception (VXSNAN) did not occur. 1 A Floating-Point Invalid Operation exception (VXSNAN) occurred.	See *Invalid Operation Exception* on page 297
8	VXISI	Floating-Point Invalid Operation Exception ($\infty - \infty$) 0 A Floating-Point Invalid Operation exception (VXISI) did not occur. 1 A Floating-Point Invalid Operation exception (VXISI) occurred.	See *Invalid Operation Exception* on page 297

9	VXIDI	Floating-Point Invalid Operation Exception (∞ ÷ ∞) 0 A Floating-Point Invalid Operation exception (VXIDI) did not occur. 1 A Floating-Point Invalid Operation exception (VXIDI) occurred.	See *Invalid Operation Exception* on page 297
10	VXZDZ	Floating-Point Invalid Operation Exception (0 ÷ 0) 0 A Floating-Point Invalid Operation exception (VXZDZ) did not occur. 1 A Floating-Point Invalid Operation exception (VXZDZ) occurred.	See *Invalid Operation Exception* on page 297
11	VXIMZ	Floating-Point Invalid Operation Exception (∞ × 0) 0 A Floating-Point Invalid Operation exception (VXIMZ) did not occur. 1 A Floating-Point Invalid Operation exception (VXIMZ) occurred.	See *Invalid Operation Exception* on page 297
12	VXVC	Floating-Point Invalid Operation Exception (Invalid Compare) 0 A Floating-Point Invalid Operation exception (VXVC) did not occur. 1 A Floating-Point Invalid Operation exception (VXVC) occurred.	See *Invalid Operation Exception* on page 297
13	FR	Floating-Point Fraction Rounded	The last *Arithmetic* or *Rounding and Conversion* instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See *Rounding* on page 95. This bit is not sticky.
14	FI	Floating-Point Fraction Inexact	The last *Arithmetic* or *Rounding and Conversion* instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See *Rounding* on page 95. This bit is not sticky. See the definition of FPSCR[XX] regarding the relationship between FPSCR[FI] and FPSCR[XX].
15	FPRF	Floating-Point Result Flag (FPRF)	
16	FL	Floating-Point Less Than or Negative	
17	FG	Floating-Point Greater Than or Positive	
18	FE	Floating-Point Equal to Zero	
19	FU	Floating-Point Unordered or NaN	
20		Reserved	
21	VXSOFT	Floating-Point Invalid Operation Exception (Software Request) 0 A Floating-Point Invalid Operation exception (Software Request) did not occur. 1 A Floating-Point Invalid Operation exception (Software Request) occurred.	See *Invalid Operation Exception* on page 297
22	VXSQRT	Floating-Point Invalid Operation Exception (Invalid Square Root) 0 A Floating-Point Invalid Operation exception (Invalid Square Root) did not occur. 1 A Floating-Point Invalid Operation exception (Invalid Square Root) occurred.	See *Invalid Operation Exception* on page 297
23	VXCVI	Floating-Point Invalid Operation Exception (Invalid Integer Convert) 0 A Floating-Point Invalid Operation exception (Invalid Integer Convert) did not occur. 1 A Floating-Point Invalid Operation exception (Invalid Integer Convert) occurred.	See *Invalid Operation Exception* on page 297

24	VE	Floating-Point Invalid Operation Exception Enabled 0 Floating-Point Invalid Operation exceptions are disabled. 1 Floating-Point Invalid Operation exceptions are enabled.	
25	OE	Floating-Point Overflow Exception Enable 0 Floating-Point Overflow exceptions are disabled. 1 Floating-Point Overflow exceptions are enabled.	
26	UE	Floating-Point Underflow Exception Enable 0 Floating-Point Underflow exceptions are disabled. 1 Floating-Point Underflow exceptions are enabled.	
27	ZE	Floating-Point Zero Divide Exception Enable 0 Floating-Point Zero Divide exceptions are disabled. 1 Floating-Point Zero Divide exceptions are enabled.	
28	XE	Floating-Point Inexact Exception Enable 0 Floating-Point Inexact exceptions are disabled. 1 Floating-Point Inexact exceptions are enabled.	
29	NI	Floating-Point Non-IEEE Mode 0 Non-IEEE mode is disabled 1 Non-IEEE mode is enabled.	If FPSCR[NI = 1, the remaining FPSCR bits may have meanings other than those given in this document, and the results of floating-point operations need not conform to the IEEE standard. If the IEEE-conforming result of a floating-point operation would be a denormalized number, the result of that operation is 0 (with the same sign as the denormalized number) if FPSCR[NI] = 1. The behavior when FPSCR[NI] = 1 can vary from one implementation to another
30:31	RN	Floating-Point Rounding Control 00 Round to nearest 01 Round toward zero 10 Round toward +Infinity 11 Round toward –Infinity	See *Rounding* on page 95.

Programming Note: Setting FPSCR[NI] = 1 is intended to permit results to be approximate and to cause performance to be more predictable and less data-dependent than when FPSCR[NI] = 0. For example, in non-IEEE mode, 0 is returned instead of a denormalized number, and non-IEEE mode may return a large number instead of an infinity. In non-IEEE mode, an implementation should provide a means for ensuring that all results are produced without software assistance (that is, without causing an Enabled exception type Program interrupt or a Floating-Point Unimplemented Instruction exception type Program interrupt, and without invoking an “emulation assist.” See *Floating Point Unit Interrupts and Exceptions* on page 293 The means may be controlled by one or more other FPSCR bits (recall that the other FPSCR bits have implementation-dependent meanings when FPSCR[NI] = 1).

4.4 Floating-Point Data Formats

Floating-point values are represented in two binary fixed-length formats. Single-precision values are represented in the 32-bit single format. Double-precision values are represented in the 64-bit double format. The single format can be used for data in storage, but cannot be stored in the FPRs. The double format can be used for data in storage and for data in the FPRs. When a floating-point value is loaded from storage using a *Load Single* instruction, it is converted to double format and placed in the target FPR. Conversely, a floating-point value stored from an FPR into storage using a Store Single instruction is converted to single format before being placed in storage.

The lengths of the exponent and the fraction fields differ between these two formats. The structure of the single and double formats are shown in *Table 4-4* and *Table 4-5*, respectively.

Table 4-4. Floating-Point Single Format

S	EXP	Fraction	
0	1	9	31

Table 4-5. Floating-Point Double Format

S	EXP	Fraction	
0	1	12	63

Values in floating-point format are composed of three fields:

Table 4-6. Format Fields

Field	Description
S	Sign Bit
EXP	Exponent + bias
FRACTION	Fraction

If only a portion of a floating-point data item in storage is accessed, such as with a load or store instruction for a byte or half word (or word in the case of floating-point double format), the value affected depends on whether the PowerPC Embedded system is operating with big endian or little endian byte ordering.

4.4.1 Value Representation

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized numbers and is located in the unit bit position (that is, the first bit to the left of the binary point). Values representable within the two floating-point formats can be specified by the parameters listed in *Table 4-7*.

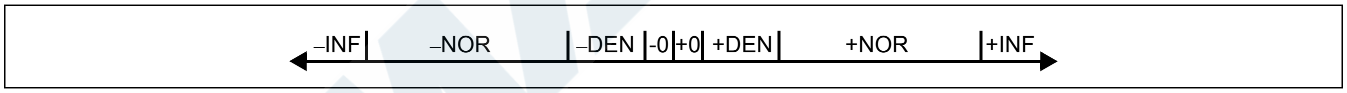
Table 4-7. IEEE 754 Floating-Point Fields

	Single	Double
Exponent Bias	+127	+1023
Maximum Exponent	+127	+1023
Minimum Exponent	–126	–1022
Field Widths (Bits)		
Sign	1	1
Exponent	8	11
Fraction	23	52
Significand	24	53

The FPRs support the floating-point double format only.

The numeric and nonnumeric values representable within each of the two supported formats are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The nonnumeric values representable are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible, however. to define restricted operations among numbers and infinities. The relative location on the real number line for each of the defined entities is shown in *Figure 4-3*.

Figure 4-3. Approximation to Real Numbers



The NaNs are not related to the numeric values or infinities by order or value, but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

4.4.2 Binary Floating-Point Numbers

Machine-representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

4.4.2.1 Normalized Numbers

Normalized numbers (±NOR) have an unbiased exponent value in the range:

- –126 to 127 in single format
- –1022 to 1023 in double format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

$$\text{NOR} = (-1)^{s} \times 2^{E} \times (1.\text{fraction})$$

where s is the sign, E is the unbiased exponent, and 1.fraction is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

The ranges covered by the magnitude (M) of a normalized floating-point number are approximately equal to:

- Single Format:

  $1.2 \times 10^{-38} \leq M \leq 3.4 \times 10^{38}$

- Double Format:

  $2.2 \times 10^{-308} \leq M \leq 1.8 \times 10^{308}$

4.4.2.2 Denormalized Numbers

Denormalized numbers (±DEN) are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as follows:

$\text{DEN} = (-1)^s \times 2^{\text{Emin}} \times (0.\text{fraction})$

where Emin is the minimum representable exponent value (–126 for single-precision, –1022 for double-precision).

4.4.2.3 Zero Values

Zero values (±0) have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations; comparison treats +0 as equal to –0).

4.4.3 Infinities

Infinities ($\pm\infty$)are values that have the maximum biased exponent value:

- 255 in single format
- 2047 in double format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

$-\infty < \text{every finite number} < +\infty$

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in *Invalid Operation Exception* on page 297.

4.4.3.1 Not a Numbers

Not a Numbers (NaNs) are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored, that is, NaNs are neither positive nor negative. If the high-order bit of the fraction field is 0, the NaN is a Signalling NaN (SNaN); otherwise it is a Quiet NaN (QNaN).

Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions.

Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation Exception is disabled (FPSCR[VE] = 0). Quiet NaNs propagate through all floating-point instructions except **fcmpo**, **frsp**, and **fctiw**. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a disabled Invalid Operation exception, the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

```
if FPR(FRA) is a NaN
then FPR(FRT) ← FPR(FRA)
else if FPR(FRB) is a NaN
then if instruction is frsp
          then FPR(FRT) ← $FPR(FRB)_{0:34} \| {}^{29}0$
          else FPR(FRT) ← FPR(FRB)
else if FPR(FRC) is a NaN
          then FPR(FRT) ← FPR(FRC)
          else if generated QNaN
                    then FPR(FRT) ← generated QNaN
```

If the operand specified by FRA is a NaN, that NaN is stored as the result. Otherwise, if the operand specified by FRB is a NaN (if the instruction specifies an FRB operand), that NaN is stored as the result, with the low-order 29 bits of the result set to 0 if the instruction is **frsp**. Otherwise, if the operand specified by FRC is a NaN (if the instruction specifies an FRC operand), that NaN is stored as the result. Otherwise, if a QNaN was generated due to a disabled Invalid Operation Exception, that QNaN is stored as the result. If a QNaN is to be generated as a result, the QNaN generated has a sign bit of 0, an exponent field of all 1s, and a high-order fraction bit of 1 with all other fraction bits 0. Any instruction that generates a QNaN as the result of a disabled Invalid Operation must generate this QNaN (that is, 0x7FF8000000000000).

A double-precision NaN is representable in single format if and only if the low-order 29 bits of the double-precision NaNs fraction are zero.

4.4.4 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same sign, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation $x - y$ is the same as the sign of the result of the add operation $x + (-y)$.

When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except Round toward -Infinity, in which mode the sign is negative.

- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.
- The sign of the result of a *Square Root* or **frsqrte** instruction is always positive, except that the square root of –0 is –0 and the reciprocal square root of –0 is –Infinity.
- The sign of the result of an **frsp[.]**, or **fctiw** operation is the sign of the operand being converted.

For the *Multiply-Add* instructions, the preceding rules are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

4.4.5 Normalization and Denormalization

The intermediate result of an arithmetic or **frsp** instruction may require normalization and/or denormalization. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or **frsp** instruction produces an intermediate result, consisting of a sign bit, an exponent, and a nonzero significand with a 0 leading bit, it is not a normalized number and must be normalized before it is stored.

A number is normalized by shifting its significand left while decrementing its exponent by 1 for each bit shifted, until the leading significand bit becomes 1. The G bit and the R bit (see *Execution Model for IEEE Operations* on page 97) participate in the shift with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result may have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be “Tiny” and the stored result is determined by the rules described in *Underflow Exception* on page 300. These rules may require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format's minimum value. If any significant bits are lost in this shifting process, “Loss of Accuracy” has occurred (see *Underflow Exception* on page 300) and an Underflow Exception is signaled.

4.4.6 Data Handling and Precision

Instructions are defined to move floating-point data between the FPRs and storage. For double format data, the data are not altered during the move. For single format data, a format conversion from single to double is performed when loading from storage into an FPR. A format conversion from double to single is performed when storing from an FPR to storage. The *Load/Store* instructions do not cause floating-point exceptions.

All computational, *Move*, and **fsel** instructions use the floating-point double format.

Floating-point single-precision values are obtained with the following types of instruction.

- Load Floating-Point Single

  This form of instruction accesses a single-precision operand in single format in storage, converts it to double format, and loads it into an FPR. No floating-point exceptions are caused by these instructions.

- Round to Floating-Point Single-Precision

  The **frsp** instruction rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into an FPR as a double-precision operand. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of the **frsp** instruction, this operation does not alter the value.

Programming Note: The **frsp** instruction enables value conversion from double-precision to single-precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions) to single-precision values before storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by an **frsp** instruction.

- Single-Precision Arithmetic Instructions

  This form of instruction takes operands from the FPRs in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single format. Status bits in the FPSCR are set to reflect the single-precision result. The result is then converted to double format and placed into an FPR. The result lies in the range supported by the single format.

All input values must be representable in single format. If they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR, are undefined.

- Store Floating-Point Single

  This form of instruction converts a double-precision operand to single format and stores that operand into storage. No floating-point exceptions are caused by these instructions. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding three types.)

When the result of a *Load Floating-Point Single*, **frsp**, or single-precision arithmetic instruction is stored in an FPR, the low-order 29 fraction bits are zero.

Programming Note: A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value is representable in single format.

4.4.7 Rounding

Rounding applies to operations that have numeric operands (operands that are not infinities or NaNs). Rounding the intermediate result of such operations may cause an Overflow Exception, an Underflow Exception, or an Inexact Exception. The following description assumes that the operations cause no exceptions and that the result is numeric. See *Value Representation* on page 90 and *Floating Point Unit Interrupts and Exceptions* on page 293 for the cases not covered here.

Execution Model for IEEE Operations on page 97 provides a detailed explanation of rounding.

The *Arithmetic* and *Rounding and Conversion* instructions produce intermediate results that can be regarded as having infinite precision and unbounded exponent range. Such intermediate results are normalized or denormalized if required, then rounded to the target format. The final result is then placed into the target FPR in double format or in integer format, depending on the instruction.

The *Arithmetic* and *Rounding and Conversion* instructions, which round intermediate results, set FPSCR[FR, FI]. If the fraction was incremented during rounding, FPSCR[FR] = 1; otherwise, FPSCR[FR] = 0. If the rounded result is inexact, FPSCR[FI] = 1; otherwise, FPSCR[FI] = 0.

The *Estimate* instructions set FPSCR[FR, FI] to undefined values. The remaining floating-point instructions do not alter FPSCR[FR, FI].

FPSCR[RN] specifies one of four programmable rounding modes.

Let z be the intermediate arithmetic result or the operand of a convert operation. If z can be represented exactly in the target format, then the result in all rounding modes is z as represented in the target format. If z cannot be represented exactly in the target format, let z1 and z2 bound z as the next larger and next smaller numbers representable in the target format. Then, z1 or z2 can be used to approximate the result in the target format.

Figure 4-4 shows the relation of z, z1, and z2 in this case. The following rules specify the rounding in the four modes. The abbreviation lsb means least-significant bit.

Figure 4-4. Selection of z1 and z2

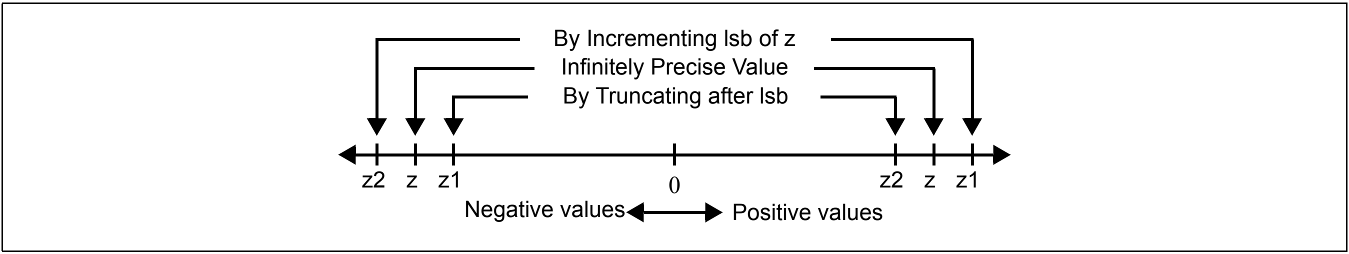


Table 4-8 describes the rounding modes.

Table 4-8. Rounding Modes

FPSCR[RN]	Rounding Mode	Description
00	Round to Nearest	Choose the value that is closest to *z*, either z1 or z2. In case of a tie, choose the one that is even (the lsb is 0).
01	Round toward Zero	Choose the smaller in magnitude (z1 or *z2*).
10	Round toward +Infinity	Choose z1.
11	Round toward –Infinity	Choose z2.

4.5 Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (i.e., operands and result that are not infinities or NaNs), and that cause no exceptions. See *Value Representation* on page 90 and *Floating Point Unit Interrupts and Exceptions* on page 293 for the cases not covered here.

Although the double format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

- Underflow during multiplication using a denormalized operand.
- Overflow during division using a denormalized divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision floating-point operations to have either (or both) single-precision or double-precision operands, but states that single-precision floating-point operations should not accept double-precision operands. Book-E follows these guidelines: double-precision arithmetic instructions can have operands of either or both precisions, while single-precision arithmetic instructions require all operands to be single-precision. Double-precision arithmetic instructions produce double-precision values, while single-precision arithmetic instructions produce single-precision values.

For arithmetic instructions, conversions from double-precision to single-precision must be done explicitly by software, while conversions from single-precision to double-precision are done implicitly.

4.5.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the FRACTION is a 23-bit field, and the single-precision Guard, Round, and Sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field.

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:55 comprise the significand of the intermediate result.

Table 4-9. IEEE 64-bit Execution Model

S	C	L	FRACTION	G	R	X
		0	1 … 52			55

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

The FRACTION is a 52-bit field that accepts the fraction of the operand.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for post-normalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the R bit, due either to shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. *Table 4-10* shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number next lower in magnitude (NL), and the representable number next higher in magnitude (NH).

Table 4-10. Interpretation of the G, R, and X Bits

G	R	X	Interpretation
0	0	0	IR is exact
0 0 0	0 1 1	1 0 1	IR closer to NL
1	0	0	IR midway between NL and NH
1 1 1	0 1 1	1 0 1	IR closer to NH

After normalization, the intermediate result is rounded, using the rounding mode specified by FPSCR[RN]. If rounding results in a carry into C, the significand is shifted right one position and the exponent incremented by one. This yields an inexact result and possibly also exponent overflow. Fraction bits to the left of the bit position used for rounding are stored into the FPR and low-order bit positions, if any, are set to zero.

Four user-selectable rounding modes are provided through FPSCR[RN] as described in *Rounding* on page 95. For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. *Table 4-11* shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the IEEE execution model.

Table 4-11. Location of the Guard, Round, and Sticky Bits in the IEEE Execution Model

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	OR of 26:52, G, R, X

Rounding can be treated as though the significand were shifted right, if required, until the least significant bit to be retained is in the low-order bit position of the FRACTION. If any of the Guard, Round, or Sticky bits is nonzero, then the result is inexact.

z1 and z2, as defined in *Rounding* on page 95, can be used to approximate the result in the target format when one of the following rules is used.

- Round to Nearest
  - Guard bit = 0

    The result is truncated. (Result exact (GRX = 000) or closest to next lower value in magnitude (GRX = 001, 010, or 011))
  - Guard bit = 1

    Depends on Round and Sticky bits:
    - Case a

      If the Round or Sticky bit is 1 (inclusive), the result is incremented. (Result closest to next higher value in magnitude (GRX = 101, 110, or 111))
    - Case b

      If the Round and Sticky bits are 0 (result midway between closest representable values), then if the low-order bit of the result is 1 the result is incremented. Otherwise (the low-order bit of the result is 0) the result is truncated (this is the case of a tie rounded to even).
- Round toward Zero

  Choose the smaller in magnitude of z1 or z2. If the Guard, Round, or Sticky bit is nonzero, the result is inexact.
- Round toward +Infinity

  Choose z1.
- Round toward –Infinity

  Choose z2.

Where the result is to have fewer than 53 bits of precision because the instruction is a *Floating Round to Single-Precision* or single-precision arithmetic instruction, the intermediate result is either normalized or placed in correct denormalized form before being rounded.

4.5.2 Execution Model for Multiply-Add Type Instructions

The PPC465-S FPU provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:106 comprise the significand of the intermediate result.

Table 4-12. Multiply-Add 64-bit Execution Model

S	C	L	FRACTION	X’
		0	1 … 105	10 6

The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), then the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the FRACTION and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the FRACTION) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount that is added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the X' bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. *Table 4-13* shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

Table 4-13. Location of Guard, Round, and Sticky Bits in the Multiply-Add Execution Model

Format	Guard	Round	Sticky
Double	53	54	OR of 55:105, X'
Single	24	25	OR of 26:105, X'

The rules for rounding the intermediate result are the same as those given in *Execution Model for IEEE Operations* on page 97.

If the instruction is *Floating Negative Multiply-Add* or *Floating Negative Multiply-Subtract*, the final result is negated.

4.6 Floating-Point Instructions

Primary opcode 63 is used for the double-precision arithmetic instructions and miscellaneous instructions, such as the *Floating-Point Status and Control Register Manipulation* instructions. Primary opcode 59 is used for the single-precision arithmetic instructions.

The single-precision instructions for which there is a corresponding double-precision instruction have the same format and extended opcode as the corresponding double-precision instruction.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in floating-point registers; to move floating-point data between storage and these registers; and to manipulate the FPSCR explicitly.

These instructions are divided into two categories.

- Computational instructions

  The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. They place status information into the FPSCR. They are the instructions described in *Floating-Point Arithmetic Instructions* on page 104, *Floating-Point Rounding and Conversion Instructions* on page 105, and *Floating-Point Compare Instructions* on page 105.

- Noncomputational instructions

  The noncomputational instructions that perform loads and stores, move the contents of a floating-point register to another floating-point register possibly altering the sign, manipulate the FPSCR explicitly, and select a value from one of two floating-point registers based on the value in a third floating-point register. These operations are not considered floating-point operations. With the exception of the instructions that manipulate the FPSCR explicitly, they do not alter the FPSCR. Those instructions are described in *Floating-Point Status and Control Register Instructions* on page 106.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number 2^{exponent}. Encodings are provided in the data format to represent finite numeric values, ±infinity, and values that are “not a number” (NaN). Operations involving infinities produce results following traditional mathematical conventions. NaNs have no mathematical interpretation, but their encoding supports a variable diagnostic information field. NaNs may be used to indicate such things as uninitialized variables, and can be produced by certain invalid operations.

One class of exceptions that occur during floating-point instruction execution is unique to floating-point operations: the Floating-point exception. Bits set in the FPSCR indicate floating-point exceptions. They can cause an Enabled exception type Program interrupt to be taken, precisely or imprecisely, if the proper control bits are set.

4.6.1 Instructions By Category

The floating-point instructions can be classified into computational and noncomputational categories. The computational instructions include those that perform arithmetic operations or conversions on operands. Noncomputational instructions perform loads/stores and moves (with possible sign changes), or select data. Additionally, some noncomputational instructions can write directly to the FPSCR. All instructions executed in the Load/Store Pipeline are noncomputational, while most executed in the Arithmetic pipe are computational.

All floating-point operands are stored internally in Double Precision Format. Arithmetic operations specified as Single, require that the internal data is representable as Single (that is, having an unbiased exponent between –126 and 127 and a significand accurately representable in 24 bits). If the data cannot be represented in this way, the results stored in FPR, and the status bits set in FPSCR and CR (as appropriate), are undefined.

For consistency, to reduce the likelihood of causing a serious malfunction due to user error, and to enable random testing, single-precision operations are performed on double-precision operands. For all cases except for **fdivs**, the operation is performed as if it were double-precision; the result is then rounded to single-precision. For **fdivs**, the appropriate number of iterations are performed to accomplish a single-precision result (potentially with early out); the quotient is then properly rounded.

In all cases, result exceptions (overflow, underflow, and inexact) are detected and reported based on the result, not on the source operands. Default (masked exception) results are the same as for the single-precision instructions. In the case of masked overflow or underflow exceptions, the least significant 11 bits of the adjusted true exponent are returned.

The results of all single-precision operations are rounded to Single Precision. These results are stored in Double-Precision format, but are restricted to Single-Precision range (exponent and fraction). All status bits are set based upon the single-precision result.

4.6.2 Load and Store Instructions

The PPC465-S FPU instruction set includes instructions to load from memory to an FPR, and to store from an FPR to memory.

For load instructions, the function of the LSC is to receive data from the 16 byte bus from the PPC465 Processor Complex and present it to the FPRs. Data received from PPC465 Processor Complex could be single or double precision, and in the big or little endian formats. Also, the data received is word aligned. Data to the FPR must be in the big endian, double precision format.

For store instructions one operand from the FPR, or a bypass path, is received. Data is to be word aligned on the output bus, not all cases can be handled with a throughput of one per cycle. FPR data (or bypassed data) for single precision stores that needs to be denormalized to fit in the single precision format requires multiple cycles. Also data for double precision stores may need to be normalized if the original data source was a single precision denormalized number. There are two basic forms of load instruction: single-precision and double-precision. Because the FPRs support only floating-point double format, single-precision *Load Floating-Point* instructions convert single-precision data to double format prior to loading the operand into the target FPR. The conversion and loading steps are as follows.

Let $WORD_{0:31}$ be the floating-point single-precision operand accessed from storage.

```
***Normalized Operand***
if $WORD_{1:8} > 0$ and $WORD_{1:8} < 255$ then
$FPR(FRT)_{0:1} \leftarrow WORD_{0:1}$
$FPR(FRT)_{2} \leftarrow \neg WORD_{1}$
$FPR(FRT)_{3} \leftarrow \neg WORD_{1}$
$FPR(FRT)_{4} \leftarrow \neg WORD_{1}$
$FPR(FRT)_{5:63} \leftarrow WORD_{2:31} \| {}^{29}0$
***Denormalized Operand***
if $WORD_{1:8} = 0$ and $WORD_{9:31} \neq 0$ then
sign $\leftarrow WORD_{0}$
exp $\leftarrow$ -126
$frac_{0:52} \leftarrow$ 0b0 $\| WORD_{9:31} \| {}^{29}0$
normalize the operand
do while $frac_{0} = 0$
        frac $\leftarrow frac_{1:52} \|$ 0b0
        exp $\leftarrow$ exp - 1
$FPR(FRT)_{0} \leftarrow$ sign
$FPR(FRT)_{1:11} \leftarrow$ exp + 1023
$FPR(FRT)_{12:63} \leftarrow frac_{1:52}$
```

```
***Zero / Infinity / NaN***
if $WORD_{1:8}$ = 255 or $WORD_{1:31}$ = 0 then
$FPR(FRT)_{0:1} \leftarrow WORD_{0:1}$
$FPR(FRT)_{2} \leftarrow WORD_{1}$
$FPR(FRT)_{3} \leftarrow WORD_{1}$
$FPR(FRT)_{4} \leftarrow WORD_{1}$
$FPR(FRT)_{5:63} \leftarrow WORD_{2:31} \,||\, {}^{29}0$
```

For double-precision *Load Floating-Point* instructions no conversion is required, as the data from storage are copied directly into the FPR.

Some of the *Floating-Point Load* instructions update GPR(RA) the effective address. For these forms, if RA $\neq$ 0, the effective address is placed into GPR(RA) and the storage element (byte, half word, word, or double word) addressed by EA is loaded into FPR(RT). If RA=0, the instruction form is invalid.

Floating-Point Load storage accesses cause Data Storage exceptions if the program is not allowed to read the storage location. *FLoating-Point Load* storage accesses cause Data TLB Error exceptions if the program attempts to access storage that is unavailable.

Note: RA and RB denote GPRs, while FRT denotes an FPR.

Both big endian and little endian byte orderings are supported.

Table 4-14. Floating-Point Load Instructions

Mnemonic	Operands	Instruction	Page
lfd	FRT, D(RA)	Load Floating-Point Double	576
lfdu	FRT, D(RA)	Load Floating-Point Double with Update	577
lfdux	FRT, RA, R	Load Floating-Point Double with Update Indexed	578
lfdx	FRT, RA, R	Load Floating-Point Double Indexed	579
lfs	FRT, D(RA)	Load Floating-Point Single	580
lfsu	FRT, D(RA)	Load Floating-Point Single with Update	581
lfsux	FRT, RA, RB	Load Floating-Point Single with Update Indexed	582
lfsx	FRT, RA, RB	Load Floating-Point Single Indexed	583

4.6.3 Floating-Point Store Instructions

There are three basic forms of store instruction: single-precision, double-precision, and integer. The integer form is provided by the **stfiwx** instruction, described on page 594. Because the FPRs support only floating-point double format for floating-point data, single-precision *Store Floating-Point* instructions convert double-precision data to single format before storing the operand in storage. The conversion steps are as follows.

Let $WORD_{0:31}$ be the word in storage written to.

```
***No Denormalization Required (includes Zero / Infinity / NaN)***
if $FPR(FRS)_{1:11}$ > 896 or $FPR(FRS)_{1:63}$ = 0 then
$WORD_{0:1} \leftarrow FPR(FRS)_{0:1}$
$WORD_{2:31} \leftarrow FPR(FRS)_{5:34}$
```

```
Denormalization Required
if $874 \le FRS_{1:11} \le 896$ then
sign ← $FPR(FRS)_0$
exp ← $FPR(FRS)_{1:11} - 1023$
frac ← 0b1 || $FPR(FRS)_{12:63}$
denormalize operand
do while exp < –126
        frac ← 0b0 || $frac_{0:62}$
        exp ← exp + 1
$WORD_0$ ← sign
$WORD_{1:8}$ ← 0x00
$WORD_{9:31}$ ← $frac_{1:23}$
else WORD ← undefined
```

Notice that if the value to be stored by a single-precision *Store Floating-Point* instruction is larger in magnitude than the maximum number representable in single format, the first case above (“No Denormalization Required”) applies. The result stored in WORD is then a well-defined value, but is not numerically equal to the value in the source register. The result of a single-precision *Load Floating-Point* from WORD will not compare equal to the contents of the original source register.

For double-precision *Store Floating-Point* instructions and for the *Store Floating-Point as Integer Word* instruction no conversion is required, as the data from the FPR are copied directly into storage.

Some of the *Floating-Point Store* instructions update GPR(RA) with the effective address. For these forms, if RA≠0, the effective address is placed into GPR(RA).

Floating-Point Store storage accesses will cause a Data Storage interrupt if the program is not allowed to write to the storage location. *Integer Store* storage accesses will cause a Data TLB Error interrupt if the program attempts to access storage that is unavailable.

Note: RA and RB denote GPRs, while FRS denotes an FPR.

Both big endian and little endian byte orderings are supported.

Table 4-15. Floating-Point Store Instructions

Mnemonic	Operands	Instruction	Page
stfd	FRS, D(RA)	Store Floating-Point Double	590
stfdu	FRS, D(RA)	Store Floating-Point Double with Update	591
stfdux	FRS, RA, RB	Store Floating-Point Double with Update Indexed	592
stfdx	FRS, RA, RB	Store Floating-Point Double Indexed	593
stfiwx	FRS, RA, RB	Store Floating-Point as Integer Word Indexed	594
stfs	FRS, D(RA)	Store Floating-Point Single	595
stfsu	FRS, D(RA)	Store Floating-Point Single with Update	596
stfsux	FRS, RA, RB	Store Floating-Point Single with Update Indexed	597
stfsx	FRS, RA, RB	Store Floating-Point Single Indexed	598

4.6.4 Floating-Point Move Instructions

These instructions copy data from one floating-point register to another, altering the sign bit (bit 0) as described in the instruction descriptions in *Floating Point Instruction Set* on page 543 for **fneg**, **fabs**, and **fnabs**. These instructions treat NaNs just like any other kind of value (for example, the sign bit of an NaN may be altered by **fneg**, **fabs**, and **fnabs**). These instructions do not alter the FSPCR.

Table 4-16. Floating-Point Move Instructions

Mnemonic	Operands	Instruction	Page
fabs	FRT, FRB	Floating Absolute Value	547
fmr	FRT, FRB	Floating Move Register	558
fnabs	FRT, FRB	Floating Negative Absolute Value	563
fneg	FRT, FRB	Floating Negate	564

4.6.5 Floating-Point Arithmetic Instructions

These instructions perform elementary arithmetic operations.

Table 4-17. Floating-Point Elementary Arithmetic Instructions

Mnemonic	Operands	Instruction	Page
fadd	FRT, FRA, FRB	Floating Add	548
fadds	FRT, FRA, FRB	Floating Add Single	549
fdiv	FRT, FRA, FRB	Floating Divide	554
fdivs	FRT, FRA, FRB	Floating Divide Single	555
fmul	FRT, FRA, FRB	Floating Multiply	561
fmuls	FRT, FRA, FRB	Floating Multiply Single	562
fres	FRT, FRB	Floating Reciprocal Estimate Single	569
frsqrte	FRT, FRB	Floating Reciprocal Square Root Estimate	571
fsub	FRT, FRA, FRB	Floating Subtract	574
fsubs	FRT, FRA, FRB	Floating Subtract Single	575

4.6.5.1 Floating-Point Multiply-Add Instructions

These instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide (L bit, FRACTION), and all 106 bits take part in the add/subtract portion of the instruction.

FPSCR bits are set as follows.

- Overflow, Underflow, and Inexact Exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, and not on the result of the multiplication.

- Invalid Operation exception bits are set as if the multiplication and the addition were performed using two separate instructions (**fmul[s]**, followed by **fadd[s]** or **fsub[s]**. That is, multiplication of infinity by 0 or of anything by an SNaN, and addition of an SNaN, cause the corresponding exception bits to be set.

Table 4-18. Floating-Point Multiply-Add Instructions

Mnemonic	Operands	Instruction	Page
fmadd	FRT, FRA, FRB, FRC	Floating Multiply-Add	556
fmadds	FRT, FRA, FRB, FRC	Floating Multiply-Add Single	557
fmsub	FRT, FRA, FRB, FRC	Floating Multiply-Subtract	559
fmsubs	FRT, FRA, FRB, FRC	Floating Multiply-Subtract Single	560
fnmadd	FRT, FRA, FRB, FRC	Floating Negative Multiply-Add	565
fnmadds	FRT, FRA, FRB, FRC	Floating Negative Multiply-Add Single	566
fnmsub	FRT, FRA, FRB, FRC	Floating Negative Multiply-Subtract	567
fnmsubs	FRT, FRA, FRB, FRC	Floating Negative Multiply-Subtract Single	568

4.6.6 Floating-Point Rounding and Conversion Instructions

Examples of uses of these instructions to perform various conversions can be found in Appendix X, “Floating-Point Conversions”, on page XREF TBD.

Table 4-19. Floating-Point Rounding and Conversion Instructions

Mnemonic	Operand	Instruction	Page
fctiw	FRT, FRB	Floating Convert To Integer Word	552
fctiwz	FRT, FRB	Floating Convert To Integer Word and round to Zero	553
frsp	FRT, FRB	Floating Round to Single-Precision	570

4.6.7 Floating-Point Compare Instructions

The floating-point *Compare* instructions compare the contents of two floating-point registers. Comparison ignores the sign of zero (+0 is treated as equal to –0). The comparison result can be ordered or unordered.

The comparison sets one bit in the designated CR field to 1 and the other three bits to 0. FPSCR[FPCC] is set in the same way.

The CR field and FPSCR[FPCC] are set as follows.

Table 4-20. Comparison Sets

Bit	Name	Description
0	FL	(FRA) < (FRB)
1	FG	(FRA) > (FRB)
2	FE	(FRA) = (FRB)
3	FU	(FRA) ? (FRB) (unordered)

Table 4-21. Floating-Point Compare and Select Instructions

Mnemonic	Operands	Instruction	Page
fcmpo	BF, FRA, FRB	Floating Compare Ordered	550
fcmpu	BF, FRA, FRB	Floating Compare Unordered	551
fsel	FRT, FRA, FRB, FRC	Floating Select	573

4.6.8 Floating-Point Status and Control Register Instructions

Every *Floating-Point Status and Control Register* instruction synchronizes the effects of all floating-point instructions executed by a given processor. Executing a *Floating-Point Status and Control Register* instruction ensures that all floating-point instructions previously initiated by the given processor have completed before the *Floating-Point Status and Control Register* instruction is initiated, and that no subsequent floating-point instructions are initiated by the given processor until the *Floating-Point Status and Control Register* instruction has completed. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the *Floating-Point Status and Control Register* instruction is initiated.
- All invocations of the Enabled exception type Program interrupt that will be caused by the previously initiated instructions have occurred before the *Floating-Point Status and Control Register* instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits is initiated until the *Floating-Point Status and Control Register* instruction has completed.

Floating-Point Load and *Floating-Point Store* instructions are not affected.

Table 4-22. Floating-Point Status and Control Register Instructions

Mnemonic	Operands	Instruction	Page
mcrfs		Move To Condition Register From FPSCR	584
mffs	FRT	Move From FPSCR	585
mtfsb0	BT	Move To FPSCR Bit 0	586
mtfsb1	BT	Move To FPSCR Bit 1	587
mtfsf	FLM, FRB	Move To FPSCR Fields	588
mtfsfi	BF,U	Move To FPSCR Field Immediate	589

5. Initialization

This chapter describes the initial state of the PPC465 (CPU core) and associated L2 cache after a hardware reset, and contains a description of the initialization software required to complete initialization so that the CPU core can begin executing application code. Initialization of other on-chip and/or off-chip system components may also be needed, in addition to the processor core initialization described in this chapter.

5.1 PPC465 State After Reset

In general, the contents of registers and other facilities within the PPC465 are undefined after a hardware reset. Reset is defined to initialize only the minimal resources required such that instructions can be fetched and executed from the initial program memory page, and so that repeatable, deterministic behavior can be guaranteed provided that the proper software initialization sequence is followed. System software must fully configure the rest of the PPC465 resources, as well as the other facilities within the chip and/or system.

The following list summarizes the requirements of the Book-E Enhanced PowerPC Architecture with regards to the processor state after reset, prior to any additional initialization by software.

- All fields of the MSR are set to 0, disabling all asynchronous interrupts, placing the processor in supervisor mode, and specifying that instruction and data accesses are to the system (as opposed to application) address space.
- DBCR0[RST] is set to 0, thereby ending any previous software-initiated reset operation.
- DBSR[MRR] records the type of the just ended reset operation (core, chip, or system; see *Reset Types* on page 95).
- TCR[WRC] is set to 0, thereby disabling the Watchdog timer reset operation.
- TSR[WRS] records the type of the just ended reset operation, if the reset was initiated by the Watchdog Timer (otherwise this field is unchanged from its pre-reset value).
- The PVR is defined, after reset and otherwise, to contain a value that indicates the specific processor implementation.
- The program counter (PC) is set to 0xFFFFFFFC, the effective address (EA) of the last word of the address space.

The memory management resources are set to values such that the processor is able to successfully fetch and execute instructions and read (but not write) data within the 4KB program memory page located at the end of the 32-bit effective address space. Exactly how this is accomplished is implementation-dependent. For example, it may or may not be the case that a TLB entry is established in a manner which is visible to software using the TLB management instructions. Regardless of how the implementation enables access to the initial program memory page, instruction execution starts at the effective address of 0xFFFFFFFC, the last word of the effective address space. The instruction at this address must be an unconditional branch backwards to the start of the initialization sequence, which must lie somewhere within the initial 4KB program memory page. The real address to which the initial effective address will be translated is also implementation or system-dependent, as are the various storage attributes of the initial program memory page such as the caching inhibited and endian attributes.

Note: In the PPC465 core, a single entry is established in the instruction shadow TLB (ITLB) and data shadow TLB (DTLB) at reset with the properties described in **Table 5-2**. It is required that initialization software insert an entry into the UTLB to cover this same memory region before performing any context synchronizing operation (including causing any exceptions which would lead to an interrupt), since a context synchronizing operation will invalidate the shadow TLB entries.

The PPC465 processor accesses the last 32 bytes of memory but only executes the instruction accessed from effect address 0xFFFFFFFC. The instruction must be an absolute branch within the last 4KB of memory. In the example below, the opcode for an absolute branch (b $+0xFFFFF004) to address 0xFFFFF000 (or 0xFFFFFFFF - 0xFFF) is 0x4BFFF004.

Table 5-1. Reset Vector

32-bit Boot Memory			16-bit Boot Memory			8-bit Boot Memory		
Access	Address	Instruction	Access	Address	Instruction	Access	Address	Instruction
1	0xFFFFFFE0		1	0xFFFFFFE0		1	0xFFFFFFE0	
2	0xFFFFFFE4		2	0xFFFFFFE2		2	0xFFFFFFE1	
3	0xFFFFFFE8		3	0xFFFFFFE4		3	0xFFFFFFE3	
4	0xFFFFFFEC		....	....	....	....	....	....
5	0xFFFFFFF0		13	0xFFFFFFF8		29	0xFFFFFFFC	0x4B
6	0xFFFFFFF4		14	0xFFFFFFFA		30	0xFFFFFFFD	0xFF
7	0xFFFFFFF8		15	0xFFFFFFFC	0x4BFF	31	0xFFFFFFFE	0xF0
8	0xFFFFFFFC	0x4BFFF004	16	0xFFFFFFFE	0xF004	32	0xFFFFFFFF	0x04

Initialization software should consider all other resources within the PPC465 core to be undefined after reset, in order for the initialization sequence to be compatible with other PowerPC implementations. There are, however, additional core resources which are initialized by reset, in order to guarantee correct and deterministic operation of the processor during the initialization sequence. **Table 5-2** shows the reset state of all PPC465 core resources which are defined to be initialized by reset. While certain other register fields and other facilities within the PPC465 core may be affected by reset, this is not an architectural nor hardware requirement, and software must treat those resources as undefined. Likewise, even those resources which are included in **Table 5-2** but which are not identified in the previous list as being architecturally required, should be treated as undefined by the initialization software.

During chip initialization, some chip control registers must be initialized to ensure proper chip operation. Peripheral devices can also be initialized as appropriate for the system design.

Table 5-2. Reset Values of Registers and Other PPC465 Facilities (Sheet 1 of 3)

Resource	Field	Reset Value	Comment
CCR0	DAPUIB	0	Enable broadcast of instruction data to auxiliary processor/FPU interface
	DTB	0	Enable broadcast of trace information
CCR1	ICDPEI	0	Disable Parity Error Insertion (enabled only for s/w testing)
	ICTPEI	0	
	DCTPEI	0	
	DCDPEI	0	
	DCUPEI	0	
	DCMPEI	0	
	FCOM	0	Do not force cache ops to miss.
	MMUPEI	0	Disable Parity Error Insertion (enabled only for s/w testing)
	FFF	0	Flush only as much data from dirty lines as needed.
	L2COBE	0	L2 Cache OP broadcast disabled.

Table 5-2. Reset Values of Registers and Other PPC465 Facilities (Sheet 2 of 3)

Resource	Field	Reset Value	Comment
DBCR0	EDM	0	External Debug mode disabled
	RST	0b00	Software-initiated debug reset disabled
	ICMP	0	Instruction completion debug events disabled
	BRT	0	Branch taken debug events disabled
	IAC1	0	Instruction Address Compare 1 (IAC1) debug events disabled
	IAC2	0	IAC2 debug events disabled
	IAC3	0	IAC3 debug events disabled
	IAC4	0	IAC4 debug events disabled
DBSR	UDE	0	Unconditional debug event has not occurred
	MRR	Reset-dependent	Indicates most recent type of reset as follows: 00 No reset has occurred since this field last cleared by software 01 Core reset 10 Chip reset 11 System reset
	ICMP	0	Instruction completion debug event has not occurred
	BRT	0	Branch taken debug event has not occurred
	IRPT	0	Interrupt debug event has not occurred
	TRAP	0	Trap debug event has not occurred
	IAC1	0	IAC1 debug event has not occurred
	IAC2	0	IAC2 debug event has not occurred
	IAC3	0	IAC3 debug event has not occurred
	IAC4	0	IAC4 debug event has not occurred
	DAC1R	0	Data address compare 1 (DAC1) read debug event has not occurred
	DAC1W	0	DAC1 write debug event has not occurred
	DAC2R	0	DAC2 read debug event has not occurred
	DAC2W	0	DAC2 write debug event has not occurred
	RET	0	Return debug event has not occurred
ESR	MCI	0	Synchronous Instruction Machine Check exception has not occurred
MCSR	MCS	0	Asynchronous Instruction Machine Check exception has not occurred
MSR	WE	0	Processor is not in wait state.
	CE	0	Asynchronous critical interrupts disabled
	EE	0	Asynchronous non-critical interrupts disabled
	PR	0	Processor in supervisor mode
	ME	0	Machine Check interrupts disabled
	DWE	0	Debug Wait mode disabled
	DE	0	Debug interrupts disabled
	FE	0	Floating-point Enabled interrupts disabled
	IS	0	Instruction fetch access is to system-level virtual address space
	DS	0	Data access is to system level virtual address space
PC		0xFFFFFFFC	Initial reset instruction fetched from last word of effective address space
PVR	0:31	System dependent	PVR value

Table 5-2. Reset Values of Registers and Other PPC465 Facilities (Sheet 3 of 3)

Resource	Field	Reset Value	Comment
RSTCFG	U0	0	U0 storage attribute
	U1	0	Memory page contain normal instructions or data
	U2	0	Storage misses do not cause a line to allocated in the data cache
	U3	0	U3 storage attribute
	E	0	Memory pages are big endian
	ERPN	System dependent	Extended Real Page Number
TCR	WRC	0b00	Watchdog Timer reset disabled
TLBentry[1]	$EPN_{0:19}$	0xFFFFF000	Match EA of initial reset instruction ($EPN_{20:21}$ are undefined, as they are not compared to the EA because the page size is 4KB).
	V	1	Translation table entry for the initial program memory page is valid.
	TS	0	Initial program memory page is in system-level virtual address space.
	SIZE	0b0001	Initial program memory page size is 4KB.
	TID	0x00	Initial program memory page is globally shared; no match required against PID register.
	$RPN_{0:21}$	0xFFFFF || 0b00	Initial program memory page mapped effective = real.
	ERPN	System-dependent	The ERPN (extended real page number) is the upper 4-bits of the 36-bit real address. The initial ERPN used by the boot vector is configured during reset based on the boot mode. It is set to 0xE when booting from on-chip memory (OCM) or 0xF when booting directly from NAND or NOR flash.
	U0	0	U0 storage attribute
	U1	0	Memory page contains normal instructions or data.
	U2	0	Storage misses do not cause a line to be allocated in the data cache.
	U3	0	U3 storage attribute
	W	0	Write-through storage attribute disabled. W and WL1 are set to 1.
	I	1	Caching inhibited storage attribute enabled. I, IL1, IL1D, IL2I and IL2D are set to 1.
	M	0	Memory coherent storage attribute disabled.
	G	1	Guarded storage attribute enabled.
	E	0	Memory pages are big endian.
	SX	1	Supervisor mode execution access enabled.
	SW	0	Supervisor mode write access disabled.
	SR	1	Supervisor mode read access enabled.
TSR	WRS	Copy of TCR[WRC]	If reset caused by Watchdog Timer
		Unchanged	If reset not caused by Watchdog Timer
		Undefined	After power-up

Note: TLBentry" refers to an entry in the shadow instruction and data TLB arrays that is automatically configured by the PPC465 to enable fetching and reading (but not writing) from the initial program memory page. This entry is not architecturally visible to software, and is invalidated upon any context synchronizing operation. Software must initialize a corresponding entry in the main unified TLB array before executing any operation which could lead to a context synchronization. Section 5.4 , "Initialization Software Requirements for PPC465 Embedded Processor Core" on page 113

5.2 Reset Types

The PPC465 supports three types of reset: core, chip, and system. The type of reset is indicated by a set of core input signals. For each type of reset, the core resources are initialized as indicated in **Table 5-2**. Core reset is intended to reset the PPC465 without necessarily resetting the rest of the on-chip logic. The chip reset operation is intended to reset the entire chip, but off-chip hardware in the system is not informed of the reset operation. System reset is intended to reset the entire chip, and also to signal the rest of the off-chip system that the chip is being reset.

5.3 Reset Sources

A reset operation can be initiated on the PPC465 through the use of any of four separate mechanisms. The first is a set of three input signals to the core, one for each of the three reset types. These signals can be asserted asynchronously by hardware outside the core to initiate a reset operation. The second reset source is the TCR[WRC] field, which can be setup by software to initiate a reset operation upon certain Watchdog Timer expiration events. The third reset source is the DBCR0[RST] field, which can be written by software to immediately initiate a reset operation. The fourth reset source is the JTAG interface, which can be used by a JTAG-attached debug tool to initiate a reset operation asynchronously to program execution on the PPC465.

5.4 Initialization Software Requirements for PPC465 Embedded Processor Core

After a reset operation occurs, the PPC465 is initialized to a minimum configuration to enable the fetching and execution of the software initialization code, and to guarantee deterministic behavior of the core during the execution of this code. Initialization software is necessary to complete the configuration of the processor core and the rest of the on-chip and off-chip system.

The system must provide non-volatile memory (or memory initialized by some mechanism other than the PPC465) at the real address corresponding to effective address 0xFFFFFFFC, and at the rest of the initial program memory page. The instruction at the initial address must be an unconditional branch backwards to the beginning of the initialization software sequence.

The initialization software functions described in this section perform the configuration tasks required to prepare the PPC465 to boot an operating system and subsequently execute an application program.

The initialization software must also perform functions associated with hardware resources that are outside the PPC465, and hence that are beyond the scope of this manual. This section makes reference to some of these functions, but their full scope is described in the user’s manual for the specific chip and/or system implementation.

Initialization software should perform the following tasks in order to fully configure the PPC465. For more information on the various functions referenced in the initialization sequence, see the corresponding chapters of this document.

1. Branch backwards from effective address 0xFFFFFFFC to the start of the initialization sequence
2. Invalidate the L1 instruction cache (**iccci**)
3. Invalidate the L1 data cache (**dccci**)
4. Synchronize memory accesses (**msync**)
5. Invalidate L2 cache, See Section 5.7 , “L2 Cache Array Invalidation” on page 117 for details.

   This step forces any data PLB operations that may have been in progress prior to the reset operation to complete, thereby allowing subsequent data accesses to be initiated and completed properly.
6. Clear DBCR0 register (disable all debug events)

Although the PPC465 is defined to reset some of the debug event enables during the reset operation (as specified in **Table 5-2**), this is not required by the architecture and hence the initialization software should not assume this behavior. Software should disable all debug events in order to prevent non-deterministic behavior on the trace interface to the core.

7. Clear DBSR register (initialize all debug event status)

   Although the PPC465 is defined to reset the DBSR debug event status bits during the reset operation (as specified in **Table 5-2**), this is not required by the architecture and hence the initialization software should not assume this behavior. Software should clear all such status in order to prevent nondeterministic behavior on the JTAG interface to the core.

8. Initialize CCR0 register
   a. Enable/disable broadcast of instructions to auxiliary processor (save power if no AP attached)
   b. Enable/disable broadcast of trace information (save power if not tracing)
   c. Specify behavior for icbt and dcbt/dcbtst instructions
   d. Enable/disable gathering of separate store accesses
   e. Enable/disable hardware support for misaligned data accesses
   f. Enable/disable parity error recoverability (recoverability lowers load/store performance marginally.)
   g. Enable/disable cache read of parity bits depending on s/w compatibility requirements

9. Initialize CCR1 register
   a. enable/disable full-line flushes as desired.
   b. disable force cache-op miss (FCOM) and various parity error insertion (xxxPEI).
   c. Users may wish to initialize CCR1[TCS] here, or in the timer facilities section.
   d. enable L2 Cache OP Broadcast Enable, CCR1[L2COBE] to ensure all cache Ops are functional with regard to L2 Cache.

10. Configure L1 instruction and data cache regions

    These steps must be performed prior to enabling the caches by setting the caching inhibited storage attribute of the corresponding TLB entry to 0.

    a. Clear the instruction and data cache normal victim index registers (INV0–INV3, DNV0–DNV3)
    b. Clear the instruction and data cache transient victim index registers (ITV0–ITV3, DTV0–DTV3)
    c. Set the instruction and data cache victim limit registers (IVLIM and DVLIM) according to the desired size of the normal, locked, and transient regions of each cache

11. Configure L2 cache and PLB interface

    L2 cache must be configured to be able to use L2 cache and enable proper PLB interface. See Section 5.7.3 , “L2 Cache Configuration Requirements” on page 118.

12. Setup TLB entry to cover initial program memory page

    Since the PPC465 core only initializes an architecturally-invisible shadow TLB entry during the reset operation, and since all shadow TLB entries are invalidated upon any context synchronization, special care must be taken during the initialization sequence to prevent any such context synchronizing operations (such as interrupts and the **isync** instruction) until after this step is completed, and an architected TLB entry has been established in the TLB. Particular care should be taken to avoid store operations, since write permission is disabled upon reset, and an attempt to execute any store operation would result in a Data Storage interrupt, thereby invalidating the shadow TLB entry.

    a. Initialize MMUCR
       - Specify TID field to be written to TLB entries
       - Specify TS field to be used for TLB searches
       - Specify store miss allocation behavior
       - Enable/disable transient cache mechanism
       - Enable/disable cache locking exceptions

b. Write TLB entry for initial program memory page
   - Specify EPN, RPN, ERPN, and SIZE as appropriate for system
   - Set valid bit
   - Specify TID = 0 (disable comparison to PID) or else initialize PID register to matching value
   - Specify TS = 0 (system address space) or else MSR[IS,DS] must be set to correspond to TS=1
   - Specify storage attributes (W, I, M, G, E, U0–U3) as appropriate for system
   - Enable supervisor mode fetch, read, and write access (SX, SR, SW)

c. Initialize PID register to match TID field of TLB entry (unless using TID = 0)

d. Setup for subsequent MSR[IS,DS] initialization to correspond to TS field of TLB entry
   Only necessary if TS field of TLB entry being set to 1 (MSR[IS,DS] already reset to 0)
   - Write new MSR value into SRR1
   - Write address from which to continue execution into SRR0

e. Setup for subsequent change in instruction fetch address
   Only necessary if EPN field of TLB entry changed from the initial value (EPN0:19 ≠ 0xFFFFF)

- Write initial/new MSR value into SRR1
- Write address from which to continue execution into SRR0

f. Fully initialize the TLB (tlbwe to all three words of each TLB entry; tlbre to TLB entries that are not fully initialized may result in parity exceptions).

g. Context synchronize to invalidate shadow TLB contents and cause new TLB contents to take effect
   - Use **isync** if not changing MSR contents and not changing the effective address of the rest of the initialization sequence
   - Use **rfi** if changing MSR to match new TS field of TLB entry (SRR1 will be copied into MSR, and program execution will resume at value in SRR0)
   - Use **rfi** if changing next instruction fetch address to correspond to new EPN field of TLB entry (SRR1 will be copied into MSR, and program execution will resume at value in SRR0)

   Instruction and data caches will now begin to be used, if the corresponding TLB entry has been setup with the caching inhibited storage attribute set to 0. Initialization software can now branch outside of the initial 4 KB memory region as controlled by the address and size of the new TLB entry and/or any other TLB entries which have been setup.

13. Initialize interrupt resources

    a. Initialize IVPR to specify high-order address of the interrupt handling routines
       Make sure that the corresponding address region is covered by a TLB entry (or entries)

    b. Initialize IVOR0–IVOR15 registers (individual interrupt vector addresses)
       Make sure that the corresponding addresses are covered by a TLB entry (or entries) Because the low order four bits of IVOR0–IVOR15 are reserved, the values written to those bits are ignored when the registers are written, and are read as zero when the registers are used. Therefore, all interrupt vector offsets are implicitly aligned on quadword boundaries. Software must take care to assure that all interrupt handlers are quadword-aligned.

    c. Setup corresponding memory contents with the interrupt handling routines

    d. Synchronize any program memory changes as required. (Section 6.2.2.1 , “Self-Modifying Code” on page 133 for more information on the instruction sequence necessary to synchronize changes to program memory prior to executing the new instructions.)

14. Configure debug facilities as desired

    a. Write DBCR1 and DBCR2 to specify IAC and DAC event conditions
    b. Clear DBSR to initialize IAC auto-toggle status
    c. Initialize IAC1–IAC4, DAC1–DAC2, DVC1–DVC2 registers to desired values
    d. Write MSR[DWE] to enable Debug Wait mode (if desired)
    e. Write DBCR0 to enable desired debug mode(s) and event(s)
    f. Context synchronize to establish new debug facility context (**isync**)

15. Configure timer facilities as desired

a. Write DEC to 0 to prevent Decrementer exception after TSR is cleared
b. Write TBL to 0 to prevent Fixed Interval Timer and Watchdog Timer exceptions after TSR is cleared, and to prevent increment into TBH prior to full initialization
c. CCR1[TCS] (Timer Clock Select) can be initialized here, or earlier with the rest of the CCR1.
d. Clear TSR to clear all timer exception status
e. Write TCR to configure and enable timers as desired
   Software must take care with respect to the enabling of the Watchdog Timer reset function, as once this function is enabled, it cannot be disabled except by reset itself
f. Initialize TBH value as desired
g. Initialize TBL value as desired
h. Initialize DECAR to desired value (if enabling the auto-reload function)
i. Initialize DEC to desired value

16. Initialize facilities outside the processor core which are possible sources of asynchronous interrupt requests (including DCRs and/or other memory-mapped resources)

    This must be done prior to enabling asynchronous interrupts in the MSR

17. Initialize the MSR to enable interrupts as desired
    a. Set MSR[CE] to enable/disable Critical Input and Watchdog Timer interrupts
    b. Set MSR[EE] to enable/disable External Input, Decrementer, and Fixed Interval Timer interrupts
    c. Set MSR[DE] to enable/disable Debug interrupts
    d. Set MSR[ME] to enable/disable Machine Check interrupts
       Software should first check the status of the ESR[MCI] field and MCSR[MCS] field to determine whether any Machine Check exceptions have occurred after these fields were cleared by reset and before Machine Check interrupts were enabled (by this step). Any such exceptions would have set ESR[MCI] or MCSR[MCS] to 1, and this status can only be cleared explicitly by software. After the MCSR[MCS] field is known to be clear, the MCSR status bits (MCSR[1:8]) should be cleared by software to avoid possible confusion upon later service of a machine check interrupt. Once MSR[ME] has been set to 1, subsequent Machine Check exceptions will result in a Machine Check interrupt.
    e. Context synchronize to establish new MSR context (**isync**)

18. Initialize any other processor core resources as required by the system (GPRs, SPRGs, and so on)

19. Initialize any other facilities outside the processor core as required by the system

20. Initialize system memory as required by the system software

    Synchronize any program memory changes as required. (Section 6.2.2.1 , “Self-Modifying Code” on page 133 for more information on the instruction sequence necessary to synchronize changes to program memory prior to executing the new instructions)

21. Start the system software

    System software is generally responsible for initializing and/or managing the rest of the MSR fields, including:

    a. MSR[FP] to enable or disable the execution of floating-point instructions and set CCR0[DAPUIB]=0 to enable broad cast to the floating point unit.
    b. MSR[FE0,FE1] to enable/disable Floating-Point Enabled exception type Program interrupts
    c. MSR[PR] to specify user mode or supervisor mode
    d. MSR[IS,DS] to specify application address space or system address space for instructions and data
    e. MSR[WE] to place the processor into Wait State (halt execution pending an interrupt)

5.5 PPC465L2C6 Core State After Reset

The various debug facilities of L2C to allow easier hardware debugging are disabled at the Reset, and these facilities need to be initialized by software as required. For additional details on these debug facilities, Section 7.2 , "L2 Debug Facilities" on page 190.

Various behaviors in the L2C Snoop unit exist, and they can be programmed. For information on configuring the Snoop unit for appropriate actions, refer to Section 7.1.6 , "Memory Coherence" on page 170.

5.6 L2 Cache Reset Requirements

PPC465L2C6, L2C, supports two levels of reset: Chip-Reset and Core-Reset.

- A chip-reset will reset the entire chip.
- A core-reset will reset the L2C core and will reset the associated PPC465, L1core. The core-reset will not reset the rest of the chip, including the PLB complex.

Because of the core-reset to reset only the L1core and L2C but not the PLB complex, the L2C will disconnect gracefully/orderly from the PLB before it honors the reset.

5.7 L2 Cache Array Invalidation

It is necessary at times to invalidate the entire L2C array. One such time is after power on reset because the valid bits of the tags in the L2C Array are in an unknown state. The Array invalidation is done by invalidating the tags, and the following is the sequence of events:

1. Software will set the Tag Array Invalidate bit (L2CR0[TAI]),
2. Upon observing this bit set, the L2C will wait for the array to become idle (meaning it is done with the current operation),
3. After transitioning to the idle state, the L2C will invalidate every tag entry in the tag array. Its state machine will sequence through all of the congruence classes. All four ways within a congruence class will be invalidated at once,
4. Upon completing this sequence, the L2C will clear the Tag Array Invalidate bit in the L2CR0[TAI]. This will indicate to software that the array invalidate sequence is completed.

Caution: Whenever L2C is invalidated, L1 core caches, especially L1 data cache, need to be carefully invalidated to maintain data coherence/integrity in the system.

5.7.1 L1core Requests During L2C Array Invalidate

While the L2C is performing the array invalidate sequence, it will accept and process any requests made by the L1core. The L2C will force all requests to be "L2-cache-inhibited" to avoid a stall on that request while it waits for the array to become available.

5.7.2 Snoop Requests During L2C Array Invalidate

If a snoop request is received during the array invalidate sequence, the L2C snoop logic will behave just as if it would for any other snoop request with one exception: The L2C snoop logic will not request an L2C array lookup and will treat the L2C array result as a cache-miss. However, "snoop flush" and "snoop kill" are honored at all times and appropriate commands will be sent to L1 cache.

5.7.3 L2 Cache Configuration Requirements

The following is a list of L2 cache configuration requirements for software, also refer the details of all DCRs to Section 7.1.12 , "L2 Cache DCR Registers" on page 179.

Caution: It is important to note that whenever L2C configuration is changed, L1 core caches, especially L1 data cache, need to be carefully invalidated to maintain data coherence/integrity in the system.

5.7.4 L2 Core Configuration

Coming out of reset the L1core and L2C must be configured to interface with the PLB. The following must be done before the L1core can start making load or store requests to the PLB:

- Write L2CR3[PLBPE, MSTOORE, MSTPD, MSTRP]:
  - Enable / disable PLB Parity (L2CR2[PLBPE]).
  - Enable / disable PLB Master Out-of-Order Read (L2CR2[MSTOORE]).
  - Configure the PLB Master Port Pipeline Depth (L2CR2[MSTPD]) to 0011 (depth of 4).
  - Configure the PLB Master Request Priority (L2CR2[MSTRP]) to 0.

5.7.5 L2C Array Configuration

At the completion of the hardware "reset" process, the L2C array is disabled (L2CR0[AS] == 00), that all requests are handled as if L2C is cache-inhibited regardless of the value of the IL2* bits of the requests from the L1core. The following must be initialized to configure the L2C array:

- Set/initialize L2CR0[TAA, DAA, DECC, DECA]:
  - Set the Tag Array Access (L2CR0[TAA]) and Data Array Access (L2CR0[DAA]) fields to zero.
  - Enable / disable ECC detection (L2CR0[DECC]) as desired.
  - If ECC is enabled, enable / disable Hardware Error Correction Writeback to the Array (L2CR0[DECA]) as desired.
- Set/initialize L2CR1[FDIOD, SL1ILU, SL1DLU, DSG, RAS, TAS]:
  - Enable / disable Force Data side In Order Data (L2CR1[FDIOD]) as desired.
  - Enable / disable Serial L1 Instruction Side Lookup Mode (L2CR1[SL1ILU]) as desired.
  - Enable / disable Serial L1 Data Side Lookup Mode (L2CR1[SL1DLU]) as desired.
  - Configure the Disable Store Gathering bit (L2CR1[DSG]) as desired.
  - Configure the Read Allocate Size (L2CR1[RAS]) as desired.
  - Configure the Touch Allocate Size (L2CR1[TAS]) as desired.
- Set/initialize L2MCRER[FARMEN, ITSBEN, ITDBEN, IDSBEN, IDDBEN, DTSBEN, DTDBEN, DDSBEN, DDDBEN]
  - Enable / disable Fixed Address Mode Mismatch Error Reporting (L2MCRER[FARMEN])
  - If ECC detection / correction is enabled:
    - Enable / disable Instruction Side Tag Single Bit Error Reporting (L2MCRER[ITSBEN])
    - Enable / disable Instruction Side Tag Double Bit Error Reporting (L2MCRER[ITDBEN])
    - Enable / disable Instruction Side Data Single Bit Error Reporting (L2MCRER[IDSBEN])
    - Enable / disable Instruction Side Data Double Bit Error Reporting (L2MCRER[IDDBEN])
    - Enable / disable Data Side Tag Single Bit Error Reporting (L2MCRER[DTSBEN])
    - Enable / disable Data Side Tag Double Bit Error Reporting (L2MCRER[DTDBEN])
    - Enable / disable Data Side Data Single Bit Error Reporting (L2MCRER[DDSBEN])
    - Enable / disable Data Side Data Double Bit Error Reporting (L2MCRER[DDDBEN])
- Set/initialize L2FER0
  - Clear all bits in this register to prevent errors from being forced.
- Set/initialize L2FER1
  - Clear all bits in this register to prevent errors from being forced.

- Set/initialize L2CR0[AS, TAI]
  - Set the Array Size field (L2CR0[AS]) to the appropriate value based on the L2's array size.
  - Set the Tag Array Invalidate bit (L2CR0[TAI]) to a '1'. Since the L2 tag array is in an unknown state coming out of reset, the tag array must be invalidated. The array is available for use when the L2 turns off L2CR0[TAI].

5.7.6 Configuring Fixed Address Mode

In order to use Fixed Address Mode, software must do the following:
- Configure the L2C array as specified in Section 5.7.3 , "L2 Cache Configuration Requirements" on page 118.
- Write the L2 Fixed Address Mode Address Register, L2FAMAR, with the base real address of the Fixed Address Region.
- Write L2CR1[PUBFAM]
  - Enable / disable Public Fixed Address Mode (L2CR1[PUBFAM])
- If Public FAM is enabled, write SLVERAPR with desired upper order 28 bits of 64-bit address.
- If Public FAM is enabled, write L2MCRER[SLVWDEN, SLVDSBEN, SLVDDBEN]
  - Enable / disable Write Data Error Reporting Enable (L2MCRER[SLVWDEN])
  - Enable / disable Slave Port Single Bit Error Reporting Enable (L2MCRER[SLVDSBEN])
  - Enable / disable Slave Port Double Bit Error Reporting Enable (L2MCRER[SLVDDBEN])
- Write L2CR1[FAMES]
  - Set the Fixed Address Mode Enable / Size field (L2CR1[FAMES]) to the desired value.
- Setup TLB entries for the Fixed Address Region:
  - The TLB entry that maps to the FAR on the processor containing the FAR must be configured as follows:
    - M == '0'
    - FAR == '1'
  - The TLB entry that maps to the FAR on all processors other than the processor containing the FAR must be configured as follows:
    - M == '0'
    - FAR == '0'

5.7.7 Configuring the Snooper

The L2C snooper is disabled coming out of reset. The snooper is enabled as follows:
- Write L2MCRER[SNPTSBEN, SNPTDBEN, SNPDSBEN, SNPDDBEN]
  - Enable / disable Snoop Tag Single Bit ECC Error Reporting Enable (L2MCRER[SNPTSBEN])
  - Enable / disable Snoop Tag Single Bit ECC Error Reporting Enable (L2MCRER[SNPTDBEN])
  - Enable / disable Snoop Data Double Bit ECC Error Reporting Enable (L2MCRER[SNPDSBEN])
  - Enable / disable Snoop Data Double Bit ECC Error Reporting Enable (L2MCRER[SNPDDBEN])
- Write L2CR2[MEI, CSNPKF, CSNPPEP, CSNPPF, FRS, SNPRQPD, L1CE, SNPME]
  - Select MESI or MEI mode (L2CR2[MEI]) as desired.
  - Enable / disable Converting Snoop Kills to Flushes (L2CR2[CSNPKF]).
  - Enable / disable Converting Snoop PushE to Pushes (L2CR2[CSNPPEP]).
  - Enable / disable Converting Snoop Pushes to Flushes (L2CR2[CSNPPF]).
  - Enable / disable Force Read-Shared (L2CR2[FRS]).
  - Configure the Snoop Request Pipeline Depth field (L2CR2[SNPRQPD]) to a desired value.
  - Enable / disable L1 Coherency (L2CR2[L1CE]).
  - Set the Snooping Master Enable bit (L2CR2[SNPME]) to '1'.

5.7.8 Debug Facility Configuration

Enabling Array Error Debug Events:

To enable L2C array Single Bit Error Debug Event:

- Write L2DBCR[SB]
    - Set the Single Bit Error Debug Event Enable (L2DBCR[SB])

To enable array Double Bit Error Debug Event:

- Write L2DBCR[DB]
    - Set the Double Bit Error Debug Event Enable (L2DBCR[DB])

5.7.8.1 Enabling Snoop Address Compare Debug Events:

The L2C supports two snoop address compare debug events. The configuration for the two is identical. The following must be set to enable the SnoopAC:

- Write SNPACn with the address value to compare the snoop address to.
- Write L2DBCR[SNPAnC, SNPAnS, SNPAnK, SNPAnF, SNPAnP, SNPAnPE]
    - Write the Snoop Address Compare n Condition field (L2DBCR[SNPAnC]) to the desired value.
    - Write the Snoop Address Compare n Size field (L2DBCR[SNPAnS]) to the desired value.
    - Set one or more of the following bits to ‘1’:
        - Snoop Address Compare n Kill Enable (L2DBCR[SNPAnK])
        - Snoop Address Compare n Flush Enable (L2DBCR[SNPAnF])
        - Snoop Address Compare n Push Enable (L2DBCR[SNPAnP])
        - Snoop Address Compare n PushE Enable (L2DBCR[SNPAnPE])

5.7.8.2 Enabling Slave Address Compare Debug Event:

The L2C supports one slave address compare debug event. The following must be set to enable the SlaveAC Write L2SLVAC0 with the address to compare the slave address to.

- Set one or more of the following bits to ‘1’ in L2DBCR:
    - Slave Address Compare Read Enable (L2DBCR[SLVAR])
    - Slave Address Compare Write Enable (L2DBCR[SLVAW])

5.7.9 Performing an Array Invalidate

To perform an array invalidate after the L2 has been configured, software must do the following:

- Write/Disable Fixed Address Mode (L2CR1[FAMES] == 00).
- Write L2CR0[AS, TAI]
    - Set the Array Size field (L2CR0[AS]) 256 Kbytes (AS=01).
    - Set the Tag Array Invalidate bit (L2CR0[TAI]) to a ‘1’.

Caution: Whenever L2C is invalidated, L1 core caches, especially L1 data cache, need to be carefully invalidated to maintain data coherence/integrity in the system.

5.7.10 Forcing Array Errors

Software capability to force single bit and/or double bit errors into the L2C array is provided via L2FER0 and L2FER1 registers.

To force a Single Bit error:

- Write L2FER0:
    - Set at least one bit in this register to a ‘1’
- Write L2FER1:
    - Clear any bits positions corresponding to set bit positions in L2FER0

To force a Double Bit Error:

- Write L2FER0:

  – Set at least one bit in this register to a ‘1’
- Write L2FER1:
  – Set any bits positions corresponding to set bit positions in L2FER0 to a ‘1’

5.7.11 Enabling L2C cache Way Locking

The L2C supports way locking in the L2C array. If Ways need to be locked, the following configuration must be set up:

- Configure the L2C array as specified in “L2C Array Configuration”.
- Write L2CR0[LKWAY, LKTW, LKEN]
  – Set one or more bits in the Locked Way field (L2CR0[LKWAY]) to indicate which ways are locked. This will prevent the L2 from writing a non-locking line fill to the locked way(s).
  – Set the Touch Target Way field (L2CR0[TTW]) to the way that the line should be touched into.
  – Set the Touch Target Way Enable bit (L2CR0[TTWEN]) to cause the next touch targeting the L2 to be touched into the way specified by L2CR0[TTW].

5.7.12 L2 Cache Example Initialization Code

The following code example starting with section showing how to initialize the L2 cache controller after reset.

1. Prior to configuring the L2 Cache controller, the memory queue (MQ) must be taken out of reset, the clocks enabled and then held in reset by the code in section l2_init below.

A: Deassert reset for memory controller and DDR PHY

```
Set SCU_SRST[MEMC] = 0.
Set SCU_SRST[DDR_PHY] = 0.
```

B: Enable the clocks for the Memory controller:

```
Set SCU_CLKEN[MEMC_AXI] = 1.
Set SCU_CLKEN[MEMC_PLB] = 1.
Set SCU_CLKEN[MEMC_DDR] = 1.
```

C: Place the memory controller in reset

```
Set SCU_CSR_SRST[MEMC] = 1.
```

2. The L2 cache controller initialized starts with section skip_mq_enable. #defines in the code select the different L2 Cache configurations for usage as a FAM, fixed address memory.

```
l2_init:
    /* This section enable DDR clken and SRST */
    lis    r3,0xdd8b
    ori    r3,r3,0x0000   /* Base address for  SCU_XXXXX registers */
    lwz    r4,0xc(r3)
    andi.  r5,r4,0x4000   /* Is SCU_SRST[MEMC]=1? */
    beq    __skip_mq_enable
    xori   r4,r4,0x5000
    stw    r4,0xc(r3)     /* SCU_SRST[MEMC]=0, SCU_SRST[DDR_PHY]=0 */
    lwz    r4,0x10(r3)
    ori    r4,r4,0x7000   /
    stw    r4,0x10(r3)    /* SCU_CLKEN[MEMC_AXI]=1, SCU_CLKEN[MEMC_PLB]=1
                          SCU_CLKEN[MEMC_DDR]=1 */
```

```
        lwz     r4,0xe0(r3)
        ori     r4,r4,0x4000
        stw     r4,0xe0(r3)     /* SCU_CSR_SRST[MEMC]=1 */
__skip_mq_enable:
        li      r5,0x2
        /* Invalidate L2 Cache */
        li      r3,L2_CACHE_L2CR0
        mtdcr   L2_CACHE_ADDR,r3
        lis     r4,0x1330
#ifdef CONFIG_SPLIT_HALF_FAM
        ori     r4,r4,0xe               /* L2CR0_TAI_MASK */
#else
        ori     r4,r4,2                 /* L2CR0_TAI_MASK */
#endif
        mtdcr   L2_CACHE_DATA,r4
        isync


        mfspr   r4,SPRN_CCR1
        ori     r4,r4,0x40              /* Enable L2C0BE */
        mtspr   SPRN_CCR1,r4


check_l2_tai_2:
        mfdcr   r3,L2_CACHE_DATA
        and     r3,r3,r5
        cmpwi   r3,0                    /* Wait for L2 Cache invalidate */
        bne     check_l2_tai_2

        li      r3,L2_CACHE_L2ERAPR
        mtdcr   L2_CACHE_ADDR, r3
        li      r4,0
        mtdcr   L2_CACHE_DATA,r4

        li      r3,L2_CACHE_L2SLVERAPR
        mtdcr   L2_CACHE_ADDR, r3
        li      r4,0
        mtdcr   L2_CACHE_DATA,r4

        li      r3,L2_CACHE_L2CR0
        mtdcr   L2_CACHE_ADDR,r3
        li      r4,0
        lis     r4,0x1110               /* Enable 256KB L2 array */
                                /* SET TAA=0x1 and DAA=0x1 */


/* NOTE: L2CR0[TAA, DAA] values used in the above code are specific to APM86190/290 Rev A. */

#ifdef CONFIG_SPLIT_HALF_FAM
        ori     r4,r4,0xc
#else
```

```
        ori     r4,r4,0x0
#endif
        mtdcr   L2_CACHE_DATA,r4

        li      r3,L2_CACHE_L2CR1
        mtdcr   L2_CACHE_ADDR,r3
#ifdef CONFIG_SPLIT_HALF_FAM
        lis     r4,0x6000
#else
        li      r4,0x0              /* L2CR1_FAMES_OFF */
#endif
        mtdcr   L2_CACHE_DATA,r4

        li      r3,L2_CACHE_L2CR2
        mtdcr   L2_CACHE_ADDR,r3
        li      r4,0                /* L2CR2_SNPME_MASK| (3 <<
L2CR2_SNPRQPD_SHIFT) | L2CR2_L1CE_MASK */
#if 0
        lis     r4,0x8030           /* Snoop enable & Snp pipeline */
#else
        lis     r4,0x8830           /* Also enable write w. flush
performace hit */
#endif
        ori     r4,r4,0x8000        /* L1CE */
        mtdcr   L2_CACHE_DATA,r4

        li      r3,L2_CACHE_L2CR3
        mtdcr   L2_CACHE_ADDR,r3
        li      r4,0                /* 3 << L2CR3_MSTPD_SHIFT */
        lis     r4,0x0300           /* Snoop enable & Snp pipeline */
        mtdcr   L2_CACHE_DATA,r4

        li      r3,L2_CACHE_L2FAMAR
        mtdcr   L2_CACHE_ADDR, r3
#ifdef CONFIG_SPLIT_HALF_FAM
        lis     r4,0x4000
        mfspr   r3,SPRN_PIR
        cmpwi   r3,CONFIG_SYS_MASTER_CPUID
        beq     set_l2_fam_done
        ori     r4,r4,0x2000
set_l2_fam_done:
#else
        li      r4,0
#endif
        mtdcr   L2_CACHE_DATA,r4

        li      r3,L2_CACHE_L2FER0
        mtdcr   L2_CACHE_ADDR, r3
        li      r4,0
        mtdcr   L2_CACHE_DATA,r4
```

```
li     r3,L2_CACHE_L2FER1
mtdcr  L2_CACHE_ADDR, r3
li     r4,0
mtdcr  L2_CACHE_DATA,r4

li     r3,L2_CACHE_L2MCSR
mtdcr  L2_CACHE_ADDR, r3
li     r4,-1
mtdcr  L2_CACHE_DATA,r4

li     r3,L2_CACHE_L2MCRER
mtdcr  L2_CACHE_ADDR, r3
li     r4,-1
mtdcr  L2_CACHE_DATA,r4

li     r3,L2_CACHE_L2DBCR
mtdcr  L2_CACHE_ADDR, r3
li     r4,0
mtdcr  L2_CACHE_DATA,r4

li     r3,L2_CACHE_L2DBSR
mtdcr  L2_CACHE_ADDR, r3
li     r3,0
mtdcr  L2_CACHE_DATA,r4

li     r3,L2_CACHE_L2SLVAC0
mtdcr  L2_CACHE_ADDR, r3
li     r4,0
mtdcr  L2_CACHE_DATA,r4

li     r3,L2_CACHE_L2SNPAC0
mtdcr  L2_CACHE_ADDR, r3
li     r4,0
mtdcr  L2_CACHE_DATA,r4

li     r3,L2_CACHE_L2SNPAC1
mtdcr  L2_CACHE_ADDR, r3
li     r4,0
mtdcr  L2_CACHE_DATA,r4

blr
```

6. Level 1 Cache

The PPC465 provides separate instruction and data cache controllers and arrays, which allow concurrent access and minimize pipeline stalls. The storage capacity of both cache arrays is 32KB. Both cache controllers have 32-byte lines, and both are highly associative, having 64-way set-associativity. The PowerPC instruction set provides a rich set of cache management instructions for software-enforced coherency. The PPC465 implementation also provides special debug instructions that can directly read the tag and data arrays. The cache controllers interface to the Processor Local Bus (PLB).

Both the data and instruction caches are parity protected against soft errors. If such errors are detected, the CPU will vector to the machine check interrupt handler, where software can take appropriate action. The details of suggested interrupt handling are described below in *Section 6.2 Instruction Cache Controller* and in *Section 6.3 Data Cache Controller*

The rest of this chapter provides more detailed information about the operation of the instruction and data cache controllers and arrays.

6.1 Cache Array Organization and Operation

The instruction and data cache arrays are organized identically, although the fields of the tag and data portions of the arrays are slightly different because the functions of the arrays differ, and because the instruction cache is virtually tagged while the data cache has real tags.

The organization of the cache into “ways” and “sets” is as follows. There are 64 ways in each set, with a set consisting of all 64 lines (one line from each way) at which a given memory location can reside. Conversely, there are 16 sets in each way, with a way consisting of 16 lines (one from each set).

Table 6-1 illustrates the ways and sets of the cache arrays. The tag field for each line in each way holds the high-order address bits associated with the line that currently resides in that way. The middle-order address bits form an index to select a specific set of the cache, while the five lowest-order address bits form a byte-offset to choose a specific byte (or bytes, depending on the size of the operation) from the 32-byte cache line.

Table 6-1. Instruction and Data Cache Array Organization

	Way 0	**Way 1**	• • •	**Way 62**	**Way 63**
Set 0	Line 0	Line 16	• • •	Line 992	Line 1008
Set 1	Line 1	Line 17	• • •	Line 993	Line 1009
⋮	⋮	⋮	⋮	⋮	⋮
Set 14	Line 14	Line 30	• • •	Line 1006	Line 1022
Set 15	Line 15	Line 31	• • •	Line 1007	Line 1023

In the cache array, an effective address (EA) is divided into three parts: tag, set, and byte offset. See *Figure 6-6* and *Figure 6-7* on page 137 for instruction cache tag address bits, and *Figure 6-8* and *Figure 6-9* on page 152 for data cache tag address bits. Also, see *Instruction Cache Synonyms* on page 133 for details on instruction cache synonyms associated with the use of virtual tags for the instruction cache. $A_{23:26}$ are the set address bits, and $A_{27:31}$ are the byte offset address bits.

6.1.1 Cache Line Replacement Policy

Memory addresses are specified as being cacheable or caching inhibited on a page basis, using the caching inhibited (I) storage attribute (see *Caching Inhibited (I)* on page 232). When a program references a cacheable memory location and that location is not already in the cache (a *cache miss*), the line may be brought into the cache (a *cache line fill* operation) and placed into any one of the ways within the set selected by the middle portion of the address (address bits $EA_{23:26}$ select the set). If the particular way within the set already contains a valid line from some other address, the existing line is removed and replaced by the newly referenced line from memory. The line being replaced is referred to as the *victim*.

The way selected to be the victim for replacement is controlled by a field within a Special Purpose Register (SPR). There is a separate “victim index field” for each set within the cache.

The following register figure shows the format for each of the four of the Victim Registers (VR):
- Instruction Cache Normal VR (INV0:INV3)
- Instruction Cache Transient VR (ITV0:ITV3)
- Data Cache Normal VR (DNV0:DNV3)
- Data Cache Transient VR (DTV0:DTV3)

Figure 6-1. Victim Registers (INV0:INV3) (ITV0:ITV3) (DNV0:DNV3) (DTV0:DTV3)

0:1		Reserved	
2:7	VNDXA	Victim Index A (for cache lines with EA[25:26] = 0b00)	
8:9		Reserved	
10:15	VNDXB	Victim Index B (for cache lines with EA[25:26] = 0b01)	
16:17		Reserved	
18:23	VNDXC	Victim Index C (for cache lines with EA[25:26] = 0b10)	
24:25		Reserved	
26:31	VNDXD	Victim Index D (for cache lines with EA[25:26] = 0b11)	

Note: Each of the victim index fields consist of six bits, as there are 64 ways in 32KB cache. Unused bits of the victim selection registers are reserved.

Each of the 16 SPRs illustrated in *Figure 6-1* can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**. In general, however, these registers are initialized by software once at startup, and then are managed automatically by hardware after that. Specifically, every time a new cache line is placed into the cache, the appropriate victim index field (as controlled by the type of access and the particular cache set being updated) is first referenced to determine which way within that set should be replaced. Then, that same field is incriminated such that the ways within that set are replaced in a *round-robin* fashion as each new line is brought into that set. When the victim index field value reaches the index of the last way (according to the size of the cache and the type of access being performed), the value is *wrapped back* to the index of the first way for that type of access. The first and last ways for the different types of accesses are controlled by fields in a pair of *victim limit* SPRs, one for each cache (see *Cache Locking and Transient Mechanism* on page 127 for more information).

The victim index field that is used varies according to the type of access and the address of the cache line. *Table 6-2* describes the correlation between the victim index fields and different access types, and addresses.

Table 6-2. Victim Index Field Selection

Address$_{23:26}$	Victim Index Field
0	xxV0[VNDXA]
1	xxV0[VNDXB]
2	xxV0[VNDXC]
3	xxV0[VNDXD]
4	xxV1[VNDXA]
5	xxV1[VNDXB]
6	xxV1[VNDXC]
7	xxV1[VNDXD]
8	xxV2[VNDXA]
9	xxV2[VNDXB]
10	xxV2[VNDXC]
11	xxV2[VNDXD]
12	xxV3[VNDXA]
13	xxV3[VNDXB]
14	xxV3[VNDXC]
15	xxV3[VNDXD]
Note: "xx" refers to "IN", "IT", "DN", or "DT", depending on whether the access is to the instruction or data cache, and whether the access is "normal" or "transient." (See *Cache Locking and Transient Mechanism* on page 127)	

6.1.2 Cache Locking and Transient Mechanism

Both caches support locking, at a "way" granularity. Any number of ways can be locked, from 0 ways to 63. At least one way must always be left unlocked, for use by cacheable line fills.

In addition, a portion of each cache can be designated as a "transient" region, by specifying that only a limited number of ways are used for cache lines from memory pages that are identified as being transient in nature by a storage attribute from the MMU (see *Memory Management* on page 219). For the instruction cache, such memory pages can be used for code sequences that are unlikely to be reused once the processor moves on to the next series of instruction lines. Thus, performance may be improved by preventing each series of instruction lines from overwriting the rest of the "regular" code in the instruction cache. Similarly, for the data cache, transient pages can be used for large "streaming" data structures, such as multimedia data. As each piece of the data stream is processed and written back to memory, the next piece can be brought in, overwriting the previous (now obsolete) cache lines instead of displacing other areas of the cache, which may contain other data that should remain in the cache.

A set of fields in a pair of victim limit registers specifies which ways of the cache are used for normal accesses and/or transient accesses, as well as which ways are locked. These registers, Instruction Cache Victim Limit (IVLIM) and Data Cache Victim Limit (DVLIM), are illustrated in *Figure 6-2*. They can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 6-2. Instruction Cache Victim Limit (IVLIM) and Data Cache Victim Limit (DVLIM) Registers			
0:3		Reserved	
4:9	TFLOOR	Transient Floor	
10:14		Reserved	
15:20	TCEILING	Transient Ceiling	
21:25		Reserved	
26:31	NFLOOR	Normal Floor	

When a cache line fill occurs as the result of a normal memory access (that is, one *not* marked as transient using the U1 storage attribute from the MMU; see *Memory Management* on page 219), the cache line to be replaced is selected by the corresponding victim index field from one of the normal victim index registers (INV0–INV3 for instruction cache lines, DNV0–DNV3 for data cache lines). As the processor increments any of these normal victim index fields according to the round-robin mechanism described in *Cache Line Replacement Policy* on page 126, the values of the fields are constrained to lie within the range specified by the NFLOOR field of the corresponding victim limit register, and the last way of the cache. That is, when one of the normal victim index fields is incremented past the last way of the cache, it wraps back to the value of the NFLOOR field of the associated victim limit register.

Similarly, when a cache line fill occurs as the result of a transient memory access, the cache line to be replaced is selected by the corresponding victim index field from one of the transient victim index registers (ITV0–ITV3 for instruction cache lines, DTV0–DTV3 for data cache lines). As the processor core increments any of these transient victim index fields according to the round-robin replacement mechanism, the values of the fields are constrained to lie within the range specified by the TFLOOR and the TCEILING fields of the corresponding victim limit register. That is, when one of the transient victim index fields is incremented past the TCEILING value of the associated victim limit register, it wraps back to the value of the TFLOOR field of that victim limit register.

Given the operation of this mechanism, if both the NFLOOR and TFLOOR fields are set to 0, and the TCEILING is set to the index of the last way of the cache, then all cache line fills—both normal and transient—are permitted to use the entire cache, and nothing is locked. Alternatively, if both the NFLOOR and TFLOOR fields are set to values greater than 0, the lines in those ways of the cache whose indexes are between 0 and the lower of the two floor values are effectively *locked*, as no cache line fills (neither normal nor transient) will be allowed to replace the lines in those ways. Yet another example is when the TFLOOR is lower than the NFLOOR, and the TCEILING is lower than the last way of the cache. In this scenario, the ways between the TFLOOR and the NFLOOR contain only transient lines, while the ways between the NFLOOR and the TCEILING may contain either normal or transient lines, and the ways from the TCEILING to the last way of the cache contain only normal lines.

Programming Note: It is a programming error for software to program the TCEILING field to a value lower than that of the TFLOOR field. Furthermore, software must initialize each of the normal and transient victim index fields to values that are between the ranges designated by the respective victim limit fields, prior to performing any cacheable accesses intended to utilize these ranges.

In order to setup a locked area within the data cache, software must perform the following steps (the procedure for the instruction cache is similar, with **icbt** instructions substituting for **dcbt** instructions):

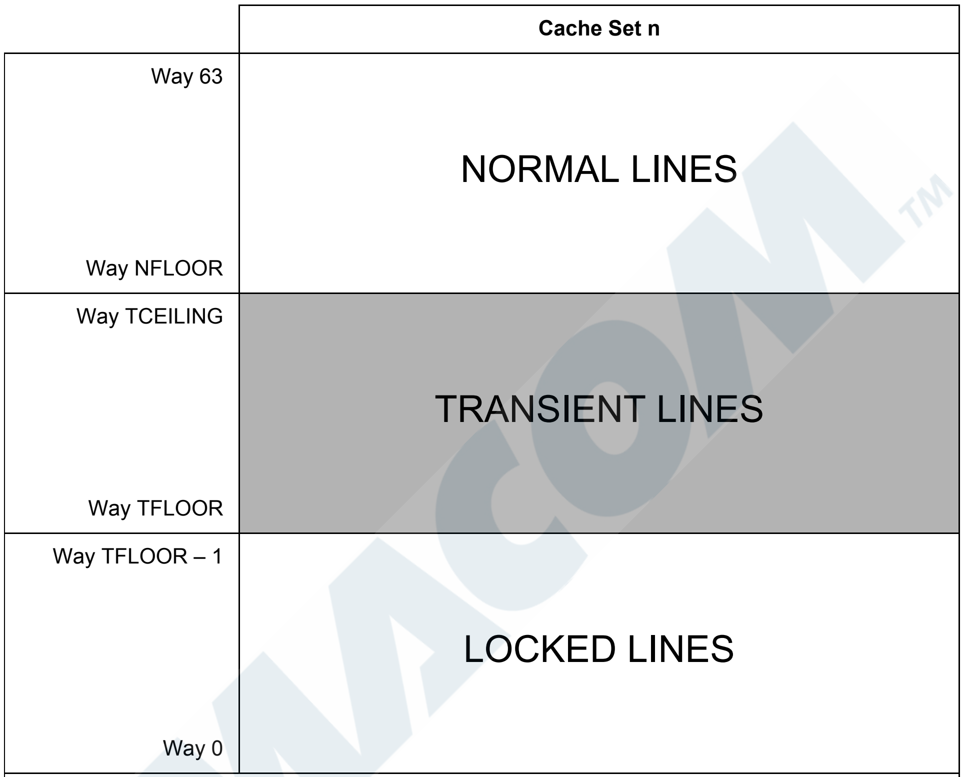
1. Execute **msync** and then **isync** to guarantee all previous cache operation have completed.
2. Mark all TLB entries associated with memory pages which are being used to perform the locking function as caching-inhibited. Leave the TLB entries associated with the memory pages containing the data which is to be locked into the data cache marked as cacheable, however.
3. Execute **msync** and then **isync** again, to cause the new TLB entry values to take effect.

4. Set both the NFLOOR and the TFLOOR values to the index of the first way which should be locked, and set the TCEILING value to the last way of the cache.
5. Set each of the normal and transient victim index fields to the same value as the NFLOOR and TFLOOR.
6. Execute **dcbt** instructions to the cache lines within the cacheable memory pages which contain the data which is to be locked in the data cache. The number of **dcbt** instructions executed to any given set should not exceed the number of ways which will exist in the locked region (otherwise not all of the lines will be able to be simultaneously locked in the data cache). Remember that when a series of **dcbt** instructions are executed to sequentially increasing addresses (with the address increment being the size of a cache block -- 32 bytes), it takes sixteen such **dcbt** operations (one for each set) before the next way of the initial set will be targeted again.
7. Execute **msync** and then **isync** again, to guarantee that all of the **dcbt** operations have completed and updated the corresponding victim index fields.
8. Set the NFLOOR, TFLOOR, and TCEILING values to the desired indices for the operating normal and transient regions of the cache. Both the NFLOOR and the TFLOOR values should be set higher than the highest locked way of the data cache; otherwise, subsequent normal and/or transient accesses could overwrite a way containing a line which was to be locked.
9. Set each of the normal and transient victim index fields to the value of the NFLOOR and TFLOOR, respectively.
10. Restore the cacheability of the memory pages which were used to perform the locking function to the desired operating values, by clearing the caching-inhibited attribute of the TLB entries which were updated in step 2.
11. Execute **msync** and then **isync** again, to cause the new TLB entry values to take effect.

The ways of the data cache whose indices are below the lower of the NFLOOR and TFLOOR values will now be locked.

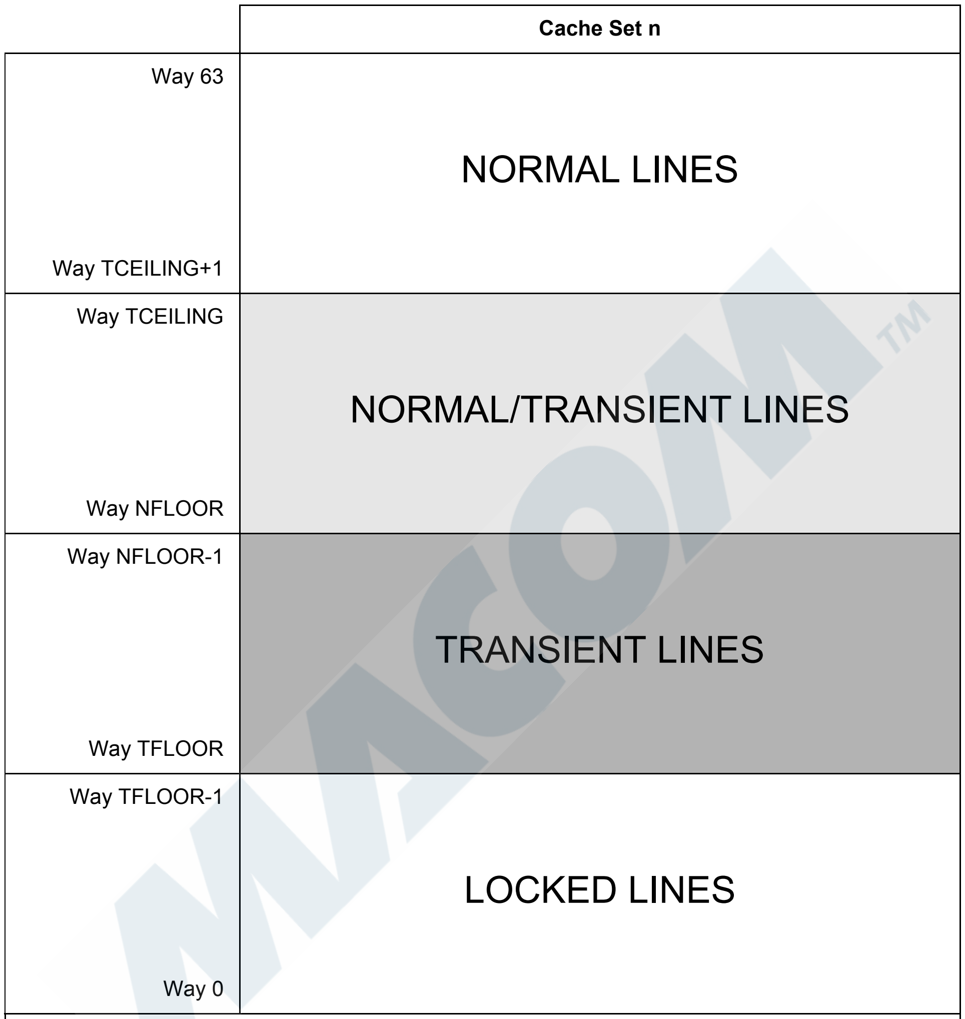
Figure 6-3 and *Figure 6-4* illustrate two examples of the use of the locking and transient mechanisms. Other configurations are possible, given the ability to program each of the victim limit fields to different relative values. Some configurations are not necessarily useful or practical.

Figure 6-3. Cache Locking and Transient Mechanism (Example 1)



Note: This example illustrates partitioning of the cache into locked, transient, and normal regions with no overlap. The figure illustrates a single set, but all sets of the cache are partitioned according to the same victim limit values.

Figure 6-4. Cache Locking and Transient Mechanism (Example 2)



Note: This example illustrates partitioning of the cache into locked, transient, and normal regions where the transient and normal regions partially overlap. The figure illustrates a single set, but all sets of the cache are partitioned according to the same victim limit values.

6.2 Instruction Cache Controller

The instruction cache controller (ICC) delivers two instructions per cycle to the instruction unit of the PPC465. The ICC interfaces to the L2C using a 128-bit read interface.

The ICC handles the execution of the PowerPC instruction cache management instructions, for touching (prefetching) or invalidating cache lines, or for flash invalidation of the entire cache. Resources for controlling and debugging the instruction cache operation are also provided.

6.2.1 ICC Operations

When the ICC receives an instruction fetch request from the instruction unit of the PPC465, the ICC simultaneously searches the instruction cache array for the cache line associated with the virtual address of the fetch request, and translates the virtual address into a real address (see *Memory Management* on page 219 for information about address translation). If the requested cache line is found in the array (a cache *hit*), the pair of instructions at the requested address are returned to the instruction unit. If the requested cache line is *not* found in the array (a cache *miss*), the ICC sends a request for the entire cache line (32 bytes) to the instruction L2C interface, using the real address. Note that the entire 32-byte cache line is requested, even if the caching inhibited (I) storage attribute is set for the memory page containing that cache line. Also note that the request to the instruction L2C interface is sent using the specific instruction address requested by the instruction unit, so that the memory subsystem may read the cache line *target word first* (if it supports such operation) and supply the requested instructions before retrieving the rest of the cache line.

As the ICC receives each portion of the cache line from the instruction L2C interface, it is placed into the instruction cache line fill data (ICLFD) buffer. Instructions from this buffer may be *bypassed* to the instruction unit as requested, without waiting for the entire cache line to be filled. Once the entire cache line has been filled into the buffer, and assuming that the memory page containing that line is cacheable, it is written into the instruction cache. If the memory page containing the line is caching inhibited, the line will remain in the ICLFD until it is displaced by a subsequent request for another cache line (either cacheable or caching inhibited).

If a memory subsystem error (such as an address time-out, invalid address, or some other type of hardware error external to the PPC465) occurs during the filling of the cache line, the line will not be written into the instruction cache, although instructions from the line may still be forwarded to the instruction unit from the ICLFD. Later, if execution of an instruction from that line is attempted, an Instruction Machine Check exception will be reported, and a Machine Check interrupt (if enabled) will result. See *Machine Check Interrupt* on page 263 for more information on Machine Check interrupts.

Once a request for a cache line read has been requested on the instruction L2C interface, the entire line read will be performed and the line will be written into the instruction cache (assuming no error occurs on the read), regardless of whether or not the instruction stream branches (or is interrupted) away from the line being read. This behavior is due to the nature of the PLB architecture, and the fact that once started, a cache line read request type cannot be abandoned. This does not mean, however, that the ICC will wait for this cache line read to complete before responding to a new request from the instruction unit (due, perhaps, to a branch redirection, or an interrupt). Instead, the ICC will immediately access the cache to determine if the cache line at the new address requested by the instruction unit is already in the cache. If so, the requested pair of instructions from this line will immediately be forwarded to the instruction unit, while the ICC in parallel continues to fill the previously requested cache line. In other words, the instruction cache is completely *non-blocking*.

If the newly requested cache line is instead a miss in the instruction cache, the ICC will immediately attempt to cancel the previous cache line read request. If the previous cache line read request has not yet been requested on the L2C bus, the old request will be cancelled and the new request will be made. If the previous cache line read request has already been requested, then as previously stated it cannot be abandoned, but the ICC will immediately present the request for the new cache line, such that it may be serviced immediately after the previous cache line read is completed. The ICC never aborts any L2C request once it has been made, except when a processor reset occurs while the L2C request is being made.

Programming Note:

It is a programming error for an instruction fetch request to reference a valid cache line in the instruction cache if the caching inhibited storage attribute is set for the memory page containing the cache line. The result of attempting to execute an instruction from such an access is undefined. After processor reset, hardware automatically sets the caching inhibited storage attribute for the memory page containing the reset address, and also automatically flash invalidates the instruction cache. Subsequently, lines will not be placed into the instruction cache unless they are accessed by reference to a memory page for which the caching inhibited attribute has been turned off. If software subsequently turns on the caching inhibited storage attribute for such a page, software must make sure that no lines from that page remain valid in the instruction cache, before attempting to fetch and execute instructions from the (now caching inhibited) page.

6.2.2 Instruction Cache Coherency

In general, the PPC465 does not automatically enforce coherency between the instruction cache, data cache, and memory. If the contents of memory location are changed, either within the data cache or within memory itself, and whether by the PPC465 through the execution of store instructions or by some other mechanism in the system writing to memory, software must use cache management instructions to ensure that the instruction cache is made coherent with these changes. This involves invalidating any obsolete copies of these memory locations within the instruction cache, so that they will be reread from memory the next time they are referenced by program execution.

6.2.2.1 Self-Modifying Code

To illustrate the use of the cache management instructions to enforce instruction cache coherency, consider the example of *self-modifying code*, whereby the program executing on the PPC465 stores new data to memory, with the intention of later branching to and executing this new “data,” which are actually instructions.

The following code example illustrates the required sequence for software to use when writing self-modifying code. This example assumes that *addr1* references a cacheable memory page.

```
stw      regN, addr1   # store the data (an instruction) in regN to addr1 in the data cache
dcbst    addr1         # write the new instruction from the data cache to memory
msync                  # wait until the data actually reaches the memory
icbi     addr1         # invalidate addr1 in the instruction cache if it exists
msync    addr1         # wait for the instruction cache invalidation to take effect
isync                  # flush any prefetched instructions within the ICC and instruction
                       # unit and re-fetch them (an older copy of the instruction at addr1
                       # may have already been fetched)
```

At this point, software may begin executing the instruction at addr1 and be guaranteed that the new instruction will be recognized.

6.2.2.2 Instruction Cache Synonyms

A synonym is a cache line that is associated with the same real address as another cache line that is in the cache array at the same time. Such synonyms can occur when different virtual addresses are mapped to the same real address, and the virtual address is used either as an index to the cache array (a *virtually-indexed* cache) or as the cache line tag (a *virtually-tagged* cache).

The instruction cache on the PPC465 is real-indexed but virtually-tagged and thus it is possible for synonyms to exist in the cache. (The data cache on the other hand is both real-indexed and real-tagged, and thus cannot have any synonyms.) Because of this, special care must be taken when managing instruction cache coherency and attempting to invalidate lines in the cache.

As explained in *Memory Management* on page 219, the virtual address (VA) consists of the 32-bit effective address (EA; for instruction fetches, this is the address calculated by the instruction unit and sent to the ICC) combined with the 8-bit Process ID (PID) and the 1-bit address space (MSR[IS] for instruction fetches). As described in *Table 6-2* on page 127, $VA_{27:31}$ chooses the byte offset within the cache line, while $VA_{23:26}$ is used as the *index* to select a set, and then the rest of the virtual address is used as the *tag*. The tag thus consists of $EA_{0:22}$, the PID, and MSR[IS] (for instruction fetches; for cache management instructions such as **icbi**, MSR[DS] is used to specify the address space; see the instruction descriptions for the instruction cache management instructions for more information). The tag portion of the VA is compared against the corresponding tag fields of each cache line within the way selected by $VA_{23:26}$.

Note that the address translation architecture of PowerPC Book-E is such that the low-order address bits 22:31 are always the same for the EA, VA, and real address (RA), because these bits are never translated due to the minimum page size being 1KB (these low-order 10 bits are always used for the byte offset within the page). As the page size increases, more and more low-order bits are used for the byte offset within the page, and thus fewer and fewer bits are translated between the VA and the RA (see *Table 8-3* on page 228). Synonyms only become possible when the system-level memory management software establishes multiple mappings to the same real page, which by definition involves different virtual addresses (either through differences in the higher-order EA bits which make up the VA, or through different process IDs, or different address spaces, or some combination of these three portions of the VA).

A further requirement for synonyms to exist in the instruction cache is for more than one of the virtual pages which map to a given real page to have *execute permission*, and for these pages to be cacheable (cache lines associated with pages without execute permission, or for which the caching inhibited storage attribute is set, cannot be placed in the instruction cache).

If the system-level memory management software permits instruction cache synonyms to be created, then extra care must be taken when attempting to invalidate instruction cache lines associated with a particular address. If software desires to invalidate only the cache line which is associated with a specific VA, then only a single **icbi** instruction need be executed, specifying that VA. If, however, software wishes to invalidate *all* instruction cache lines which are associated with a particular RA, then software must issue an **icbi** instruction for *each* VA which has a mapping to that particular RA and for which a line might exist in the instruction cache. In order to do this, the memory management software must keep track of which mappings to a given RA exist (or ever existed, if a mapping has been removed but cache lines associated with it might still exist), so that **icbi** instructions can be executed using the necessary VAs.

Alternatively, software can execute an **iccci** instruction, which flash invalidates the entire instruction cache without regard to the addresses with which the cache lines are associated.

6.2.3 Instruction Cache Control and Debug

The PPC465 provides various registers and instructions to control instruction cache operation and to help debug instruction cache problems.

6.2.3.1 Instruction Cache Management and Debug Instruction Summary

For detailed descriptions of the instructions summarized in this section, see *Instruction Set* on page 343 Also, see *Instruction Cache Coherency* on page 133 for more information on how these instructions are used to manage coherency in the instruction cache.

In the instruction descriptions, the term “block” describes the unit of storage operated on by the cache block instructions. For the PPC465, this is the same as a cache line.

The following instructions are used by software to manage the instruction cache:

icbi	Instruction Cache Block Invalidate Invalidates a cache block. No L2 cache lines will be invalidated (dcbi can be used for L2 cache invalidate).
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. See *icbt Operation* on page 135.
iccci	Instruction Cache Congruence Class Invalidate Flash invalidates the entire instruction cache. Execution of this instruction is privileged.
icread	Instruction Cache Read Reads a cache line (tag and data) from a specified index of the instruction cache, into a set of SPRs. Execution of this instruction is privileged. See *icread Operation* on page 136.

6.2.3.2 Core Configuration Register 0 (CCR0)

The CCR0 register controls the behavior of the **icbt** instruction. The CCR0 register also controls various other functions within the PPC465 that are unrelated to the instruction cache. Each of the these functions is discussed in more detail in the related sections of this manual. See *Figure 3-11* on page 74 for the detailed bit assignment of the CCR0 register.

6.2.3.3 Core Configuration Register 1 (CCR1)

The CCR1 register controls parity error insertion for software testing, one option for line flush behavior in the D-cache, and a control bit that selects the timer input clock. Each of the these functions is discussed in more detail in the related sections of this manual. See *Figure 3-12* on page 75 for the detailed bit assignment of the CCR1 register.

6.2.3.4 icbt Operation

The **icbt** instruction is typically used as a “hint” to the processor that a particular block of instructions is likely to be executed in the near future. Thus the processor can begin filling that block into the instruction cache, so that when the executing program eventually branches there the instructions will already be present in the cache, thereby improving performance.

Of course, it would not typically be advantageous if the filling of the cache line requested by the **icbt** itself caused a delay in the fetching of instructions needed by the currently executing program. For this reason, the default behavior of the **icbt** instruction is for it to have the lowest priority for sending a request to the L2C. If a subsequent instruction cache miss occurs due to a request from the instruction unit, then the line fill for the **icbt** will be abandoned (if it has not already been acknowledged on the L2C).

On the other hand, the **icbt** instruction can also be used as a convenient mechanism for setting up a fixed, known environment within the instruction cache. This is useful for establishing contents for cache line locking, or for deterministic performance on a particular sequence of code, or even for debugging of low-level hardware and software problems.

When being used for these latter purposes, it is important that the **icbt** instruction deliver a deterministic result, namely the guaranteed establishment in the cache of the specified line. Accordingly, the PPC465 provides a field in the CCR0 register that can be used to cause the **icbt** instruction to operate in this manner. Specifically, when the CCR0 [GICBT] field is set, the execution of **icbt** is *guaranteed* to establish the specified cache line in the instruction cache (assuming that a TLB entry for the referenced memory page exists and has both read and execute permission, and that the caching inhibited storage attribute is not set). The cache line fill associated with such a guaranteed **icbt** will not be abandoned due to subsequent instruction cache misses.

Operation of the **icbt** instruction is affected by the CCR1[FCOM] bit, which forces the **icbt** to appear to miss the cache, even if it should really be a hit. This causes two copies of the line to be established in the cache, simulating a multi-hit parity error. See *Simulating Instruction Cache Parity Errors for Software Testing* on page 138.

6.2.3.5 icread Operation

The **icread** instruction can be used to directly read both the tag and instruction information of a specified word in a specified entry of the instruction cache. The instruction information is read into the Instruction Cache Debug Data Register (ICDBDR), while the tag information is read into a pair of SPRs, the Instruction Cache Debug Tag Register High (ICDBTRH) and Instruction Cache Debug Tag Register Low (ICDBTRL). From there, the information can subsequently be moved into GPRs using **mfspr** instructions.

The execution of the **icread** instruction generates the equivalent of an EA, which is then broken down and used to select a specific instruction word from a specific cache line. $EA_{0:16}$ are ignored, $EA_{17:22}$ select the way, $EA_{23:26}$ select the set, and $EA_{27:29}$ select the word.

The EA generated by the **icread** instruction must be word-aligned (that is, $EA_{30:31}$ must be 0); otherwise, it is a programming error and the result is undefined.

If the CCR0[CRPE] bit is set, execution of the **icread** instruction also loads parity information into the ICBDTRH.

Execution of the **icread** instruction is privileged, and is intended for use for debugging purposes only.

Programming Note:

The PPC465 does not automatically synchronize context between an **icread** instruction and the subsequent **mfspr** instructions which read the results of the **icread** instruction into GPRs. In order to guarantee that the **mfspr** instructions obtain the results of the **icread** instruction, a sequence such as the following must be used:

```
icread      regA,regB    # read cache information (the contents of GPR A and GPR B are
                         # added and the result used to specify a cache line index to be read)
isync                    # ensure icread completes before attempting to read results
mficdbdr    regC         # move instruction information into GPR C
mficdbtrh   regD         # move high portion of tag into GPR D
mficdbtrl   regE         # move low portion of tag into GPR E
```

The following figures illustrate the ICDBDR, ICDBTRH, and ICDBTRL registers.

Figure 6-5. Instruction Cache Debug Data Register (ICDBDR)			
0:31		Instruction machine code from instruction cache	

Figure 6-6. Instruction Cache Debug Tag Register High (ICDBTRH)			
0:23		Tag Effective Address	Bits 0:23 of the 32-bit effective address associated with the cache line read by **icread**.
24	V	Cache Line Valid 0 Cache line is not valid. 1 Cache line is valid.	The valid indicator for the cache line read by **icread**.
25:26	TPAR	Tag Parity	The parity bits for the address tag for the cache line read by **icread**, if CCR0[CRPE] is set.
27	DAPAR	Instruction Data parity	The parity bit for the instruction word at the 32-bit effective address specified in the **icread** instruction, if CCR0[CRPE] is set.
28:31		Reserved	

Figure 6-7. Instruction Cache Debug Tag Register Low (ICDBTRL)			
0:21		Reserved	
22	TS	Translation Space	The address space portion of the virtual address associated with the cache line read by **icread**.
23	TD	Translation ID (TID) Disable 0 TID enable 1 TID disable	TID Disable field for the memory page associated with the cache line read by **icread**.
24:31	TID	Translation ID	TID field portion of the virtual address associated with the cache line read by **icread**.

6.2.3.6 Instruction Cache Parity Operations

The instruction cache contains parity bits and multi-hit detection hardware to protect against soft data errors. Both the instruction tags and data are protected. Instruction cache lines consist of a tag field, 256 bits of data, and 10 parity bits. The tag field is stored in CAM (Content Addressable Memory) cells, while the data and parity bits are stored in normal RAM cells. The instruction cache is real-indexed but virtually-tagged, so the tag field contains a TID field that is compared to the PID value, a TD bit that can be set to disable the TID comparison for shared pages, and the effective address bits to be compared to the fetch request. The exact number of effective address bits depends on the specific cache size.

Two types of errors may be detected by the instruction cache parity logic. In the first type, the parity bits stored in the RAM array are checked against the appropriate data in the instruction cache line when the RAM line is read for an instruction fetch. Note that a parity error will *not* be signaled as a result of an **icread** instruction.

The second type of parity error that may be detected is a multi-hit, sometimes referred to as an MHIT. This type of error may occur when a tag address bit is corrupted, leaving two tags in the instruction cache array that match the same input address. Multi-hit errors may be detected on any instruction fetch. No parity errors of any kind are detected on speculative fetch lookups or **icbt** lookups, Rather, such lookups are treated as cache hits and cause no further action until an instruction fetch lookup at the offending address causes an error to be detected.

If a parity error is detected, and the MSR[ME] is asserted, (i.e., Machine Check interrupts are enabled), the processor vectors to the Machine Check interrupt handler. As is the case for any Machine Check interrupt, after vectoring to the machine check handler, the MCSRR0 contains the value of the oldest "uncommitted" instruction in the pipeline at the time of the exception and MCSRR1 contains the old Machine Status Register (MSR) context. The interrupt handler is able to query Machine Check Status Register (MCSR) to find out that it was called due to a instruction cache parity error, and is then expected to invalidate the I-cache (using **iccci**). The handler returns to the interrupted process using the **rfmci** instruction.

As long as parity checking and machine check interrupts are enabled, instruction cache parity errors are always *recoverable*. That is, they are detected and cause a machine check interrupt before the parity error can cause the machine to update the architectural state with corrupt data. Also note that the machine check interrupt is *asynchronous*; that is, the return address in the MCSRR0 does not point at the instruction address that contains the parity error. Rather, the Machine Check interrupt is taken as soon as the parity error is detected, and some instructions in progress will get flushed and re-executed after the interrupt, just as if the machine were responding to an external interrupt.

6.2.3.7 Simulating Instruction Cache Parity Errors for Software Testing

Because parity errors occur in the cache infrequently and unpredictably, it is desirable to provide users with a way to simulate the effect of an instruction cache parity error so that interrupt handling software may be exercised. This is exactly the purpose of the CCR1[ICDPEI], CCR1[ICTPEI], and CCR1[FCOM] fields.

There are 10 parity bits stored in the RAM cells of each instruction cache line. Two of those bits hold the parity for the tag information, and the remaining 8 bits hold the parity for each of the 8 32-bit instruction words in the line. (There are two parity bits for the tag data because the parity is calculated for alternating bits of the tag field, to guard against a single particle strike event that upsets two adjacent bits. The instruction data bits are physically interleaved in such a way as to allow the use of a single parity bit per instruction word.) The parity bits are calculated and stored as the line is filled into the cache. Usually parity is calculated as the even parity for each set of bits to be protected, which the checking hardware expects. However, if any of the CCR1[ICTPEI] bits are set, the calculated parity for the corresponding bits of the tag are inverted and stored as odd parity. Similarly, if any of the CCR1[ICDPEI] bits are set, the parity for the corresponding instruction word is set to odd parity. Then, when the instructions stored with odd parity are fetched, they will cause a Parity exception type Machine Check interrupt and exercise the interrupt handling software. The following pseudo-code is an example of how to use the CCR1[ICDPEI] field to simulate a parity error on word 0 of a target cache line:

```
; make sure all this code in the cache before execution
icbi <target line address>     ; get the target line out of the cache
msync                          ; wait for the icbi
mtspr CCR1, 0x80000000         ; Set CCR1[ICDPEI0]
isync                          ; wait for the CCR1 context to update
icbt <target line address>     ; this line fills and sets odd parity for word 0
msync                          ; wait for the fill to finish
mtspr CCR1, 0x0                ; Reset CCR1[ICDPEI0]
isync                          ; wait for the CCR1 context to update
br <word 0 of target line>     ; fetching the target of the branch causes interrupt
```

Note that any instruction lines filled while bits are set in the CCR1[ICDPEI] or CCR1[ICTPEI] field will be affected, so users must code carefully to affect only the intended addresses.

The CCR1[FCOM] (Force Cache Operation Miss) bit enables the simulation of a multi-hit parity error. When set, it will cause an **icbt** to appear to be a miss, initiating a line fill, *even if the line is really already in the cache*. Thus, this bit allows the same line to be filled to the cache multiple times, which will generate a multi-hit parity error when an attempt is made to fetch an instruction from those cache lines. The following pseudo-code is an example of how to use the CCR1[FCOM] field to simulate a multi-hit parity error in the instruction cache:

```
; make sure all this code is cached and the "target line" is also
; in the cache before execution (use icbt as necessary)
```

```
mtspr CCR1, 0x00010000     ; Set CCR1[FCOM]
isync                      ; wait for the CCR1 context to update
icbt <target line address> ; this line fills a second copy of the target line
msync                      ; wait for the fill to finish
mtspr CCR1, 0x0            ; Reset CCR1[FCOM]
isync                      ; wait for the CCR1 context to update
br <word 0 of target line> ; fetching the target of the branch causes interrupt
```

6.3 Data Cache Controller

The data cache controller (DCC) handles the execution of the storage access instructions, moving data between memory, the data cache, and the PPC465 GPR file. The DCC interfaces to the L2C using two independent 128-bit interfaces, one for read operations and one for writes.

The DCC handles the execution of the PowerPC data cache management instructions, for touching (prefetching), flushing, invalidating, or zeroing cache lines, or for flash invalidation of the entire cache. Resources for controlling and debugging the data cache operation are also provided.

Extensive load, store, and flush queues are also provided, such that up to three outstanding line fills, up to four outstanding load misses, and up to two outstanding line flushes can be pending, with the DCC continuing to service subsequent load and store hits in an out-of-order fashion.

The rest of this section describes each of these functions in more detail.

6.3.1 DCC Operations

When the DCC executes a load, store, or data cache management instruction, the DCC first translates the effective address specified by the instruction into a real address (see *Memory Management* on page 219 for more information on address translation). Next, the DCC searches the data cache array for the cache line associated with the real address of the requested data. If the cache line is found in the array (a cache *hit*), that cache line is used to satisfy the request, according to the type of operation (load, store, and so on).

If the cache line is *not* found in the array (a cache *miss*), the next action depends upon the type of instruction being executed, as well as the storage attributes of the memory page containing the data being accessed. For most operations, and assuming the memory page is cacheable (see *Caching Inhibited (I)* on page 232), the DCC will send a request for the entire cache line (32 bytes) to the data read L2C interface. The request to the data read L2C interface is sent using the specific byte address requested by the instruction, so that the memory subsystem may read the cache line *target word first* (if it supports such operation) and supply the specific byte[s] requested before retrieving the rest of the cache line.

While the DCC is waiting for a cache line read to complete, it can continue to process subsequent instructions, and handle those accesses that hit in the data cache. That is, the data cache is completely *non-blocking*.

As the DCC receives each portion of the cache line from the data read L2C interface, it is placed into one of three data cache line fill data (DCLFD) buffers. Data from these buffers may be *bypassed* to the GPR file to satisfy load instructions, without waiting for the entire cache line to be filled. Once the entire cache line has been filled into the buffer, it will be written into the data cache at the first opportunity (either when the data cache is otherwise idle, or when subsequent operations require that the DCLFD buffer be written to the data cache).

If a memory subsystem error (such as an address time-out, invalid address, or some other type of hardware error external to the PPC465) occurs during the filling of the cache line, the line will still be written into the data cache, and data from the line may still be delivered to the GPR file for load instructions. However, the DCC will also report a Data Machine Check exception to the instruction unit of the PPC465, and a Machine Check interrupt (if enabled) will result. See *Machine Check Interrupt* on page 263 for more information on Machine Check interrupts.

Once a data cache line read request has been made, the entire line read will be performed and the line will be written into the data cache, regardless of whether or not the instruction stream branches (or is interrupted) away from the instruction which prompted the initial line read request. That is, if a data cache line read is initiated speculatively, before knowing whether or not a given instruction execution is really required (for example, on a load instruction which is after an unresolved branch), that line read will be completed, even if it is later determined that the cache line is not really needed. The DCC never aborts any L2C request once it has been made, except when a processor reset occurs while the L2C request is being made.

In general, the DCC will initiate memory read requests without waiting to determine whether the access is actually required by the *sequential execution model (SEM)*. That is, the request will be initiated *speculatively*, even if the instruction causing the request might be abandoned due to a branch, interrupt, or other change in the instruction flow. Of course, write requests to memory cannot be initiated speculatively, although a line fill request in response to a cacheable store access which misses in the data cache could be.

On the other hand, if the guarded storage attribute is set for the memory page being accessed, then the memory request will *not* be initiated until it is guaranteed that the access is required by the SEM. Once initiated, the access will not be abandoned, and the instruction is guaranteed to complete, *prior* to any change in the instruction stream. That is, if the instruction stream is interrupted, then upon return the instruction execution will resume *after* the instruction which accessed guarded storage, such that the guarded storage access will *not* be re-executed.

See *Guarded (G)* on page 233 for more information on accessing guarded storage.

Programming Note:

It is a programming error for a load, store, or **dcbz** instruction to reference a valid cache line in the data cache if the caching inhibited storage attribute is set for the memory page containing the cache line. The result of such an access is undefined. After processor reset, hardware automatically sets the caching inhibited storage attribute for the memory page containing the reset address, and software should flash invalidate the data cache (using **dccci**; see *Data Cache Management and Debug Instruction Summary* on page 146) before executing any load, store, or **dcbz** instructions. Subsequently, lines will not be placed into the data cache unless they are accessed by reference to a memory page for which the caching inhibited attribute has been turned off. If software subsequently turns on the caching inhibited storage attribute for such a page, software must make sure that no lines from that page remain valid in the data cache (typically by using the **dcbf** instruction), before attempting to access the (now caching inhibited) page with load, store, or **dcbz** instructions.

The only instructions that are permitted to reference a caching inhibited line which is a hit in the data cache are the cache management instructions **dcbst**, **dcbf**, **dcbi**, **dccci**, and **dcread**. The **dcbt** and **dcbtst** instructions have no effect if they reference a caching inhibited address, regardless of whether the line exists in the data cache.

6.3.1.1 Load and Store Alignment

The DCC implements all of the integer load and store instructions defined for 32-bit implementations by the PowerPC Book-E architecture. These include byte, half word, and word loads and stores, as well as load and store string (0 to 127 bytes) and load and store multiple (1 to 32 registers) instructions. Integer byte, half word, and word loads and stores are performed with a single access to memory if the entire data operand is contained within an aligned 16-byte (quad word) block of memory, regardless of the actual operand alignment within that block. If the data operand crosses a quad word boundary, the load or store is performed using two accesses to memory.

The load and store string and multiple instructions are performed using one memory access for each four bytes, unless and until an access would cross an aligned quad word boundary. The access that would cross the boundary is shortened to access just the number of bytes left within the current quad word block, and then the accesses are resumed with four bytes per access, starting at the beginning of the next quad word block, until the end of the load or store string or multiple is reached.

The DCC handles all misaligned integer load and store accesses in hardware, without causing an Alignment exception. However, the control bit CCR0[FLSTA] can be set to force all misaligned storage access instructions to cause an Alignment exception (see *Figure 3-11* on page 74). When this bit is set, all integer storage accesses must be aligned on an operand-size boundary, or an Alignment exception will result.

Load and store multiple instructions must be aligned on a 4-byte boundary, while load and store string instructions can be aligned on any boundary (these instructions are considered to reference *byte* strings, and hence the operand size is a byte).

Lwarx and stwcx are expected to be word aligned when L2CCR1[L2COBE] bit is set indicating L2C being present. All misaligned lwarx and stwcx operations result in an Alignment exception as permitted by PowerPC Book-E.

The DCC also supports load and store operations for the Floating Point unit. Floating-point loads and stores can access either four or eight bytes. Floating-point loads and stores are not subject to the function of CCR0[FLSTA].

6.3.1.2 Load Operations

Load instructions that reference cacheable memory pages and miss in the data cache result in cache line read requests being presented to the data read L2C interface. Load operations to caching inhibited memory pages, however, will only access the bytes specifically requested, according to the type of load instruction. This behavior (of only accessing the requested bytes) is only architecturally required when the guarded storage attribute is also set, but the DCC will enforce this requirement on any load to a caching inhibited memory page. Subsequent load operations to the same caching inhibited locations will cause new requests to be sent to the data read L2C interface (data from caching inhibited locations will not be reused from the DCLFD buffer).

The DCC includes three DCLFD buffers, such that a total of three independent data cache line fill requests can be in progress at one time. The DCC can continue to process subsequent load and store accesses while these line fills are in progress.

The DCC also includes a 4-entry *load miss queue (LMQ)*, which holds up to four outstanding load instructions that have either missed in the data cache or access caching inhibited memory pages. Collectively, any LMQ entries which reference cacheable memory pages can reference no more than three different cache lines, since there are only three DCLFD buffers. A load instruction in the LMQ remains there until the requested data arrives in the DCLFD buffer, at which time the data is delivered to the register file and the instruction is removed from the LMQ.

6.3.1.3 Store Operations

The processing of store instructions in the DCC is affected by several factors, including the caching inhibited (I), write-through (W), and guarded (G) storage attributes, as well as whether or not the allocation of data cache lines is enabled for cacheable store misses. There are three different behaviors to consider:

- Whether a data cache line is allocated (if the line is not already in the data cache)
- Whether the data is written directly to memory or only into the data cache
- Whether the store data can be *gathered* with store data from previous or subsequent store instructions before being written to memory

Allocation of Data Cache Line on Store Miss

Of course, if the caching inhibited attribute is set for the memory page being referenced by the store instruction, no data cache line will be allocated. For cacheable store accesses, allocation is controlled by one of two mechanisms: either by a “global” control bit in the Memory Management Unit Control Register (MMUCR), which is applied to all cacheable store accesses regardless of address; or by the U2 storage attribute for the memory page being accessed. See *Memory Management Unit Control Register (MMUCR)* on page 236 for more information on how store miss cache line allocation is controlled.

Regardless of which mechanism is controlling the allocation, if the corresponding bit is set, the cacheable store miss is handled as a *store without allocate (SWOA)*. That is, if SWOA is indicated, then if the access misses in the data cache, then the line will not be allocated (read from memory), and instead the byte[s] being stored will be written directly to memory. Of course, if the cache line has already been allocated and is being read into a DCLFD buffer (due perhaps to a previous cacheable load access), then the SWOA indication is ignored and the access is treated as if it were a store *with* allocate. Similarly, if SWOA is *not* indicated, the cache line *will* be allocated and the cacheable store miss will result in the cache line being read from memory.

Direct Write to Memory

Of course, if the caching inhibited attribute is set for the memory page being referenced by the store instruction, the data must be written directly to memory. For cacheable store accesses that are also write-through, the store data will also be written directly to memory, regardless of whether the access hits in the data cache, and independent of the SWOA mechanism. For cacheable store accesses that are *not* write through, whether the data is written directly to memory depends on both whether the access hits or misses in the data cache, and the SWOA mechanism. If the access is *either* a hit in the data cache, or if SWOA is *not* indicated, then the data will only be written to the data cache, and not to memory. Conversely, if the cacheable store access is both a miss in the data cache and SWOA is indicated, the access will be treated as if it were caching inhibited and the data will be written directly to memory and not to the data cache (since the data cache line is neither there already nor will it be allocated).

Store Gathering

In general, memory write operations caused by separate store instructions that specify locations in either write-through or caching inhibited storage may be gathered into one simultaneous access to memory. Similarly, store accesses that are handled as if they were caching inhibited (due to their being both a miss in the data cache and being indicated as SWOA) may be gathered. Store accesses that are only written into the data cache do not need to be gathered, because there is no performance penalty associated with the separate accesses to the array.

A given sequence of two store operations may only be gathered together if the targeted bytes are contained within the same aligned quad word of memory, and if they are contiguous with respect to each other. Subsequent store operations may continue to be gathered with the previously gathered sequence, subject to the same two rules (same aligned quad word and contiguous with the collection of previously gathered bytes). For example, a sequence of three store word operations to addresses 4, 8, and 0 may all be gathered together, as the first two are contiguous with each other, and the third (store word to address 0) is contiguous with the gathered combination of the previous two.

An additional requirement for store gathering applies to stores which target caching inhibited memory pages. Specifically, a given store to a caching inhibited page can only be gathered with previous store operations if the bytes targeted by the given store do not *overlap* with any of the previously gathered bytes. In other words, a store to a caching inhibited page must be both *contiguous* and *non-overlapping* with the previous store operation(s) with which it is being gathered. This ensures that the multiple write operations associated with a sequence of store instructions which each target a common caching inhibited location will each be performed independently on that target location.

Finally, a given store operation will *not* be gathered with an earlier store operation if it is separated from the earlier store operation by an **msync** or an **mbar** instruction, or if either of the two store operations reference a memory page which is both guarded and caching inhibited, or if store gathering is disabled altogether by CCR0[DSTG] (see *Figure 3-11* on page 74).

Table 6-3 summarizes how the various storage attributes and other circumstances affect the DCC behavior on store accesses.

Table 6-3. Data Cache Behavior on Store Accesses

Store Access Attributes					DCC Actions		
Caching Inhibited (I)	**Hit/Miss**	**SWOA**	**Write through (W)**	**Guarded (G)**	**Write Cache?**	**Write Memory?**	**Gather?[1]**
0	Hit	—	0	—	Yes	No	N/A
0	Hit	—	1	0	Yes	Yes	Yes
0	Hit	—	1	1	Yes	Yes	No
0	Miss	0	0	—	Yes	No	N/A
0	Miss	0	1	0	Yes	Yes	Yes
0	Miss	0	1	1	Yes	Yes	No
0	Miss	1	—	0	No	Yes	Yes
0	Miss	1	—	1	No	Yes	No
1	—[2]	—	—[3]	0	No	Yes	Yes[4]
1	—[2]	—	—[3]	1	No	Yes	No

Note 1: If store gathering is disabled altogether (by setting CCR0[DSTG] to 1), then such gathering will not occur, regardless of the indication in this table. Furthermore, where this table indicates that store gathering may occur it is presumed that the operations being gathered are targeting the same aligned quad word of memory, and are contiguous with respect to each other.

Note 2: It is a programming error for a data cache hit to occur on a store access to a caching inhibited page. The result of such an access is undefined.

Note 3: It is programming error for the write-through storage attribute to be set for a page which also has the caching inhibited storage attribute set. The result of an access to such a page is undefined.

Note 4: Stores to caching inhibited memory locations may only be gathered with previous store operations if none of the targeted bytes overlap with the bytes targeted by the previous store operations.

6.3.1.4 Line Flush Operations

When a store operation (or the **dcbz** instruction) writes data into the data cache without also writing the data to main memory, the cache line is said to become *dirty*, meaning that the data in the cache is the current value, whereas the value in memory is obsolete. Of course, when such a dirty cache line is replaced (due, for example, to a new cache line fill overwriting the existing line in the cache), the data in the cache line must be copied to memory. Otherwise, the results of the previous store operation[s] that caused the cache line to be marked as dirty would be lost. The operation of copying a dirty cache line to memory is referred to as a *cache line flush*. Cache lines are flushed either due to being replaced when a new cache line is filled, or in response to an explicit software flush request associated with the execution of a **dcbst** or **dcbf** instruction.

The DCC implements four dirty bits per cache line, one for each aligned double word within the cache line. Whenever any byte of a given double word is stored into a data cache line without also writing that same byte to memory, the corresponding dirty bit for that cache line is set (if CCR1[FFF] is set, then all four dirty bits are set instead of just the one corresponding dirty bit). When a data cache line is flushed, the type of request made to the data write L2C interface depends upon which dirty bits associated with the line are set, and the state of the CCR1[FFF] bit. If the CCR1[FFF] bit is set, the request will always be for an entire 32-byte line. Most users will leave the CCR1[FFF] reset to zero, in which case the controller minimizes the size of the transfer by the following algorithm. If only one dirty bit is set, the request type will be for a single double word write. If only two dirty bits are set, and they are in the same quad word, then the request type will be for a 16-byte line write. If two or more dirty bits are set, and they are in different quad words, the request type will be for an entire 32-byte line write. Regardless of the type of request generated by a cache line flush, the address is always specified as the first byte of the request.

If a store access occurs to a cache line in a memory page for which the write-through storage attribute is set, the dirty bits for that cache line do not get updated, since such a store access will be written directly to memory (and into the data cache as well, if the access is either a hit or if the cache line is allocated upon a miss).

On the other hand, it is permissible for there to exist multiple TLB entries that map to the same real memory page, but specify different values for the write-through storage attribute. In this case, it is possible for a store operation to a virtual page which is marked as non-write-through to have caused the cache line to be marked as dirty, so that a subsequent store operation to a different virtual page mapped to the same real page but marked as write-through encounters a dirty line in the data cache. If this happens, the store to the write-through page will write the data for the store to both the data cache and to memory, but it will not modify the dirty bits for the cache line.

6.3.1.5 Data Read L2C Interface Requests

When a L2C read request results from an access to a cacheable memory location, the request is always for a 32-byte line read, regardless of the type and size of the access that prompted the request. The address presented will be for the first byte of the target of the access.

On the other hand, when a L2C read request results from an access to a caching-inhibited memory location, only the byte[s] specifically accessed will be requested from the L2C, according to the type of instruction prompting the access. The following types of L2C read requests can occur due to caching inhibited requests:

- 1-byte read (any byte address 0–15 within a quad word)
- 2-byte read (any byte address 0–14 within a quad word)
- 3-byte read (any byte address 0–13 within a quad word)
- 4-byte read (any byte address 0–12 within a quad word)
- 8-byte read (any byte address 0–8 within a quad word)

  This request can only occur due to a double word floating-point load instruction
- 16-byte line fill (must be for byte address 0 of a quad word)

6.3.1.6 Data Write L2C Interface Requests

When aL2C write request results from a data cache line flush, the specific type and size of the request is as described in *Line Flush Operations* on page 143.

When a L2C write request results from store operations to caching-inhibited, write-through, and/or store without allocate (SWOA) memory locations, the type and size of the request can be any one of the following (this list includes the possible effects of store gathering; see *Store Gathering* on page 142):

- 1-byte write request (any byte address 0–15 within a quad word)
- 2-byte write request (any byte address 0–14 within a quad word)
- 3-byte write request (any byte address 0–13 within a quad word)
- 4-byte write request (any byte address 0–12 within a quad word)
- 5-byte write request (any byte address 0–11 within a quad word)

  Only possible due to store gathering
- 6-byte write request (any byte address 0–10 within a quad word)

Only possible due to store gathering

- 7-byte write request (any byte address 0–9 within a quad word)

  Only possible due to store gathering
- 8-byte write request (any byte address 0–8 within a quad word)

  Only possible due to store gathering, or due to a floating-point doubleword store
- 9-byte write request (any byte address 0–7 within a quad word)

  Only possible due to store gathering
- 10-byte write request (any byte address 0–6 within a quad word)

  Only possible due to store gathering
- 11-byte write request (any byte address 0–5 within a quad word)

  Only possible due to store gathering
- 12-byte write request (any byte address 0–4 within a quad word)

  Only possible due to store gathering
- 13-byte write request (any byte address 0–3 within a quad word)

  Only possible due to store gathering
- 14-byte write request (any byte address 0–2 within a quad word)

  Only possible due to store gathering
- 15-byte write request (any byte address 0–1 within a quad word)

  Only possible due to store gathering
- 16-byte line write request (must be to byte address 0 of a quad word)

  Only possible due to store gathering

6.3.1.7 Storage Access Ordering

In general, the DCC can perform load and store operations *out-of-order* with respect to the instruction stream. That is, the memory accesses associated with a sequence of load and store instructions may be performed in memory in an order different from that implied by the order of the instructions. For example, loads can be processed ahead of earlier stores, or stores can be processed ahead of earlier loads. Also, later loads and stores that hit in the data cache may be processed before earlier loads and stores that miss in the data cache.

The DCC does enforce the requirements of the SEM, such that the net result of a sequence of load and store operations is the same as that implied by the order of the instructions. This means, for example, that if a later load reads the same address written by an earlier store, the DCC guarantees that the load will use the data written by the store, and not the older “pre-store” data. But the memory subsystem could still see a read access associated with an even later load before it sees the write access associated with the earlier store.

If the DCC needs to make a read request to the data read L2C interface, and this request conflicts with (that is, references one or more of the same bytes as) an earlier write request which is being made to the data write L2C interface, the DCC will withhold the read request from the data read L2C interface until the write request has been acknowledged on the data write L2C interface. Once the earlier write request has been acknowledged, the read request will be presented, and it is the responsibility of the L2C subsystem to ensure that the data returned for the read request reflects the value of the data written by the write operation.

Conversely, if a write request conflicts with an earlier read request, the DCC will withhold the write request until the read request has been acknowledged, at which point it is the responsibility of the L2C subsystem to ensure that the data returned for the read request does *not* reflect the newer data being written by the write request.

The PPC465 provides storage synchronization instructions to enable software to control the order in which the memory accesses associated with a sequence of instructions are performed. See *Storage Ordering and Synchronization* on page 80 for more information on the use of these instructions.

6.3.2 Data Cache Coherency

The PPC465 core enforces the coherency of the data cache with respect to alterations of memory performed by entities/other devices when the storage attributes (see Chapter 5 Memory Management) WL1: L1 Data cache Write-Through, or IL1D: Caching Inhibited L1 Data cache, is set to '1', and M: Memory Coherence Required is set to '1', and the L2C is set to a Copy-back mode. The PPC465 core does not enforce 'Cache coherence' in all other storage attributes combinations. In a system comprised of the PPC465 cores, corresponding L2C cores, and PLB(v5), with the above storage attributes setting, Memory/Data Cache coherence is supported with MESI, Modified-Exclusive-Shared-Invalid, protocol.

Similarly, if devices other than the PPC465 cores/L2C cores are connected via the PLB(v5) bus in a system, Memory coherence can be enforced as long as such devices adapt the PLB(v5) protocol (see IBM Processor-Local-Bus version 5 spec). The PLB(v5) supports MESI protocol via the Snoop bus, and L2C and L1 Data cache have their own snooping mechanism to maintain data integrity/coherency.

It is the responsibility of software to select hardware supported Memory/Data cache coherency via storage attributes settings as above, or through the appropriate use of the caching inhibited storage attribute, the write-through storage attribute, and/or the data cache management instructions.

6.3.3 Data Cache Control and Debug

The PPC465 core provides various registers and instructions to control data cache operation and to help debug data cache problems. With the existence of L2C, an additional function is provided in Core Configuration Register1, CCR1, to control L1 and L2 cache operations.

6.3.3.1 Data Cache Management and Debug Instruction Summary

For detailed descriptions of the instructions summarized in this section, see *Instruction Set* on page 343

In the instruction descriptions, the term "block" describes the unit of storage operated on by the cache block instructions. For the PPC465, this is the same as a cache line.

The following instructions are used by software to manage the data cache.

dcba	Data Cache Block Allocate This instruction is implemented as a nop on the PPC465.

dcbf Data Cache Block Flush

Writes a cache block to memory (if the block has been modified) and then invalidates the block.

Also refer to *dcbf, dcbi, dcbst Operations* on page 94 for its behavior with regard to L2 cache

dcbi Data Cache Block Invalidate

Invalidates a cache block. Any modified data is discarded and not flushed to memory.

Execution of this instruction is privileged.

Also refer to *dcbf, dcbi, dcbst Operations* on page 94 for its behavior with regard to L2 cache

dcbst Data Cache Block Store

Writes a cache block to memory (if the block has been modified) and leaves the block valid but marked as unmodified.

Also refer to *dcbf, dcbi, dcbst Operations* on page 94 for its behavior with regard to L2 cache

dcbt Data Cache Block Touch

Initiates a cache block fill, enabling the fill to begin prior to the executing program requiring any data in the block. The program can subsequently access the data in the block without incurring a cache miss.

Also refer to *dcbf, dcbi, dcbst Operations* on page 94 for its behavior with regard to L2 cache.

dcbtst Data Cache Block Touch for Store

Implemented identically to the **dcbt** instruction.

Also refer to *dcbf, dcbi, dcbst Operations* on page 94 for its behavior with regard to L2 cache.

dcbz Data Cache Block Set to Zero

Establishes a cache line in the data cache and sets the line to all zeros, without first reading the previous contents of the cache block from memory, thereby improving performance. All four doublewords in the line are marked as dirty.

This instruction does not affect L2 cache.

If a storage space with IL1d = 1 or WL1 = 1 is accessed with this instruction, “Alignment exception” will be generated.

dccci Data Cache Congruence Class Invalidate

Flash invalidates the entire data cache. Execution of this instruction is privileged.

This instruction does not affect L2 cache.

dcread Data Cache Read

Reads a cache line (tag and data) from a specified index of the data cache, into a GPR and a pair of SPRs. Execution of this instruction is privileged.

See *dcread Operation* on page 151.

6.3.3.2 Core Configuration Register 0 (CCR0)

The CCR0 register controls the behavior of the **dcbt** instruction, the handling of misaligned memory accesses, and the store gathering mechanism. The CCR0 register also controls various other functions within the PPC465 that are unrelated to the data cache. Each of these functions is discussed in more detail in the related sections of this manual.

Figure 3-11 on page 74 illustrates the fields of the CCR0 register.

6.3.3.3 Core Configuration Register 1 (CCR1)

The CCR1 register controls the behavior of the line flushes in response to cast-outs or **dcbf** or **dcbst** instructions. It also contains bits to control the artificial injection of parity errors for software testing purposes. Some of those bits affect the data cache and L2 cache, while other control the MMU or instruction cache. CCR1[L2COBE] (CCR1 bit 25, “L2 Cache OP Broadcast Enable”): This bit is not guaranteed to be reset to any particular value. It should be set to one to communicate with the L2 cache controller. When set to 1, cache ops are supported by communication with the L2 cache as detailed in the related sections. When set to 0, there is presumed to be no L2 cache, and the logic will work identically to that of the previous core (i.e. cache ops such as dcbi, dcbf, and dcbst will not be signalled to the L2 cache.)

Each of these functions is discussed in more detail in the related sections of this manual.

Figure 3-12 on page 75 illustrates the fields of the CCR1 register.

6.3.3.4 dcbt and dcbtst Operation

The **dcbt** instruction is typically used as a “hint” to the processor that a particular block of data is likely to be referenced by the executing program in the near future. Thus the processor can begin filling that block into the data cache, so that when the executing program eventually performs a load from the block it will already be present in the cache, thereby improving performance.

The **dcbtst** instruction is typically used for a similar purpose, but specifically for cases where the executing program is likely to *store* to the referenced block in the near future. The differentiation in the purpose of the **dcbtst** instruction relative to the **dcbt** instruction is only relevant within shared-memory systems with hardware-enforced support for cache coherency. In such systems, the **dcbtst** instruction would attempt to establish the block within the data cache in such a fashion that the processor would most readily be able to subsequently write to the block (for example, in a processor with a *MESI-protocol* cache subsystem, the block might be obtained in *Exclusive* state).

Of course, it would not typically be advantageous if the filling of the cache line requested by the **dcbt/dcbtst** itself caused a delay in the reading of data needed by the currently executing program. For this reason, the default behavior of the **dcbt/dcbtst** instruction is for it to be ignored if the filling of the requested cache block cannot be immediately commenced and waiting for such commencement would result in the DCC execution pipeline being stalled. For example, the **dcbt/dcbtst** instruction will be ignored if all three DCLFD buffers are already in use, and execution of subsequent storage access instructions is pending.

On the other hand, the **dcbt/dcbtst** instruction can also be used as a convenient mechanism for setting up a fixed, known environment within the data cache. This is useful for establishing contents for cache line locking, or for deterministic performance on a particular sequence of code, or even for debugging of low-level hardware and software problems.

When being used for these latter purposes, it is important that the **dcbt/dcbtst** instruction deliver a deterministic result, namely the guaranteed establishment in the cache of the specified line. Accordingly, the PPC465 provides a field in the CCR0 register which can be used to cause the **dcbt/dcbtst** instruction to operate in this manner. Specifically, when the CCR0 [GDCBT] field is set, the execution of **dcbt/dcbtst** is *guaranteed* to establish the specified cache line in the data cache (assuming that a TLB entry for the referenced memory page exists and has read permission, and that the caching inhibited storage attribute is not set). The cache line fill associated with such a guaranteed **dcbt/dcbtst** will occur regardless of any potential instruction execution-stalling circumstances within the DCC.

Operation of the **dcbt/dcbtst** instruction is affected by the CCR1[FCOM] bit, which forces the **dcbt/dcbtst** to appear to miss the cache, even if it should really be a hit. This causes two copies of the line to be established in the cache, simulating a multi-hit parity error. See *Simulating Data Cache Parity Errors for Software Testing* on page 155.

With the existence of a L2 cache, the behaviors of **dcbt** and **dcbtst** instructions are modified and affect both L1 and L2 caches as described in *Table 6-4*.

*Table 6-4. Behavior of **dcbt** and **dcbtst** Operations*

L1 Hit	IL1D	I	Operation without L2 CCR1[L2COBE]=0	Operation with L2 CCR1[L2COBE]=1
yes	0	n/a	Do nothing else. L1 cache has the required data. Note that IL1D must be 0 to get an L1 hit.	Do nothing else. L1 cache has the required data. Note that IL1D must be 0 to get an L1 hit.
no	0	n/a	Allocate a line fill buffer and request the data from the L2/PLB5 interface. Do not indicate dcbt/dcbtst to the L2.	Allocate a line fill buffer and request the data from the L2/PLB5 interface. Assert signals to inform the L2 that the request is a dcbt/dcbst with IL1D=0. (The L2 uses the touch indicators to enable filling locked lines.) L1 waits for data and fills the cache normally.
n/a	n/a	1	Do nothing else. The location is inhibited in all levels of cache.	Do nothing else. The location is inhibited in all levels of cache.
n/a	1	0	Do nothing else. The location is inhibited in L1 and no broadcast of cache ops is enabled.	Allocate a line fill buffer and request the data from the L2/PLB5 interface. Assert signals to inform the L2 that the request is a dcbt/dcbst with IL1D=1. Use the ack from the L2 to mark the fill done, then treat is as if it were a non-cacheable request; i.e. do not fill the L1 cache. L2 must not return any data.

Note: that guaranteed touch modes (see CCR0[GICBT] and CCR0[GDBCT]) are guaranteed only with respect to the L1 caches. No guarantee is made with respect to the L2 cache.

6.3.4 dcbf, dcbi, dcbst Operations

The behavior of the **dcbf**, **dcbi**, and **dcbst** instructions is described in *Table 6-5*

*Table 6-5. Behavior of **dcbf, dcbi, dcbst***

op	L1 hit?	dirty?	Operation without L2 CCR1[L2COBE]=0	Operation with L2 CCR1[L2COBE]=1
dcbf	yes	yes	Invalidate the entry in the L1 cache and send the address and data down the write pipeline to be written to memory.	IInvalidate the entry in the L1 cache and send the address and data down the write pipeline to the L2. Assert signals to inform L2 cache that this operation is a dirty dcbf hit. L2 will invalidate any valid copy it has and write L1 data to memory (see note).
dcbf	yes	no	Invalidate the entry in the L1 cache. No data is written to memory.	Invalidate the entry in the L1 cache and send the address (no data) down the write pipeline to the L2. Assert signals to inform L2 cache that this operation is a clean dcbf hit. L2 will search for a valid copy of the data. If that search misses or finds a clean line, no data is copied to memory. If the L2 search finds dirty data, write it to memory. Invalidate any valid copy in the L2 (see note).

Table 6-5. Behavior of ***dcbf, dcbi, dcbst*** *(continued)*

op	L1 hit?	dirty?	Operation without L2 CCR1[L2COBE]=0	Operation with L2 CCR1[L2COBE]=1
dcbf	no	n/a	Do nothing else.	Send the address (no data) down the write pipeline to the L2, as in the case of a clean dcbf hit. L2 will search for a valid copy of the data. If that search misses or finds a clean line, no data is copied to memory. If the L2 search finds dirty data, write it to memory. Invalidate any valid copy in the L2 (see note).
dcbst	yes	yes	Mark the entry in the L1 cache as clean, and send the address and data down the write pipeline to be written to memory.	Mark the entry in the L1 cache as clean, and send the address and data down the write pipeline to the L2. Assert signals to inform L2 cache that this operation is a dirty dcbst hit. L2 will update any valid copy it has with new data, mark the entry clean, and write L1 data to memory (see note).
dcbst	yes	no	Do nothing else.	Send the address (no data) down the write pipeline to the L2. Assert signals to inform L2 cache that this operation is a clean dcbst hit. L2 will search for a valid copy of the data. If that search misses or finds a clean line, no data is copied to memory. If the L2 search finds dirty data, write it to memory, and mark the entry as clean in the L2 (see note).
dcbst	no	n/a	Do nothing else.	Send address (no data) down the write pipeline to the L2. Assert signals to inform L2 cache that this operation is a dcbst miss. L2 will search for a valid copy of the data. If that search misses or finds a clean line, no data is copied to memory. If the L2 search finds dirty data, write it to memory, and mark the entry as clean in the L2 (see note).
dcbi	yes	n/a	Invalidate the entry in the L1 cache.	Invalidate the entry in the L1 cache. Send address (no data) down the write pipeline to the L2. Assert signals to inform L2 cache that this operation is a dcbi. L2 searches for the address and invalidates any valid L2 entry that matches the address (see note).
dcbi	no	n/a	Do nothing else.	Send address (no data) down the write pipeline to the L2. Assert signals to inform L2 cache that this operation is a dcbi. L2 searches for the address and invalidates any valid L2 entry that matches the address (see note).

Note: dcbf: If the cache line block containing the byte addressed by EA is in storage that is Memory coherency required and WL1=1, a cache line block containing the byte addressed by EA of any locations of L2 cache line is considered to be modified, and the cache line will be written to memory and invalidated.

Note: dcbi: If the cache line block containing the byte addressed by EA is in storage that is Memory coherency required and WL1=1, a cache line block containing the byte addressed by EA of any locations of L2 cache line will be invalidated.

Note: dcbst: If the cache line block containing the byte addressed by EA is in storage that is Memory coherency required and WL1=1, a cache line block containing the byte addressed by EA of any locations of L2 cache line is considered to be modified, and the byte in L2 will be updated with L1 data. In addition, the L2 cache line including the L1 data byte will be written to memory, and the L2 cache line will be marked as clean.

6.3.4.1 dcread Operation

The **dcread** instruction can be used to directly read both the tag information and a specified data word in a specified entry of the data cache. The data word is read into the target GPR specified in the instruction encoding, while the tag information is read into a pair of SPRs, Data Cache Debug Tag Register High (DCDBTRH) and Data Cache Debug Tag Register Low (DCDBTRL). The tag information can subsequently be moved into GPRs using **mfspr** instructions.

The execution of the **dcread** instruction generates the equivalent of an EA, which is then broken down and used to select a specific data word from a specific cache line. $EA_{0:16}$ are ignored, $EA_{17:22}$ select the way, $EA_{23:26}$ select the set, and $EA_{27:29}$ select the word.

The EA generated by the **dcread** instruction must be word-aligned (that is, $EA_{30:31}$ must be 0); otherwise, it is a programming error and the result is undefined.

If the CCR0[CRPE] bit is set, execution of the **dcread** instruction also loads parity information into the DCDBTRL. Note that the DCDBTRL[DPAR] field, unlike all the other parity fields, loads the *check values* of the parity, instead of the raw parity values. That is, the DPAR field will always load with zeros unless a parity error has occurred, or been inserted intentionally using the appropriate bits in the CCR1. This behavior is an artifact of the hardware design of the parity checking logic.

Execution of the **dcread** instruction is privileged, and is intended for use for debugging purposes only.

Programming Note:

The PPC465 does not support the use of the **dcread** instruction when the DCC is still in the process of performing cache operations associated with previously executed instructions (such as line fills and line flushes). Also, the PPC465 does not automatically synchronize context between a **dcread** instruction and the subsequent **mfspr** instructions that read the results of the **dcread** instruction into GPRs. In order to guarantee that the **dcread** instruction operates correctly, and that the **mfspr** instructions obtain the results of the **dcread** instruction, a sequence such as the following must be used:

```
#If reading by cache line number, set GPR A=0 and load GPR B with a line number; otherwise, load GPR A
with a base address and GPR B with an index.

lis         regA, 0           # load GPR A and GPR B with 0
lis         regB, 0
li          regB, LineNumber  # load GPR B with a cache line number between 0 and 1023
rlwinm      regB, regB,5,0,31 # shift left by 5
msync                         # ensure that all previous cache operations have completed
dcread      regT,regA,regB    # read cache information; the contents of GPR A and GPR B are
                              # added and the result used to specify a cache line index to be read;
                              # the data word is moved into GPR T and the tag information is read
                              # into DCDBTRH and DCDBTRL
isync                         # ensure dcread completes before attempting to read results
mfdcdbtrh   regD              # move high portion of tag into GPR D
mfdcdbtrl   regE              # move low portion of tag into GPR E
```

Figure 6-8. Data Cache Debug Tag Register High (DCDBTRH)

0:23	TRA	Tag Real Address	Bits 0:23 of the lower 32 bits of the 36-bit real address associated with the cache line read by **dcread**.
24	V	Cache Line Valid 0 Cache line is not valid. 1 Cache line is valid.	The valid indicator for the cache line read by **dcread**.
25:27		Reserved	
28:31	TERA	Tag Extended Real Address	Upper 4 bits of the 36-bit real address associated with the cache line read by **dcread**.

Figure 6-9. Data Cache Debug Tag Register Low (DCDBTRL)

0:12		Reserved	
13	UPAR	U bit parity UPAR = U0 ⊕ U1 ⊕ U2 ⊕ U3	The parity for the U0-U3 bits in the cache line read by **dcread** if CCR0[CRPE] = 1, otherwise 0.
14:15	TPAR	Tag parity Bit 14 - XOR of odd address bits Bit 15 - XOR of even address bits	The parity for the tag bits in the cache line read by **dcread** if CCR0[CRPE] = 1, otherwise 0. TPAR bit 14 = XOR(DCDBTRH[TERA29,31], DCD-BTRH[TRA1,3...21, 23]) TPAR bit 15 = XOR(DCDBTRH[TERA28,30], DCD-BTRH[TRA0,2...20, 22])
16:19	DPAR	Data parity error Bit 16 - Doubleword 0 Data Parity Error Bit 17 - Doubleword 1 Data Parity Error Bit 18 - Doubleword 2 Data Parity Error Bit 19 - Doubleword 3 Data Parity Error	Each DPAR bit represents a doubleword in the cache line. When any bit in DPAR is set to 1, there is a parity error. The parity *check values* for the data bytes in the word read by **dcread** if CCR0[CRPE] = 1, other-wise 0. Doubleword 0 - Address 0xXXXXXX00 Doubleword 1 - Address 0xXXXXXX08 Doubleword 2 - Address 0xXXXXXX10 Doubleword 3 - Address 0xXXXXXX18
20:23	MPAR	Modified (dirty) parity Bit 20 - Dirty bit Parity for Doubleword 0 Bit 21 - Dirty bit Parity for Doubleword 1 Bit 22 - Dirty bit Parity for Doubleword 2 Bit 23 - Dirty bit Parity for Doubleword 3	The parity for the modified (dirty) indicators for each of the four double words in the cache line read by **dcread** if CCR0[CRPE] = 1, otherwise 0.
24:27	D	Dirty Indicators Bit 24 - Dirty bit for Doubleword 0 Bit 25 - Dirty bit for Doubleword 1 Bit 26 - Dirty bit for Doubleword 2 Bit 27 - Dirty bit for Doubleword 3	The “dirty” (modified) indicators for each of the four double words in the cache line read by **dcread**.
28	U0	U0 Storage Attribute	The U0 storage attribute for the memory page associated with this cache line read by **dcread**.
29	U1	U1 Storage Attribute	The U1 storage attribute for the memory page associated with this cache line read by **dcread**.
30	U2	U2 Storage Attribute	The U2 storage attribute for the memory page associated with this cache line read by **dcread**.
31	U3	U3 Storage Attribute	The U3 storage attribute for the memory page associated with this cache line read by **dcread**.

6.3.4.2 Data Cache Parity Operations

The data cache contains parity bits and multi-hit detection hardware to protect against soft data errors. Both the data cache tags and data are protected. Data cache lines consist of a tag field, 256 bits of data, 4 modified (dirty) bits, 4 user attribute (U) bits, and 39 parity bits. The tag field is stored in CAM (Content Addressable Memory) cells, while the data and parity bits are stored in normal RAM cells. The data cache is physically tagged and indexed, so the tag field contains a real address that is compared to the real address produced by the translation hardware when a load, store, or other cache operation is executed. The exact number of effective address bits depends on the specific cache size.

Two types of errors are detected by the data cache parity logic. In the first type, the parity bits stored in the RAM array are checked against the appropriate data in the RAM line any time the RAM line is read. The RAM data may be read by an indexed operation such as a reload dump (RLD), or by a CAM lookup that matches the tag address, such as a load, **dcbf**, **dcbi**, or **dcbst**. If a line is to be cast out of the cache due to replacement or in response to a **dcbf**, **dcbi**, or **dcbst**, and is determined to have a parity error of this type, no effort is made to prevent the erroneous data from being written onto the L2C. However, the write data on the L2C interface is accompanied by a signal indicating that the data has a parity error.

The second type of parity error that may be detected is a multi-hit, also referred to as an MHIT. This type of error may occur when a tag address bit is corrupted, leaving two tags in the memory array that match the same input. This type of error may be detected on any CAM lookup cycle, such as for stores, loads, **dcbf**, **dcbi**, **dcbst**, **dcbt**, **dcbtst**, or **dcbz** instructions. Note that a parity error will *not* be signaled as a result of an **dcread** instruction.

If a parity error is detected and the MSR[ME] is asserted, (i.e. Machine Check interrupts are enabled), the processor vectors to the Machine Check interrupt handler. As is the case for any machine check interrupt, after vectoring to the machine check handler, the MCSRR0 contains the value of the oldest "uncommitted" instruction in the pipeline at the time of the exception and MCSRR1 contains the old (MSR) context. The interrupt handler is able to query Machine Check Status Register (MCSR) to find out that it was called due to a D-cache parity error, and is then expected to either invalidate the data cache (using **dccci**), or to invoke the OS to abort the process or reset the processor, as appropriate. The handler returns to the interrupted process using the **rfmci** instruction.

If the interrupt handler is executed before a parity error is allowed to corrupt the state of the machine, the executing process is *recoverable*, and the interrupt handler can just invalidate the data cache and resume the process. In order to guarantee that all parity errors are recoverable, user code must have two characteristics: first, it must mark all cacheable data pages as "write-through" instead of "copy-back." Second, the software-settable bit (CCR0[PRE]) must be set. This bit forces all load instructions to stall in the last stage of the load/store pipeline for one cycle, but only if needed to ensure that parity errors are recoverable. The pipeline stall guarantees that any parity error is detected and the resulting Machine Check interrupt taken before the load instruction completes and the target GPR is corrupted. Setting CCR0[PRE] degrades overall application performance. However, if the state of the load/store pipeline is such that a load instruction stalls in the last stage for some reason unrelated to parity recovery, then CCR0[PRE] does not cause an additional cycle stall.

Note that the Parity exception type Machine Check interrupt is *asynchronous*; that is, the return address in the MCSRR0 does not necessarily point at the instruction address that detected the parity error in the data cache. Rather, the Machine Check interrupt is taken as soon as the parity error is detected, and some instructions in progress may get flushed and re-executed after the interrupt, just as if the machine were responding to an external interrupt.

MCSR[DCSP] and MCSR[DCFP] indicate what type of data cache operation caused a parity exception. One of the two bits will be set if a parity error is detected in the data cache, along with MCSR[MCS]. See *Machine Check Interrupts* on page 249.

MCSR[DCSP] is set if a parity error is detected during these search operations:

1. Multi-hit parity errors on any instruction that does a CAM lookup
2. Tag or data parity errors on load instructions

3. Tag parity errors on **dcbf**, **dcbi**, or **dcbst** instructions

MCSR[DCFP] is set if a parity error is detected during these flush operations:

1. Data, dirty, or user parity errors on **dcbf** or **dcbst** instructions
2. Tag, data, dirty, or user parity errors on a line that is cast out for replacement

6.3.4.3 D-Cache Parity Error Recovery Algorithm

The following recovery algorithm assumes the D-Cache is configured for write-back and CCR0[PRE]=1. When the D-Cache is configured for write-through and CCR0[PRE]=1, this algorithm is not needed.

Step 1. Using the DCDBTRL search the content of the D-Cache to find the parity error.

Step 2. Determine the best way to recover from the parity error. The DCDBTRL[UPAR, TPAR, MPAR, DPAR, D] bitfields indicate where in the cache line the error occurred and if the data is dirty. When the data is dirty, the system may not be able to recover depending on the location of the error. If the data is not dirty, it is often possible to recover by writing the known good data back to cached memory.

```
if( (TAG Parity Error OR DPAR>0 OR U bit Parity Error) AND (D>0 OR MPAR!=D) ) {
        Reboot. The cache line is dirty and contains a parity error. Rebooting is the only recovery.
}
else {
        Perform the following steps to recover from a parity error in an unmodified cacheline:
     1. Touch 32KB of cacheable memory into the data cache using the DCBT instruction. This step is necessary
        to avoid having to calculate what the effective address is for each cache line. It also forces most of the dirty
        cache lines to be flushed.

     2. Flush all cache lines in the D-cache using the dcbf instruction with the effective addresses generated by
        step 1. There is a chance some of the addresses used in step 1 were already in the D-cache. This step
        ensures all dirty lines have been flushed.

     3. Invalidate the D-cache with the DCCCI instruction. This step is necessary if the parity error is in the tag.
}
```

DCDBTRL[DPAR] are check bits indicating a parity error. DCDBTRL[MPAR, TPAR, UPAR] are parity bits. To determine if there is a TAG, U storage bit or dirty bit parity error, software must calculate the parity error.

TAG Parity Error Calculation:

To determine whether there is a TAG parity error, software must calculate the even TAG bit parity and odd TAG bit parity from the DCDBTRH[TRA] bit field and then compare these two parity bits with parity bits DCDBTRL[TPAR].

Odd TAG bit parity = DCDBTRH bit1 ⊕ DCDBTRH bit3 ⊕ DCDBTRH bit5 ⊕ ..... ⊕ DCDBTRH bit19 ⊕ DCDBTRH bit21 ⊕ DCDBTRH bit23

Even TAG bit parity = DCDBTRH bit0 ⊕ DCDBTRH bit2 ⊕ DCDBTRH bit4 ⊕ ..... ⊕ DCDBTRH bit18 ⊕ DCDBTRH bit20 ⊕ DCDBTRH bit22

If (Odd TAG bit parity) != (DCDBTRL bit 14), then there is a TAG parity error.

If (Even TAG bit parity) != (DCDBTRL bit 15), then there is a TAG parity error.

U bit Parity Error Calculation:

When UPAR != U0 ⊕ U1 ⊕ U2 ⊕ U3, then there is a U storage bit parity error.

6.3.4.4 Simulating Data Cache Parity Errors for Software Testing

Because parity errors occur in the cache infrequently and unpredictably, it is desirable to provide users with a way to simulate the effect of an data cache parity error so that interrupt handling software may be exercised. This is exactly the purpose of the CCR1[DCDPEI], CCR1[DCTPEI], CCR1[DCUPEI], CCR1[DCMPEI],and CCR1[FCOM] fields.

The 39 data cache parity bits in each cache line contain one parity bit per data byte (i.e. 32 parity bits per 32 byte line), plus 2 parity bits for the address tag (note that the valid (V) bit, is *not* included in the parity bit calculation for the tag), plus 1 parity bit for the 4-bit U field on the line, plus a parity bit for *each* of the 4 modified (dirty) bits on the line. (There are two parity bits for the tag data because the parity is calculated for alternating bits of the tag field, to guard against a single particle strike event that upsets two adjacent bits. The other data bits are physically interleaved in such a way as to allow the use of a single parity bit per data byte or other field.) All parity bits are calculated and stored as the line is initially filled into the cache. In addition, the data and modified (dirty) parity bits (but not the tag and user parity bits) are updated as the line is updated as the result of executing a store instruction or **dcbz**.

Usually parity is calculated as the even parity for each set of bits to be protected, which the checking hardware expects. However, if any of the CCR1[DCTPEI] bits are set, the calculated parity for the corresponding bits of the tag are inverted and stored as odd parity. Likewise, if the CCR1[DCUPEI] bit is set, the calculated parity for the user bits is inverted and stored as odd parity. Similarly, if the CCR1[DCDPEI] bit is set, the parity for any data bytes that are written, either during the process of a line fill or by execution of a store instruction, is set to odd parity. Then, when the data stored with odd parity is subsequently loaded, it will cause a Parity exception type Machine Check interrupt and exercise the interrupt handling software. The following pseudocode is an example of how to use the CCR1[DCDPEI] field to simulate a parity error on byte 0 of a target cache line:

```
dcbt <target line address>   ; get the target line into the cache
msync                        ; wait for the dcbt
mtspr CCR1, Rx               ; Set CCR1[DCDPEI]
isync                        ; wait for the CCR1 context to update
stb <target byte address>    ; store some data at byte 0 of the target line
msync                        ; wait for the store to finish
mtspr CCR1, Rz               ; Reset CCR1[ICDPEI0]
isync                        ; wait for the CCR1 context to update
lb <byte 0 of target line>   ; load byte causes interrupt
```

If the CCR1[DCMPEI] bit is set, the parity for any modified (dirty) bits that are written, either during the process of a line fill or by execution of a store instruction or **dcbz**, is set to odd parity. If the CCR1[FFF] bit is also set in addition to CCR1[DCMPEI], then the parity for all four modified (dirty) bits is set to odd parity. Store access to a cache line that is already in the cache and in a memory page for which the write-through storage attribute is set does not update the modified (dirty bits) nor the modified (dirty) parity bits, so for these accesses the CCR1[DCMPEI] setting has no effect.

The CCR1[FCOM] (Force Cache Operation Miss) bit enables the simulation of a multi-hit parity error. When set, it will cause an **dcbt** to appear to be a miss, initiating a line fill, *even if the line is really already in the cache*. Thus, this bit allows the same line to be filled to the cache multiple times, which will generate a multi-hit parity error when an attempt is made to read data from those cache lines. The following pseudocode is an example of how to use the CCR1[FCOM] field to simulate a multi-hit parity error in the data cache:

```
mtspr CCR0, Rx               ; set CCR0[GDCBT]
dcbt <target line address>   ; this dcbt fills a first copy of the target line, if necessary
```

```
msync                          ; wait for the fill to finish
mtspr CCR1, Ry                 ; set CCR1[FCOM]
isync                          ; wait for the CCR1 context to update
dcbt <target line address>     ; fill a second copy of the target line
msync                          ; wait for the fill to finish
mtspr CCR1, Rz                 ; reset CCR1[FCOM]
isync                          ; wait for the CCR1 context to update
br <byte 0 of target line>     ; load byte causes interrupt
```

6.3.4.5 D-Cache Invalidation Algorithm

The following algorithm invalidates the L1 cache:

1. Touch 32KB of cacheable memory into the data cache using the dcbt or lwz instruction. This step is necessary to avoid having to calculate the effective address used for each cache line. It also forces most of the dirty cache lines to be flushed from the cache. When using dcbt instruction, set CCR0[GDCBT]=1.
2. Flush all cache lines in the D-cache using the dcbf instruction with the effective addresses generated by step 1. There is a chance some of the addresses used in step 1 were already in the D-cache. This step ensures all dirty lines have been flushed.

Note: Interrupts must be disabled when invalidating the D-Cache.

7. Level 2 Cache

This chapter discusses the Level 2 cache, debug and processor local bus operations. Some related aspects of Level 2 cache are also discussed in various other chapters of the book as needed.

7.1 L2 Cache Operations

The PPC465L2AC6 core provides an unified Level-2 cache, and contains a function that manages the transfers between PPC465 CPU core caches and the Processor Local Bus (PLB). The storage capacity of the cache arrays supported are 0 and 256KB L2 Array sizes via configuration bits.

The L2C Array has a 128 Byte line size, which is equivalent of 8 data beats of PLB transfer, and is structured as 4-way set associative cache being managed by a true Least-Recently-Used, LRU, replacement algorithm. The L2C controller interfaces to the First-Level L1 Instruction cache and Data cache, and to the PLB(v5).

The L2C implements multi-processor hardware support for memory coherency. ECC protection of the tag and data arrays provides high reliability. The L2 cache can also be configured as fast, on-chip fixed addressable memory.

7.1.1 L2 Cache TAG and Data Array Organization

Each line of the L2C data arrays is segmented into four 32-Byte granules. The state of each of these granules is independent of the state of the other three granules. However, the associativity of the L2 cache is fixed at 4-way. The L2 cache has 512 sets and a fixed line size of 128 bytes.

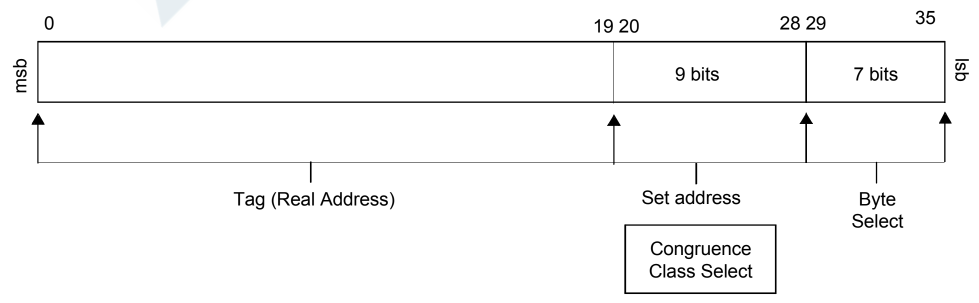
7.1.1.1 TAG Organization

Each TAG array entry contains page attributes (M-bit, G-bit and Ubits), the line state (Shared/Modified), and ECC for the TAG entry along with TAG address to support MESI protocol and L2C TAG ECC bits. See Figure 7-4 and *Table 7-2* in the ECC section for the Tag bit positions.

The L2C is addressed using a 36-bit real address. The real addresses are organized into the TAG portions, “ways”, and “sets” (or “congruence class select”) as illustrated in *Table 7-1.*

The 0 to 35 bit range in the TAG diagram corresponds to the real address bits 28:63 on the PLB bus.

Figure 7-1. Address Breakdown for 256KB L2



7.1.1.2 L2 Cache Data Array Organization

Each 128 Bytes cache line of the L2C data cache lines are segmented into four 32-Byte granules. The state of each of these granules is independent of the state of the other three granules.

7.1.2 L2 Cache Controller

The L2 cache controller (L2CC) handles the data transactions between the PPC465 core, L1core Instruction and Data caches, and the memory system via PLB. A 128-bit read interface is provided for the L1 core Instruction cache, and two independent 128-bit read and write interfaces are provided for the L1 data cache. For the PLB, there are two independent separate sets of interfaces, a Master and a Slave, each having independent 128-bit read and write interfaces.

The L2C is attached to the L1 core interfaces through synchronization logic The 'frequency synchronization ratio between the PPC465 core and the L2CC's logic must be set for the L2C interfaces to operate properly. The L2CC synchronization logic supports N:1, processor core to L2C frequency ratios, where N= 2, 3 or 4.

The L2CC also handles two separate DCR buses of 32-bits: one for DCR reads and other for DCR writes, from/to the PPC465 core. It also has a snoop bus running between the L2C and the PPC465 core and a snoop interface port on the PLB.

7.1.2.1 L1 Core Instruction and Data caches Interfaces

Since the L2C is the Level-2 cache for the processor, the L2C will allocate a cache line based on L1core operations/requests. In general, the main L1core operations that L2C allocates lines are:

- L1core Instruction cache read-miss request that is L2 cacheable,
- L1core Data cache read-miss request that is L2 cacheable,
- L1core Data cache store-miss with allocate.

The L2C is also designed to handle full memory coherence as long as the L1 Data cache is set for write-thru mode. The TLB storage attributes, such as M (memory coherence required) and the cache inhibited bits change the mode of PLB data transaction. The cache lines of Memory coherence required storage space will be allocated in the L2C based on the L1 core request commands and/or attributes.

The L2 operations are also affected by L2 Configuration Register 1, L2CR1, such as "Touch Allocation Size", L2CR1[TAS], "Read Allocation Size", L2CR1[RAS], and "Fixed Address Region mode", L2CR1[FAR], that are provided for performance improvements and conveniences for software applications. Some cacheable store misses are handled as *store without allocate, which is denoted by **SWOA**.*

There are store buffers provided at the L1D store interface to prevent (or alleviate) L1core stalls due to store operations in L1core. This will provide some elasticities between L1D and L2C in the store path in addition to L1D store-gathering.

Overall, the L2C will provide up to four pending read or write misses from L1 core.

7.1.2.2 L1 Core Request Priority

Priority for L1core requests to L2C resources is based on age and cacheability. Non-cacheable requests will be allocated L2C resources and ordered to the PLB based on relative age between other non-cacheable requests. Cacheable requests will be ordered to the array and subsequently the PLB based on relative age between other cacheable requests.

7.1.2.3 PLB Interactions

The L2C is implemented to handle up to four outstanding read requests to the PLB and pass data to L1core from PLB in the cases of L2C misses, and can handle an L2C hit with up to four pending read or write misses of L1 core. These functions provide a non-blocking cache capability to improve L1 core misses. The functions also provide an additional advantage during PLB reads by bypassing L2C array and transferring data directly to/from L1core for L1core read of L2C missed data, or cache inhibited data transfers.

7.1.3 L2 Cache Operations For L1 Cache Management Ops

The PPC465 core provides various instructions to control instruction and data cache operations to help debug L1 cache problems and/or enhance cache operations. These operations are also reflected in the L2C to maintain data integrity in the system.

The detailed descriptions of the instructions are summarized in See “Instruction Cache Controller” on page 131. and See “Data Cache Controller” on page 139.. In the instruction descriptions, the term “block” describes the unit of storage operated on by the cache block instructions. For the PPC465 core, this is the same as a cache line.

7.1.3.1 iccci and icbi

Neither the iccci or the icbi have any effect on the L2 Cache. The iccci performs a flash invalidate on the L1 cache and the icbi does a line invalidation in the L1 cache. To invalidate an instruction line that was loaded into the L2 from an i-fetch, a dcbi instruction must be used.

7.1.3.2 icbt

The icbt instruction is a “hint” that performance will probably be improved if the block containing the byte addressed is fetched into the cache system. Their behaviors are affected by the existence of an L2C and the cacheability attribute of the address in the L1 and L2 caches.

The L2C fills the cache normally and sends the instructions to L1 core when all levels of caches are **not inhibited**. The L2C also uses the touch indicators to enable filling locked lines.

When L1 instruction cache is inhibited, indicated by page attribute “IL1I” being set, the L2C fills the cache, but no data will be returned to L1 instruction cache.

In the above cases, **icbt** causes the touched cache line to be either “Shared” or “Exclusive” depending upon whether other processors have the same cache line (sharing) or not.

When caches are inhibited, indicated by page attribute “I bit” or “IL2I bit” being set, no touch operations will be performed.

7.1.3.3 dccci

The **dccci** performs a flash invalidate on the L1 Data cache and does not have any effect on the L2 Cache. To invalidate the complete L2 cache, the L2ccci function of the L2COPR register must be performed on each congruence class.

7.1.3.4 dcbi

When the L2C accepts a dcbi from the L1core data cache, L1D, the L2C will always perform an array lookup on the 32-Byte granule address, which is the L1D line size, regardless of cacheability. If the cache line block containing the byte addressed by EA is in storage that is Memory coherency required and the page Attribute WL1=1, a cache line block containing the byte addressed by EA of any locations of L1 data cache and L2C cache line will be invalidated.

If the page is marked Memory Coherence required, the L2c will also broadcast the dcbi over the PLB to any other processors on the bus. This will cause the other processors to invalidate the line in their L1 and L2 caches.

If the storage space is in the Fixed-Address-Region or non-coherent, the above process will not take place; the hardware does not support Memory coherency in those spaces.

7.1.3.5 dcbf

When the L2C accepts a dcbf from the L1D, the L2C will always perform an array lookup for the 32-Byte granule address, regardless of cacheability. If the cache line block containing the byte addressed by EA is in storage that is Memory coherency required and the page Attribute WL1=1, a cache line block containing the byte addressed by EA of any locations of L2 cache line and L1D line in transit (L1 data cache is in Writethrough) are considered to be modified, and the cache line will be written to memory and then invalidated in the L2 cache. The L2C will send a request for dcbf to the PLB to flush/invalidate the cache line in other processors on the bus if the page is Memory Coherence required (M=1).

In the Fixed Address Region and non-coherent storage space, the above operation will not take place; the hardware does not support Memory coherency for those spaces.

7.1.3.6 dcbst

When the L2C accepts a dcbst from the L1 data cache, the L2C will always perform an array lookup for the 32-Byte granule address, regardless of cacheability. If the cache line block containing the byte addressed by the EA is in storage that is Memory coherency required and the page attribute WL1=1, a cache line block containing the byte addressed by the EA is in L2 cache and L1 data cache line in transit (L1 data cache is in Writethrough) are considered to be modified, and the byte in L2C will be updated with the L1 data. In addition, the L2 cache line including the L1 data byte will be written to memory, and the L2 cache line will be marked as clean.

In the Fixed Address Region and non-coherent storage space, the above operation will not take place; the hardware does not support Memory coherency for those spaces.

7.1.3.7 dcbt, dcbtst

Instructions dcbt and dcbtst are “hints” that performance will probably be improved if the block containing the byte addressed is fetched into the cache system. Their behaviors are affected by the existence of an L2C and the cacheability status of the data access in the L1 and L2 caches.

The L2C fills the cache normally and sends the data to L1 dcache when all levels of caches are **not inhibited**. The L2C also uses the touch indicators to enable filling locked lines.

When L1 data cache is inhibited, indicated by page attribute “IL1D” being set, the L2C fills the cache, but no data will be returned to L1 data cache.

In the above cases, **dcbt** causes the touched cache line to be either “Shared” or “Exclusive” depending upon whether other processors have the same cache line (sharing) or not. On the other hand, **dcbtst** will always result in “Exclusive” ownership.

When caches are inhibited, indicated by page attribute “I bit” or “IL2D bit” being set, no touch operations will be performed.

See *dcbt and dcbtst Operation* on page 136 for more details on these instructions.

7.1.3.8 L2ccci - L2 Cache Congruence Class Invalidate

L2ccci is not an instruction but is a operational sequence built into the L2 Cache controller. The L2C is able to invalidate a particular congruence class via a mtdcr to L2COPR with bit 0 set to a ‘1’. Upon detecting this mtdcr, the L2 will perform an L2ccci to the congruence class specified by bits [n:24] of the L2COPR, where n is based on cache size. The following table illustrates which bits of the L2COPR are used.

Table 7-1. L2COPR Fields Based for L2ccci

Array Size	L2ccci bit	Reserved	Congruence Class	Reserved
256 KB	0	1: 15	16: 24	25: 31

Further details of L2COPR for “L2ccci” are found in the L2COPR register description section.

The L2C will write a ‘0’ to all four valid bits for all four ways during an L2ccci. The rest of the tag and state bits will be undefined after the L2ccci.

The L2C will with-hold the DCR acknowledge signal from the L1core until the L2ccci has completed. The main usage for this instruction will be for software to invalidate a cache line that has a double bit ECC error on it. Upon reporting the double bit error, the L2C will write the congruence class that the error occurred in to the L2ECCIDX[9:24]. Software can move the L2ECCIDX to a GPR to be used as the source for the mtdcr operation. See more details on L2COPR in Figure 7-19, on page 187.

An L2ccci causes the LRU entry at the congruence class of the L2ccci to be reset to the following sequence from least recently used to most recently used - Way 0, Way 1, Way 2, Way 3.

7.1.4 L2 Cache Hazards: L1 Core Storage Reference Ordering

The L2C handles certain hazards in order to support the BookE sequential execution model. Requests are allowed to be serviced out of order in certain instances. If two requests do not have matching addresses and are not separated by an msync or mbar, then the L2C is allowed to service these requests in any order. The msync and mbar are handled in the PPC465 core, which holds any subsequent storage referencing operations until the L2C completes all outstanding storage references, at least those ownership processes, by Address-Acknowledges, AAck. This leaves only the matching address case that must be handled by the L2C. Note that matching address between the L1core Data cache write and the L1core Instruction cache read are not taken into consideration because these would be self-modifying code sequences that require an msync to preserve the correct program order.

The L1core Data cache must make requests in order between the read and write interfaces. A read request and write request to the same cache line cannot be made in the same cycle unless the ordering does not matter to the L1core, or the ordering is ensured by L1core.

Once the requests are in the L2C, the L2 tracks requests (L1core requests, snoop castouts, castouts) by age information to maintain proper ordering.

Non-cacheable L1core requests are ordered to the desired fill buffer / store buffer in the following manner. If the L1D Write-Queue and L1D Read-Queue become valid, the newer request will be held in its respective queue. However, if both queues are valid at the same time, the newer request will be held only if a collision, the same cache line requests, occurs. Note that this allows non-cacheable requests to be presented to the PLB in a different order than they were received by the L2C. L1D write requests must always use the Write- Queue but L1D read requests are allowed to bypass the L1D Read-Queue under certain conditions. If the Write-Queue is valid with a non-cacheable request, non-cacheable L1core read requests are not allowed to bypass the L1D Read-Queue if empty (request is forced into L1D Read-Queue). No special block is performed if either request is cacheable.

7.1.5 Error Detection/Correction

The L2C contains ECC logic for correcting single bit errors and detecting double bit errors on both the tags and data arrays. The data arrays will contain eight parity/ecc bits per 64 bits of data. The tag arrays will contain eight parity/ecc bits for the entire tag of 38 bits.

See L2CR0[DECC] and L2CR0[DECA] for enabling ECC and ECC single bit correction control.

See MCSR L2MCSR and L2MCRER DCR registers for ECC error status and control.

The L2C uses the Hamming Code scheme for error detection and correction. The Hamming Code scheme provides an avenue for inexpensive implementation of error control.

7.1.5.1 Number of Parity Bits

The Hamming rule states that the minimum number of parity bits (p) needed for a given data packet size (d) is provided by the following relation.

$$d + p + 1 \leq 2^{p}$$

The minimum number of parity bits yields either Single Error Correction (SEC) or Double Error Detection (DED), but not both. An additional parity bit is needed in order to obtain both SEC and DED. The L2C will use 8 bits for encoding a 64 bit data word. This choice will provide both SEC and DED for data within the L2C. The L2C will use 8 bits in order to encode each of the 59 bit tags. This choice of checkbits will also provide the L2C with SEC and DED on the cache tags. The result of combining the data bits with the parity bits is defined as the codeword.

7.1.5.2 Encode

Encoding occurs via use of matrix multiplication. While encoding, the data bits are left unchanged. The parity bits, however, are formed as a result of an encoding process. The following example illustrates a possible encode (ENC) matrix.

Figure 7-2. ECC Encode Matrix

$$\mathrm{ENC} = \overbrace{\begin{bmatrix} 1&0&0&0&0&0&0&1&1&1&1 \\ 0&1&0&0&0&0&1&0&1&1&1 \\ 0&0&1&0&0&0&1&1&0&1&0 \\ 0&0&0&1&0&0&1&1&1&0&1 \\ 0&0&0&0&1&0&1&1&0&0&1 \\ 0&0&0&0&0&1&0&0&1&1&0 \end{bmatrix}}^{\text{d+p columns}} \left.\vphantom{\begin{bmatrix}1\\1\\1\\1\\1\\1\end{bmatrix}}\right\} \text{d rows}$$

The matrix above is obtained by combining an identity matrix with a randomly generated parity matrix. In the case of the encode matrix, the identity portion is placed on the left side and the parity portion is placed on the right side of the matrix. Parity matrices must have an identical number of 1s in each column. In addition, no row or column may contain all 1s or 0s. The example above contains four 1s in each column; therefore, an even parity matrix is present. The encode matrix should have one row for each bit in the data package (d) and one column for each bit in the codeword (d + p). The encode matrix illustrated above may be used in order to produce a codeword for a given data package. The following example illustrates how the word 110111 may be encoded into a codeword:

Figure 7-3. 5-Bit ECC Encoding Example

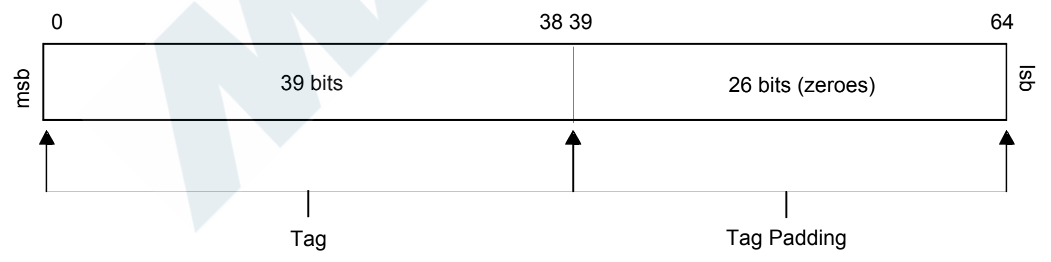
$$
\begin{bmatrix} 1\,1\,0\,1\,1\,1 \end{bmatrix} \times
\begin{bmatrix}
1\,0\,0\,0\,0\,0\,0\,1\,1\,1\,1 \\
0\,1\,0\,0\,0\,0\,1\,0\,1\,1\,1 \\
0\,0\,1\,0\,0\,0\,1\,1\,0\,1\,0 \\
0\,0\,0\,1\,0\,0\,1\,1\,1\,0\,1 \\
0\,0\,0\,0\,1\,0\,1\,1\,0\,0\,1 \\
0\,0\,0\,0\,0\,1\,0\,0\,1\,1\,0
\end{bmatrix}
=
\begin{bmatrix}
\text{XOR }(1,0,0,0,0,0) \\
\text{XOR }(0,1,0,0,0,0) \\
\text{XOR }(0,0,0,0,0,0) \\
\text{XOR }(0,0,0,1,0,0) \\
\text{XOR }(0,0,0,0,1,0) \\
\text{XOR }(0,0,0,0,0,1) \\
\text{XOR }(0,1,0,1,1,0) \\
\text{XOR }(1,0,0,1,1,0) \\
\text{XOR }(1,1,0,1,0,1) \\
\text{XOR }(1,1,0,0,0,1) \\
\text{XOR }(1,1,0,1,1,0)
\end{bmatrix}
\begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}
\begin{array}{l}
\\ \\ \text{Data Bits} \\ \\ \\ \\ \hdashline \\ \\ \\ \text{Check Bits} \\ \\
\end{array}
$$

Therefore, the data package 110111 becomes 11011111010 when encoded. The first d bits of the encoded version are identical to the data package and the last p bits represent the ECC generated for the data package.

Tag ECC Encoding

The ECC bits for the tag array will also be created using the matrix from Figure 7-2, although there is one significant difference. Since the tag array contains only 39 bits of information to be encoded, an additional mechanism must be added in order for the tag array to be compatible with the 64 x 72 encode matrix. This is accomplished conceptually by padding the 26 LSB's of the tag with zeroes. Note that this is not physically done in hardware, it is only done theoretically so that the correct matrix multiplication can be accomplished. Therefore, the check bit generation for the tag array will ignore any data bit sampling which occurs after bit 38 of the tag. Figure 7-1 details which tag bits are sampled for each ECC bit.

Figure 7-4. Tag Packet to be Encoded



Each Tag entry packet depicted in *Figure 7-4* is defined as follows:

Table 7-2. TAG Packet Definition

Bit(s)	Field Assignments
0-20	Tag Address
21	Memory coherency required
22	Guarded
23-26	User Bits
27-30	Valid [granule0,1, 2, 3]
31-34	Shared [granule0,1, 2, 3]
35-38	Modified [granule0,1, 2, 3]

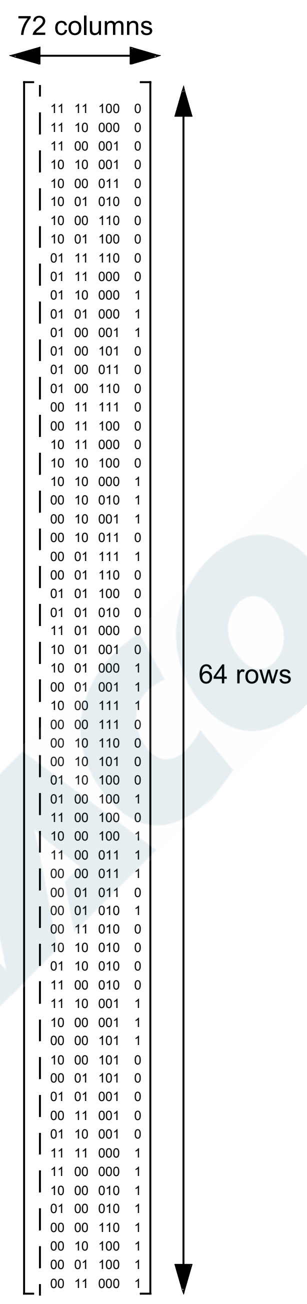
Table 7-3. Tag Bits XORed for Tag ECC Bit Generation

ECC Bit	Tag Bit Locations XORed for Tag ECC Bits
0	0,1,2,3,4,5,6,7,18,19,20,28,29,30,32,38
1	0,1,2,8,9,10,11,12,13,14,15,26,27,28,36,37,38
2	0,1,3,8,9,10,16,17,18,19,20,21,22,23,34,35,36
3	0,5,7,8,9,11,16,17,18,24,25,26,27,28,29,30,31
4	0,6,7,8,13,15,16,17,19,24,25,26,32,33,34,35,36,37,38
5	4,5,6,8,14,15,16,21,23,24,25,27,32,33,34
6	2,3,4,12,13,14,16,22,23,24,29,31,32,33,35
7	10,11,12,20,21,22,24,30,31,32,37

Data ECC Encoding

The ECC bits for the data array will be generated using the encode matrix detailed in Figure 7-5. Each of the columns in the parity section (non-identity) of the encode matrix contain twenty-six ‘1’s. Therefore, the incoming 64-bit data packet will be sampled at the specific bit locations shown in the matrix. *Table 7-4* details which data bits are sampled for each ECC bit. The following is the encode matrix that will be for the data and tag ECC. Note that I is a 64 X 64 identity matrix.

Figure 7-5. Encode Matrix



```
11 11 100 0
11 10 000 0
11 00 001 0
10 10 001 0
10 00 011 0
10 01 010 0
10 00 110 0
10 01 100 0
01 11 110 0
01 11 000 0
01 10 000 1
01 01 000 1
01 00 001 1
01 00 101 0
01 00 011 0
01 00 110 0
00 11 111 0
00 11 100 0
10 11 000 0
10 10 100 0
10 10 000 1
00 10 010 1
00 10 001 1
00 10 011 0
00 01 111 1
00 01 110 0
01 01 100 0
01 01 010 0
11 01 000 0
10 01 001 0
10 01 000 1
00 01 001 1
10 00 111 1
00 00 111 0
00 10 110 0
00 10 101 0
01 10 100 0
01 00 100 1
11 00 100 0
10 00 100 1
11 00 011 1
00 00 011 1
00 01 011 0
00 01 010 1
00 11 010 0
10 10 010 0
01 10 010 0
11 00 010 0
11 10 001 1
10 00 001 1
00 00 101 1
10 00 101 0
00 01 101 0
01 01 001 0
00 11 001 0
01 10 001 0
11 11 000 1
11 00 000 1
10 00 010 1
01 00 010 1
00 00 110 1
00 10 100 1
00 01 100 1
00 11 000 1
```

Table 7-4. Data Bits XORed for Data ECC Bit Generation

ECC Bit	Data Bit Locations XORed for Data ECC bits
0	0,1,2,3,4,5,6,7,18,19,20,28,29,30,32,38,39,40,45,47,48,49,51,56,57,58
1	0,1,2,8,9,10,11,12,13,14,15,26,27,28,36,37,38,40,46,47,48,53,55,56,57,59
2	0,1,3,8,9,10,16,17,18,19,20,21,22,23,34,35,36,44,45,46,48,54,55,56,61,63
3	0,5,7,8,9,11,16,17,18,24,25,26,27,28,29,30,31,42,43,44,52,53,54,56,62,63
4	0,6,7,8,13,15,16,17,19,24,25,26,32,33,34,35,36,37,38,39,50,51,52,60,61,62
5	4,5,6,8,14,15,16,21,23,24,25,27,32,33,34,40,41,42,43,44,45,46,47,58,59,60
6	2,3,4,12,13,14,16,22,23,24,29,31,32,33,35,40,41,42,48,49,50,51,52,53,54,55
7	10,11,12,20,21,22,24,30,31,32,37,39,40,41,43,48,49,50,56,57,58,59,60,61,62,63

7.1.5.3 Decode

The decode matrix is created by placing the transpose of the parity matrix on the left side and an identity matrix on the right side. Note that the size of the identity matrix used in the decoding scheme does not match the one used in the encoding matrix. The decode matrix should have one row for each parity bit and one column for each codeword bit (i.e., data package size + number of parity bits).

Figure 7-6. ECC Decode Matrix

$$\mathrm{DCD} = \begin{bmatrix} 0&1&1&1&1&0&1&0&0&0&0 \\ 1&0&1&1&1&0&0&1&0&0&0 \\ 1&1&0&1&0&1&0&0&1&0&0 \\ 1&1&1&0&0&1&0&0&0&1&0 \\ 1&1&0&1&1&0&0&0&0&0&1 \end{bmatrix}$$

A procedure similar to encoding may be followed in order to decode and detect/correct errors in the codeword. The following example illustrates how the word above is decoded when no errors are present

Figure 7-7. 5-Bit ECC Decode Example

$$\begin{bmatrix} 0\,1\,1\,1\,1\,0\,1\,0\,0\,0\,0 \\ 1\,0\,1\,1\,1\,0\,0\,1\,0\,0\,0 \\ 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,0 \\ 1\,1\,1\,0\,0\,1\,0\,0\,0\,1\,0 \\ 1\,1\,0\,1\,1\,0\,0\,0\,0\,0\,1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \text{XOR }(0,1,0,1,1,0,1,0,0,0,0) \\ \text{XOR }(1,0,0,1,1,0,0,1,0,0,0) \\ \text{XOR }(1,1,0,1,0,1,0,0,0,0,0) \\ \text{XOR }(1,1,0,0,0,1,0,0,0,1,0) \\ \text{XOR }(1,1,0,1,1,0,0,0,0,0,0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The result of the calculation during decode will always be a matrix containing only 0s when the codeword is correct. In this case, the first d bits of the codeword may be forwarded to the requester. When a single error is present, however, the result of the calculation illustrated above will match one of the columns in the decode matrix. The following illustration demonstrates how a single error may be may be found and corrected.

Figure 7-8. 5-Bit ECC Decode with Single Bit Error

$$\begin{bmatrix} 0\,1\,1\,1\,1\,0\,1\,0\,0\,0\,0 \\ 1\,0\,1\,1\,1\,0\,0\,1\,0\,0\,0 \\ 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,0 \\ 1\,1\,1\,0\,0\,1\,0\,0\,0\,1\,0 \\ 1\,1\,0\,1\,1\,0\,0\,0\,0\,0\,1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \text{XOR }(0,0,0,1,1,0,1,0,0,0,0) \\ \text{XOR }(1,0,0,1,1,0,0,1,0,0,0) \\ \text{XOR }(1,0,0,1,0,1,0,0,0,0,0) \\ \text{XOR }(1,0,0,0,0,1,0,0,0,1,0) \\ \text{XOR }(1,0,0,1,1,0,0,0,0,0,0) \end{bmatrix} == \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Match

The non-zero result of the calculation yields that an error is present. The decode matrix may be consulted in order to find a column matching the bits yielded via calculation. If a match is found, the corresponding column indicates which codeword bit is in error. The second column matches in this case; therefore, bit 1 is incorrect and requires correction. After correction, the first d bits of the codeword may be forwarded to the requester.

The same decode scheme also allows for detection of double-bit errors. This kind of error is present when the calculated result of the above calculation does not match any of the columns in the decode matrix. The following example demonstrates how the algorithm detects the existence of double-bit errors.

Figure 7-9. 5-Bit ECC Decode with Double Bit Error

$$\begin{bmatrix} 0\,1\,1\,1\,1\,0\,1\,0\,0\,0\,0 \\ 1\,0\,1\,1\,1\,0\,0\,1\,0\,0\,0 \\ 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,0 \\ 1\,1\,1\,0\,0\,1\,0\,0\,0\,1\,0 \\ 1\,1\,0\,1\,1\,0\,0\,0\,0\,0\,1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \text{XOR}\ (0,0,0,1,1,0,1,0,0,0,0) \\ \text{XOR}\ (1,0,0,1,1,0,0,1,0,0,0) \\ \text{XOR}\ (1,0,0,1,0,1,0,0,0,0,0) \\ \text{XOR}\ (1,0,0,0,0,1,0,0,0,1,0) \\ \text{XOR}\ (1,0,0,1,1,0,0,0,0,0,1) \end{bmatrix} == \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

The result of the decode process above yields non-zero matrix elements; therefore, an error of some kind is present. Since none of the columns in the decode matrix match, a double-bit error has occurred. This result is correct because bits 1and 7 are in error.

Tag ECC Decoding

The tag ECC decoding sequence is similar to the data ECC decoding. Syndrome bits are generated and are then compared to a known set of vectors. If the syndrome bits match any of the combinations, then a single bit error is present. If the syndrome bits are all zeroes, then the tag is error free. If there is no match in the syndrome bit combinations then there is a double bit error. One difference from the data array decoding is that not all syndrome bits will require the same amount of XORs for generation. This is due to the same circumstances detailed in the "Tag ECC Encoding."

The Tag Array decode packet is shown in Figure 7-10, and Tag bits usage for *Tag Syndrome Bit Generation* is shown in *Table 7-5.*

Figure 7-10. Tag Array Decode Packet

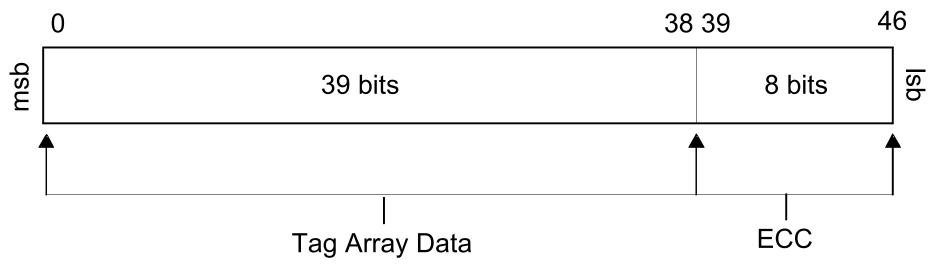


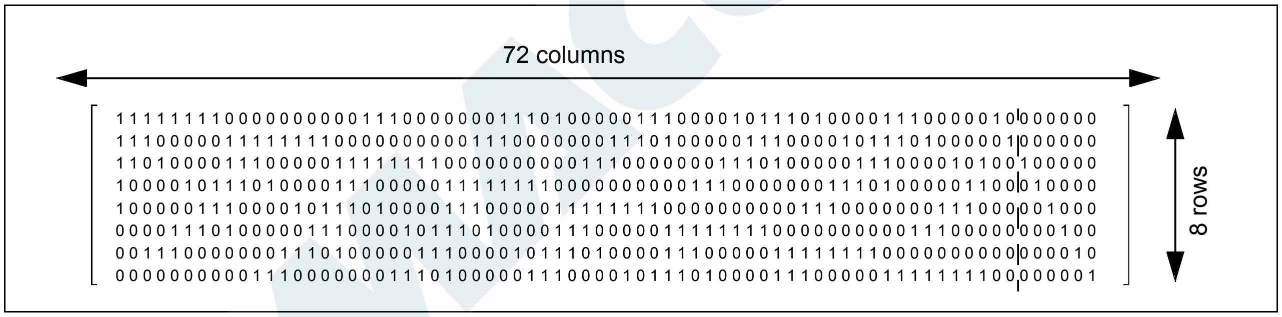
Table 7-5. Bits XORed for Tag Syndrome Bit Generation

Syndrome Bit	Data Bit (DB) and ECC Bit (EB) Locations XORed for Tag Array
0	DB(0,1,2,3,4,5,6,7,18,19,20,28,29,30,32,38),EB(0)
1	DB(0,1,2,8,9,10,11,12,13,14,15,26,27,28,36,37,38),EB(1)
2	DB(0,1,3,8,9,10,16,17,18,19,20,21,22,23,34,35,36),EB(2)
3	DB(0,5,7,8,9,11,16,17,18,24,25,26,27,28,29,30,31),EB(3)
4	DB(0,6,7,8,13,15,16,17,19,24,25,26,32,33,34,35,36,37,38),EB(4)
5	DB(4,5,6,8,14,15,16,21,23,24,25,27,32,33,34),EB(5)
6	DB(2,3,4,12,13,14,16,22,23,24,29,31,32,33,35),EB(6)
7	DB(10,11,12,20,21,22,24,30,31,32,37),EB(7)

Data ECC Decoding

The following matrix is the decode matrix used for the data and tag ECC decoding schemes. As outlined in the Section 7.1.5.3 , “Decode” on page 166, the decode matrix is nothing more than the transpose of the parity portion of the encode matrix with an 8 x 8 identity matrix appended on the right hand side.

Figure 7-11. Decode Matrix



```
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0|0 0 0 0 0 0
1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 0 1|0 0 0 0 0 0
1 1 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 1 1 0 0|0 1 0 0 0 0
1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0|0 0 1 0 0 0
0 0 0 0 1 1 1 0 1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 0 0
0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0|0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0|0 0 0 0 0 1
```

The first step in the data ECC decoding sequence is to calculate the syndrome bits. The syndrome bits are the pieces of data which will be evaluated to check if an error is present and are defined as the result of the matrix multiplication of the decode matrix times the codeword. Twenty seven XORs will be needed for each of the eight syndrome bits. The calculation of the syndrome bits is almost identical to that of the ECC bits. As seen in “Data ECC Decoding” on page 169, there are twenty-six bits of data which need to be XORed together to get each check bit. The calculation of the syndrome bit involves the same XORs used to calculate the ECC bit XORed with the corresponding ECC bit. So each syndrome bit is a result of the following XOR logic.

Table 7-6. Bits XORed for Data Syndrome Bit Generation

Syndrome Bit	Data Bit (DB) and ECC Bit (EB) Locations XORed for Data Array
0	DB(0,1,2,3,4,5,6,7,18,19,20,28,29,30,32,38,39,40,45,47,48,49,51,56,57,58),EB(0)
1	DB(0,1,2,8,9,10,11,12,13,14,15,26,27,28,36,37,38,40,46,47,48,53,55,56,57,59),EB(1)
2	DB(0,1,3,8,9,10,16,17,18,19,20,21,22,23,34,35,36,44,45,46,48,54,55,56,61,63),EB(2)
3	DB(0,5,7,8,9,11,16,17,18,24,25,26,27,28,29,30,31,42,43,44,52,53,54,56,62,63),EB(3)
4	DB(0,6,7,8,13,15,16,17,19,24,25,26,32,33,34,35,36,37,38,39,50,51,52,60,61,62),EB(4)
5	DB(4,5,6,8,14,15,16,21,23,24,25,27,32,33,34,40,41,42,43,44,45,46,47,58,59,60),EB(5)
6	DB(2,3,4,12,13,14,16,22,23,24,29,31,32,33,35,40,41,42,48,49,50,51,52,53,54,55),EB(6)
7	DB(10,11,12,20,21,22,24,30,31,32,37,39,40,41,43,48,49,50,56,57,58,59,60,61,62,63),EB(7)

Once the syndrome bits are calculated, the syndrome bit vector is compared against known combinations which will result in single bit errors. As described in "Decode" on page 166 if the configuration of the syndrome bits matches the configuration of column 0 of the decode matrix (*Figure 7-11*), then there is a single bit error in position 0 of the codeword. If the syndrome bits match the bits of column 1, then there is a single bit error in position 1 of the codeword, etc. If the syndrome bits are all zeroes, then there are no errors present in the codeword. A double bit error is only present when the syndrome bits are not all zeroes and they do not match any of the decode matrix columns. *Table 7-6* details the different combinations that the syndrome bits will be checked against and the corresponding location of the single bit error if there is a match:

If any of the single-bit error combinations match the syndrome bit vector, then an error correcting procedure is invoked. The error correcting algorithm involves XORing the erroneous data with a 64-bit vector that contains a single non-zero entry. The logic '1' entry is located at the same bit location as the single-bit error location in the data. Therefore, when the two vectors are XORed together the single-bit error will be corrected. For example, if the syndrome bits matched the bits corresponding to location 0, then the error-correcting vector would be 0x8000000000000000 (64 bits). For location 1, the vector would be 0x4000000000000000 (64 bits), etc.

7.1.6 Memory Coherence

The PPC465 core, together with the L2c and the PLB5, implement memory coherence for page that are marked Memory Coherence Required (M=1). The protocol to maintain coherence for multiple processors implemented in the system is a "Snooping". This snooping technique tracks the state of any sharing of a data block.

Only the Master PLB(v5) port and its associated snoop port interface supports snooping protocol. The Slave PLB port is used for the public Fixed Address Region and is not memory coherent. The subsequent discussions in this section are all referred to the Master PLB port.

7.1.6.1 Snooping

The L2C provides snoop logic to track all PLB snoop activities by checking the L2C cache line states, forwarding the snoop request to the L1 data cache for the cache line invalidation or the reservation granule invalidation, and responding to the PLB accordingly.

Note: Coherence is supported only for the L1 data cache. The L1 instruction cache is not snooped and software is responsible for maintaining memory coherence.

The snoop address will always be aligned on a 32-byte boundary, and the L2C is implemented to handle up to two snoop requests at any given time. All the snoops are processed in order.

There are four types of data cache related PLB snoop requests that are handled by the L2C, and they are snoop push, snoop pushE, snoop kill, and snoop flush.

The hardware reservation which implements the atomic operation instructions (lwarx and stwcx.), are snooped using the same mechanism. This allows atomic operations between processors. A reservation kill is generated and sent to the L1D as part of the snoop command for all PLB snoops. It is sent with the normal L1D snoop request unless the snoop misses in the L2C. In that case the reservation snoop is sent to L1D after L2C has completed its own snoop processes. Reservation kills are only generated due to ’PLB snoop flush’ or ’PLB snoop kill’ types. A data cache block flush, dcbf instruction, of the reserved cache line by another processor(s) will remove any storage reservation.

Note: This means that atomic operations using the lwarx and stwxc. instructions cannot be used and will fail even if the L2 cache array is configured to be zero size or is configured to be Fixed Address Mode of 256k (all FAM).

Caution: When L2CR2[3] is used to change the protocol to MEI, L2CR2[7], FRS (Force Read Shared) can not be set. This is to avoid a “Livelock” condition. Refer to L2CR2. Similarly, L2CR2[6], CSNPPF (Convert Snoop Push to Flush) can not be set if a proper operation of the storage reservation is required in the system. Refer to L2CR2.

7.1.6.2 Snoop Handling of Fixed-Address-Mode (FAM) space

If the entire array is in Fixed-Address-Mode, no snoop requests are made to the L2 array. However, the L2C resources are still checked if needed, and the request is sent to the L1core. If only half of the array is in FAM, the snoop is handled for that non-FAM region, but FAM region will not be snooped; the address cannot match the FAM region, so no address checking is done. See “L2 Cache Fixed Address Mode (FAM)” on page 177.

Note: No snoop castout will be done in FAM region.

7.1.6.3 MESI Protocol

The Memory coherency/consistency is provided with MESI: Modified (M), Exclusive (E), Shared (S), and Invalid (I), states to support MultiProcessors, MP, data sharing. The MESI states are managed and maintained in the L2C based on PLB(v5) actions on data transfers and snoop actions.

Four states of the MESI (Modified, Exclusive, Shared, Invalid) protocol are maintained per cache line granule in the Coherency required storage space. The four states are described in detail below:

1. Modified (M)
   In this state the granule has been modified. The data is this state would be the most current, meaning memory is not up to date. When a granule in a processor enters this state, all other cache subsystems must invalidate their copy of this granule. *If a granule is in the M state it is invalid in all other processors* and out of date in memory.
2. Exclusive (E)
   In this state the granule is valid only in this processor. Memory also contains a valid copy. When a granule in a processor enters this state, all other cache subsystems must invalidate their copy of this granule. This state is used by a processor that intends to eventually write the location. This way all that must be done to write the location is to change state to M. No bus transaction would be necessary. *If a granule is* in the E state it is invalid in all other processors, but valid in memory.
3. Shared (S)
   In this state the granule is valid in at least one processor. Another processor may also contain this granule in the S state. Memory also contains a valid copy. When a granule in a processor enters this state, any other cache subsystem with this granule in the M state must push the data to memory and change to the S state. If the granule is in the E state it must change to the S state. *If a granule is in the S state it is* valid in memory and may reside in other processors.

4. Invalid (I)

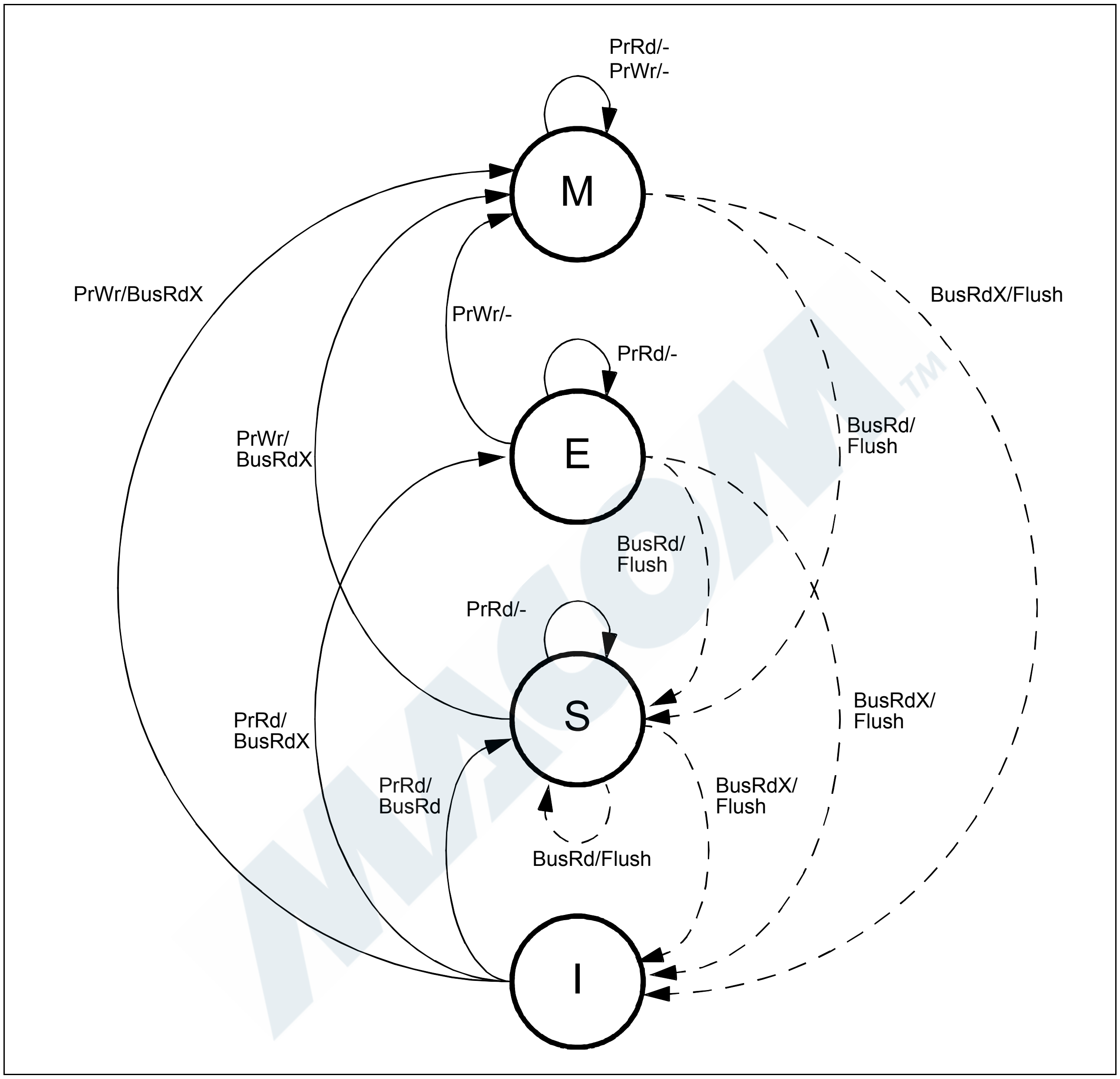
   In this state the granule is not valid. Granules not in the cache are assumed to be in this state. This is the starting state for all addresses. *If a granule is in the I state it is valid in memory and may reside in other* processors.

The bits assignment and organization are shown in *Table 5-2.*

Figure 7-12 shows the state transition diagram for the MESI protocol. In the figure, solid lines represent transitions caused by processor actions. Dashed lines represent transitions caused by PLB bus actions. The two values for each transition represent the processor or bus action followed by the bus transaction or action. The PLB bus Read, BusRd, transaction is caused by a L1 core cache read miss. The BusRdX transaction is caused by a cache read miss that wants exclusive access or a write miss. PrRd and PrWr refer to a processor read and write.

In general, L1 i-cache fetches and d-cache reads to memory coherency required pages will request that the data be placed in the E state in the L2C. However, depending on the PLB snoop results from contents of other coherent PLB masters during the PLB request, it can be placed in the S state instead. Other affects could be due to various configuration settings. Please refer to the MMU chapter and to the PLBv5 specifications.

Figure 7-12. MESI Protocol State Transition Diagram



The default state for all coherency granules is invalid. When a read miss brings in a granule, and a BusRd is requested, it is set to Shared (S) after pushing any dirty versions from other processors. If a BusRdX is requested, the granule is set to exclusive (E) after flushing it from all other caches. A subsequent write to that granule can change state to modified without a bus transaction. A write miss will always set the state to M, after flushing the granule from all other processors.

7.1.7 Coherency Configuration Settings

The L2C in conjunction with the PLB5 arbiter can maintain memory coherence of L1 data caches and L2 caches in the PPC465 Processor Complex . To enable hardware support for memory coherence, certain TLB attributes and control register settings are required.

Hardware can only maintain coherency over addresses with TLB attribute settings defining the memory page as write-through in L1 and copy-back in L2. The required settings are defined in the following table.

Table 7-7. TLB Attribute for Coherent Operation

Attribute	Value	Comment
WL1	1	Write-through in L1 DCache
W	0	Copy-Back, not Write-Through, in L2
I	0	Not cache inhibited
M	1	Coherent
G	0	Not Guarded

In addition, the SPR and L2 DCR control bits included in the table below must be set to the values indicated.

Table 7-8. Required Configuration Settings for Coherent Operation

Register	Bit	Value	Comment
CCR1	L2COBE	1	L2 Cache Op Broadcast Enable. This enables broadcasts of DCBI, DCBF, DCBST, MSYNC, and MBAR commands to other CPUs. It is also required to be set to 1 to enable coherent treatment of STWCX instructions. In general it indicates the L2 supports coherency.
L2CR2	L1CE	1	L2 controller to maintain inclusiveness of coherent granules in the L1 DCache and to generate L1 snoop commands as required.
L2CR2	SNPME	1	L2 Snoop Master Enable. Indicates that this L2 will accept and respond to snoop requests from the PLB5 arbiter.

Figure 7-9 below indicates recommended control register settings for optimal performance in a coherent system. These settings are not strictly required for coherent operation, but they are recommended.

Table 7-9. Required Configuration Settings for Coherent Operation

Register	Bit	Value	Comment
L2CR2	MEI	0	When 1, MEI, not MESI, coherency is maintained. The L2 will not allow shared coherency granules. All reads are converted to RWITM (read with intent to modify). MESI is the preferred protocol.
L2CR2	CSNPKF	0	This field indicates whether or not the L2C should convert snoop kills to flushes.
L2CR2	CSNPPEP	0	This field indicates whether or not the L2 should convert snoop pushE’s to pushes.
L2CR2	CSNPPF	0	This field indicates whether or not the L2 should convert snoop push and pushE’s to flushes. **When this bit is set, lwarx instruction will kill the reservation of other processor(s), which can result in a “livelock” condition.**
L2CR2	FRS	0	This field indicates whether the L2 takes a granule exclusive or shared for a read w/push if all snoopers respond miss. This FRS bit and the MEI bit should not be set at the same time. FRS=0 will allow exclusive granules as a result of a read.

Table 7-9. Required Configuration Settings for Coherent Operation

PLB PCICR	DIE	1	Data Intervention Enable. When set to 1, the PLB5 Arbiter will forward modified snoop data in response to a coherent read directly to the requesting master in certain circumstances. When set to 0, the snoop data must be written to memory and the coherent read then proceeds normally, returning data read from memory. Setting this bit to 1 provides some performance improvement in coherent systems. Note: the IBM PLB5 documentation refers to this bit as the “Disable Data Intervention” (DID) bit. However, the bit must be set to 1 to **enable** intervention.

7.1.8 MBAR/MSYNC Operation

Msync and mbar are ‘Memory Barrier’ instructions used to control the order of storage accesses. The PPC465 core implements ‘heavyweight syncs’ to target mainly load, store and dcbz instructions specifying Memory Coherence Required locations. Since a ‘Weakly Consistent’ execution model is implemented in PPC465 core, stores and loads are all independent and may occur in any order and therefore, a synchronizing instruction is required between stores and loads if storage access ordering is required.

The PPC465 core and PPC465L2C6 system implementation is that both msync and mbar will be executed similarly: They are ‘heavyweight sync’ rather than ‘lightweight sync’ instructions with the following exceptions:

- **dcbz** does not affect the L2 cache. **dcbz** in storage space with WL1=1 or IL1D=1 is not supported, therefore, the block containing the byte addressed by the EA in L2C will not be set to zero.

7.1.8.1 Local mbar and msync Operations

The **msync** and **mbar** are handled in the PPC465 core, which holds any subsequent storage referencing operations until the L2C completes all outstanding storage references, at least those ownership processes, by Address-Acknowledges, AAck. This implementation is to support the BookE sequential execution model.

Storage referencing requests are allowed to be serviced out of order in certain instances. If two requests do not have matching addresses and are not separated by an msync or mbar then the L2C is allowed to service these requests in any order. Since the msync and mbar are handled in the PPC465 core to ensure storage ordering, only the matching address case that must be handled by the L2C. Note that matching addresses between the L1core Data side write and the L1core Instruction side read interfaces are not taken into consideration because these would be self-modifying code sequences that require an msync to preserve the correct program order. Further details are discussed in Section 7.1.4 , “L2 Cache Hazards: L1 Core Storage Reference Ordering” on page 161.

7.1.8.2 Remote mbar and msync Operations

When the L2C receives a remote mbar or msync command from the PLB’s snoop interface, it will immediately notify the PLB that it has completed all necessary requirements to allow the mbar/msync complete. The L2C does not need to do anything internally when it sees a remote **mbar**/**msync**. Storage access ordering is also managed by PLB to ensure their proper operations.

7.1.9 Cache Line Replacement Policy

To reduce the chance of throwing out information that will be needed soon, both the instruction and data accesses by PPC465 core to L2C blocks are recorded. Since the L2C does not have knowledge of every activity of the caches of PPC465 core, the block replaced is the one that has been unreferenced by PPC465 core for the longest time. If recently used blocks are likely to be used again, then the best candidate for disposal/removal is the least-recently used block; the same assumption is made for the L2C replacement.

The replacement algorithm used for the L2C is based on the true Least-Recently-Used (LRU) algorithm. This allows the most frequently referenced (used) data to remain in the L2C cache.

7.1.9.1 LRU Algorithm with Way Locking

The L2C is implemented with a true LRU aging algorithm to maintain the best L2 cache utilization. Each set of L2 cache arrays' usages are tracked based on L1 core requests/cache line usages with 6-bits of usage information. In addition, the L2C gives software the ability to lock an entire way of the array on a per way granularity. That is, software can lock any of the ways independently of each other via L2CR0[LKWAY]. If a way is locked, the L2C will not replace the cache line in that way, but an un-locked least recently used way will be replaced.

7.1.9.2 LRU Operations for special cases

The following are LRU array operations for special cases:

• An entry is set to least recently used when the L2 invalidates that entry or on a transient stream hit or writing into the L2C array. Reasons for invalidating the entry are: dcbi, dcbf, snoop kill, and snoop flush. The LRU array update for these operations is performed when the L2 does the tag write to clear the valid bit for the entry. In the case of the transient stream lookup, the LRU array is updated to least recent after the hit determination is made and before the next tag array access. A transient stream array write sets the LRU to least recent at the time of the array write.

- An entry is set to most recently used when an entry is established (except for transient streams) or when the entry is a lookup hit from an L1 request. An entry is established via a L2C array write in which case the LRU array is written at the same time as the tag for the entry. Lookup hits cause the LRU array to be written after the hit is determined but before the next operation to the tag array. The following operations will cause the LRU array to be updated if their lookups result in a hit: L1 reads (I-side or D-side - excluding transient streams), Read with intent to modify, promote to exclusive, stores.
- An L2 Tag Array Invalidate sequence causes all LRU entries to be reset to the following sequence from least recent to most recent - Way 0, Way 1, Way 2, Way 3.
- An L2ccci causes the LRU entry at the congruence class of the L2ccci to be reset to the following sequence from least recent to most recent - Way 0, Way 1, Way 2, Way 3.

7.1.10 Inclusiveness

The L2 cache can operate in either inclusive mode or non-inclusive mode.

7.1.10.1 Inclusive Mode

Inclusive mode is enabled by setting the L1 Coherency Enable Bit L2CR2[L1CE]. When this mode is set, the L1 data cache must have all coherency required pages in the TLB indicated by M=1 (Memory Coherence Required) also marked as "L1 Write-through", WL1=1. While in inclusive mode, the L2 cache is inclusive with respect to the L1 data cache for lines that are marked Memory Coherence Required (M=1). That is, the L2 cache will hold all M=1 cache lines that are held by the L1 data cache. This is desired by a user because it allows the L2C to only snoop the L1 data cache after determining that the granule is in the L2C array. This will eliminate unnecessary snoops to L1D, and therefore, it will prevent CPU stalls due to non-relevant to L1D snoops.

A few steps are taken to ensure inclusiveness by the L2C. When replacing a line in the L2C that is marked M=1, the L2C will generate a snoop kill request to the data side L1 cache to remove that line from it.

When processing snoops in this mode, the L2C will send only "snoop kills" and "snoop flushes" for inclusive cache lines to the L1D.

Caution: When L2ccci instruction is performed to L2C cache lines that are within the page specified as coherency required, M=1, the L1 cache lines must also be invalidated by the user to preserve cache data coherency.

7.1.10.2 Non-inclusive Mode

When operating in non-inclusive mode, the L2 does not perform the steps detailed in the above section to ensure that the L2 array is inclusive of the data side L1 array.

7.1.11 L2 Cache Fixed Address Mode (FAM)

The L2C supports Fixed Address Mode (FAM) on the L2C array. When FAM is enabled, half or all of the L2C array no longer behaves like a cache but rather behaves like local memory. FAM is enabled via L2CR1[FAMES]. The starting real address of the Fixed Address Region (FAR) is specified by the FAMAR. When FAM is enabled, the L2C will compare all incoming L1core requests to the FAMAR (see table below) and determine whether the request is in the FAR or not. If the requested 36 bit real address falls within the FAR, the L2C will read or write that entry of the data array directly without performing an tag access. If the request is not in the FAR, the L2C will treat the request just as it would any request when FAM is disabled.

Note: TLB memory pages that map to a FAR must have the FAR storage attribute bit set.

L2CR1[PUBFAM] must also be set accordingly to indicate whether or not the L2C's Fixed Address Region is public or private. See "L2C Configuration Register 1 (L2CR1)" on page 183.

Caution: When the entire cache array is set to FAM (FAMES=10 for 256KB), the atomic access instructions, lwarx and stwcx. will not function correctly and instead will always fail.

Table 7-10. Address Bits Compared for FAM

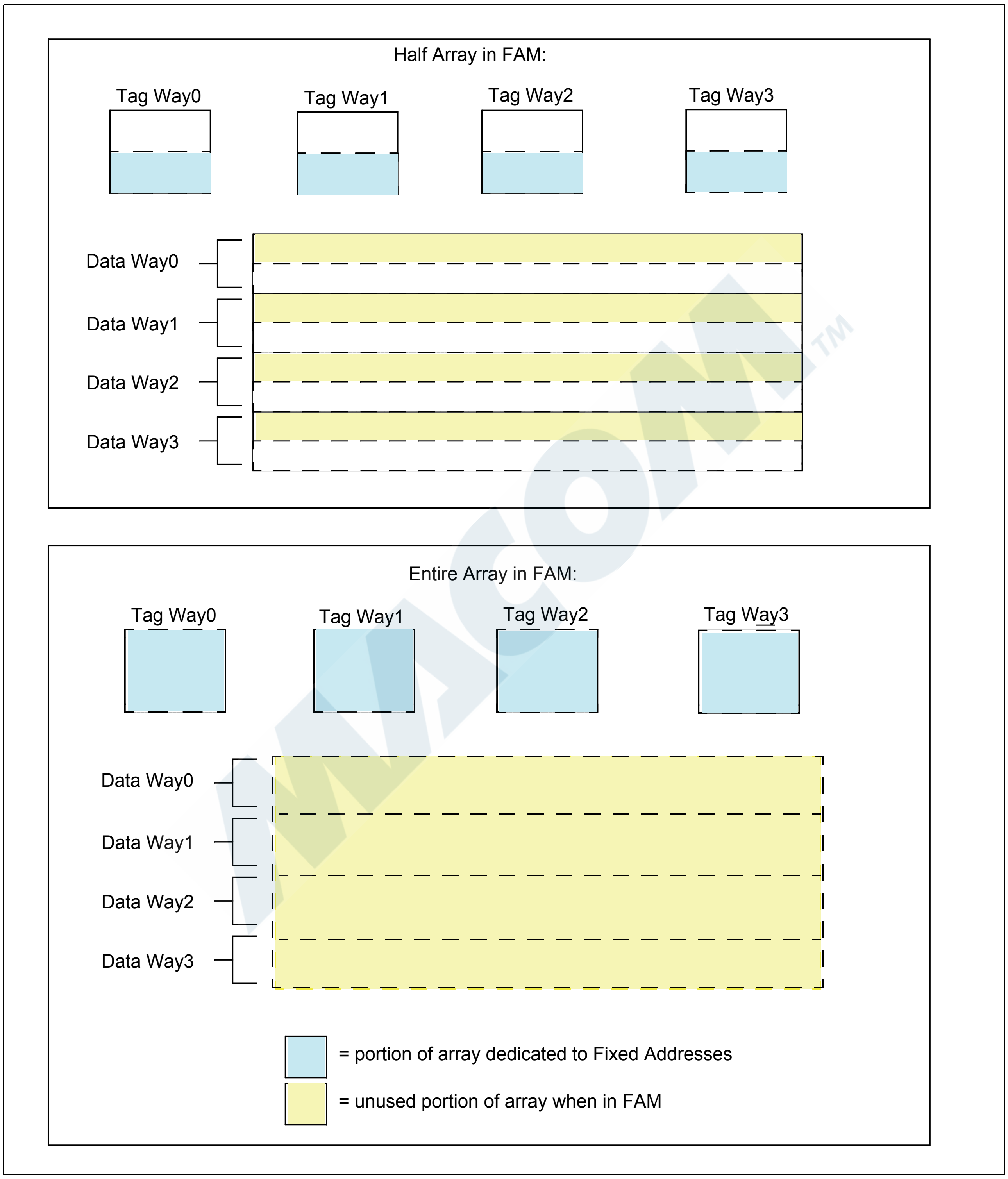
L2C Array Size	FAM Size	Address bits compared
0	—	n/a
256KB	1/2 array (128KB)	Address[0:18]
256KB	Full array (256KB)	Address[0:17]

Note: When FAM is enabled, the FAR is considered memory and is not subject to snoop requests from the PLB or cache op requests from the L1core.

Note: A CPU can only access its own FAM (whether it is configured as private or public) by using a TLB entry with the FAR bit set to '1'. **A CPU must not attempt to access its own FAM using slave way 2 on the PLB.**

The following figure depicts how the L2C array will be segmented when the array is in its different FAM modes.

Figure 7-13. Fixed Address Mode Array Allocation



Note: The upper order address half of the tag and data ways will be used for the fixed address region when half of the array is in fixed address mode.

7.1.11.1 Public vs. Private FAM

The L2 Fixed Address Region can be configured as either public or private. If the FAR is configured as private, the addresses that fall within the FAR is considered private to the L1 core connected to the L2. Given this, pages that are mapped to the FAR must not be marked as memory coherency required, that is, the M bit must be set to 0. This means that the L2 will never receive a snoop that matches private FAR.

If the FAR is configured as public, the L2 FAR is the system memory device for the addresses that fall within the FAR. In this case, the L2 slave port will respond to any PLB requests that fall within the FAR.

In this case, the M bit in the TLB entry must be set to zero because memory coherency is not supported by the hardware on the Fixed Address Region.
The FAM configuration process differs slightly depending on whether it is performed at start-up or after the L2 has been in use as a cache. In either case, all of L2 must be initialized in order to guarantee that valid ECC has been written to the array memory. **Note: care must be taken not to have overlapping public FAM addresses in multi-core systems**

7.1.11.2 Configuring L2 as FAM from start-up

- Invalidate array and Enable L2 (L2CR0[TAI] = ‘1’, L2CR0[AS] = ‘01’ (256KB))
- Pre-fill entire L2 (using DCBZ or DCBT)
- Invalidate array (L2CR0[TAI] = ‘1’)
- Enable FAM (L2CR1[FAMES] = ‘01’ for 128KB or ‘10’ for 256KB FAM)
- Enable Public FAM if desired (L2CR1[PUBFAM] = ‘0’ or ‘1’)
- Configure FAM Address Register (L2FAMAR) to the address region of the PLB5 slave segment to which the FAM is attached, if public access is required.
- Set L2SLVERAPR to 0

7.1.11.3 Configuring L2 as FAM after L2 has been in use

To configure part or all of the L2 cache as Fixed Address Memory, the following procedure must be used.

First, it is important that there be no outstanding requests in the L2 cache before the configuration routine is executed and that no L2 cache requests occur during the configuration procedure. To accomplish this, place the configuration routine in a cache-inhibited page.

Next, the following steps must be executed.

- Flush L2 cache of any modified data
- Invalidate array (L2CR0[TAI] = ‘1’)
- Enable FAM (L2CR1[FAMES] = ‘01’ for 128KB or ‘10’ for 256KB FAM)
- Enable Public FAM if desired (L2CR1[PUBFAM] = ‘0’ or ‘1’)
- Configure FAM Address Register (L2FAMAR) to the address region of the PLB5 slave segment to which the FAM is attached, if public access is required.
- SetL2SLVERAPR to 0

Note: To insure that the FAM configuration has completed before any L2 cache requests are made, execute a read of the L2CR1 and check that the FAMES field has the new value.

7.1.12 L2 Cache DCR Registers

There are two types of DCR’s in the L2C core, architected and indirect. The architected DCR’s are accessed directly on the DCR bus using their DCR number.

The architected DCRs are:

- L2DCDCRAI (L2 D-Cache DCR Address Indirect)

- L2DCDCRDI (L2 D-Cache DCR Data Indirect)

Table 7-11. L2DCDCRAI and L2DCDCRDI *Register Summary*

Mnemonic	Register	DCR Number	Access	Privileged	Page
L2DCDCRAI	L2 D-Cache DCR Address Indirect	0x0000	RW	Yes	182
L2DCDCRDI	L2 D-Cache DCR Data Indirect	0x0001	RW	Yes	182

The indirect DCRs for the L2 Cache are read and written using indirect addressing. In order to access an L2C indirect DCR, software must perform the following sequence:

- Write the L2DCDCRAI (using mtdcr) with the L2 Internal DCR number that software wishes to read or write.
- Issue a mtdcr or mfdcr using the L2DCDCRDI DCR number. This will cause the L2 to read or write the DCR currently pointed to by L2DCDCRAI.

Each indirect DCR has the following associated information:

- DCR Address: DCR Index number
- Valid Bits: valid data bits for the DCR
- Privileged: whether or not this DCR is privileged
- Access: the type of access permitted - R=read, W=write, S=set, C=clear
- Reset value: DCR value after reset
- Internal update: whether or not the L2C can update the DCR directly

7.1.12.1 L2C Indirect DCR Summary

The L2 Cache device control registers are written and read in an indirect fashion using DCR ports L2DCDCRAI (for address pointer) and L2DCDCRDI (for the data). *Table 7-12* only lists the lower offset address assignment for the L2 DCR registers.

Table 7-12 summarizes the L2 cache device control registers sorted by DCR number.

Table 7-12. L2 Cache Device Control Register Summary

Mnemonic	Register	DCR Indirect Number	Access	Privileged	Page
L2CR0	L2 Configuration Register 0	0x00	RW	Yes	182
L2CR1	L2 Configuration Register 1	0x01	RW	Yes	184
L2CR2	L2 Configuration Register 2	0x02	RW	Yes	185
L2CR3	L2 Configuration Register 3	0x03	RW	Yes	186
L2ERAPR	L2 Extended Real Address Prefix Register	0x08	RW	Yes	188
L2SLVERAPR	PLB Slave Extended Real Address Prefix Register	0x09	RW	Yes	188
L2FAMAR	Fixed Address Mode Address Register	0x0A	RW	Yes	189
L2REVID	L2 Revision ID Register	0x0B	RO	Yes	190
L2COPR	L2 Cache Op Register	0x0C	RW	Yes	187
L2THRR	L2 Throttle Register	0x0D	RW	Yes	190
L2FER0	Force Error Register 0	0x0E	RW	Yes	194

Table 7-12. L2 Cache Device Control Register Summary (continued)

Mnemonic	Register	DCR Indirect Number	Access	Privileged	Page
L2FER1	Force Error Register 1	0x0F	RW	Yes	195
L2MCSR	Machine Check Status Register	0x10	RW	Yes	197
L2MCRER	Machine Check Reporting Enable Register	0x11	RW	Yes	196
L2ECCIDX	ECC Index Register	0x12	RO	Yes	189
L2SLVEAR0	PLB Slave Port Error Address Register 0	0x14	RO	Yes	217
L2SLVEAR1	PLB Slave Port Error Address Register 1	0x15	RO	Yes	217
L2SLVMIDR	PLB Slave Port Error Master ID Register	0x16	RO	Yes	218
L2DBACMR	L2 Debug Address Compare Mask Register	0x17	RW	Yes	199
L2DBSR	L2 Debug Status Register	0x18	RC	Yes	200
L2DBCR	L2 Debug Control Register	0x19	RW	Yes	200
L2DBACSR	L2 Debug Address Compare Status Register	0x1A	RC	Yes	203
L2DBACCR	L2 Debug Address Compare Control Register	0x1B	RW	Yes	203
L2SLVAC0	Slave Address Compare 0 Register	0x1C	RW	Yes	202
L2DBACR	L2 Debug Address Compare Register	0x1D	RW	Yes	204
L2SNPAC0	Snoop Address Compare 0 Register	0x1E	RW	Yes	202
L2SNPAC1	Snoop Address Compare 1 Register	0x1F	RW	Yes	202
L2DBDR0	L2 Debug Data Register 0	0x20	RO	Yes	191
L2DBDR1	L2 Debug Data Register 1	0x21	RO	Yes	191
L2DBDR2	L2 Debug Data Register 2	0x22	RO	Yes	192
L2DBDR3	L2 Debug Data Register 3	0x23	RO	Yes	192
L2DBDR4	L2 Debug Data Register 4	0x24	RO	Yes	192
L2DBTR0	L2 Debug Tag Register 0	0x25	RO	Yes	193
L2DBTR1	L2 Debug Tag Register 1	0x26	RO	Yes	193
L2DBTR2	L2 Debug Tag Register 2	0x27	RO	Yes	194
L2MCSRS	L2 Machine-Check Status Register (set)	0x30	S	Yes	n/a
L2DBSRS	L2 Debug Status Register (set)	0x38	S	Yes	n/a
L2DBACSRS	L2 Debug Address Compare Status Register (set)	0x3A	S	Yes	n/a

7.1.12.2 L2DCDCRAI (L2 D-Cache DCR Address Pointer)

DCR Address: {L2 DCR Address, 0x0000}
Valid Bits: 26-31
Privileged: Yes
Access: RW

This DCR is loaded with the address for an indirect L2 DCR that will be written to or read from when a mtdcr or mfdcr is executed using the L2DCDCRDI DCR address.

.

Figure 7-14. L2 D-Cache DCR Address Pointer (L2DCDCRAI)

Bits	Mnemonic	Description	Notes
0:25		Reserved	
26:31	DCRADDR	Address of indirect DCR to access	

7.1.12.3 L2DCDCRDI (L2 D-Cache DCR Data Indirect)

DCR Address: {L2 DCR Address, 0x0001}
Privileged: Yes
Access: RW
Reset Value: N/A

This DCR does not exist as a register in hardware, but is simply a command to the L2C to write or read on of its indirect DCRs. When a mtdcr or mfdcr is executed to this DCR, the L2C will write or read the indirect DCR pointed to by L2DCDCRAI.

7.1.12.4 L2C Configuration Register 0 (L2CR0)

L2CR0 register described in *Figure 7-15* contains the configuration directives for L2C. The definitions are described in the figure below. This is one of four Configuration Register, L2CR0 - L2CR3.

L2 Indirect DCR number: 0x00
Privileged: Yes
Access: R/W
Reset Value: All bits are reset to zero.

Figure 7-15. L2C Configuration Register 0 *(L2CR0)*

Bits	Mnemonic	Description	Notes
0:2		Reserved	
2:3	AS	Array Size 00 - No L2 Array 01 - 256 KB L2 Array 10 - Reserved 11 - Reserved	This field is set to the size of the array contained in the L2C. If the L2 array is disabled, atomic accesses using lwarx/stwcx instructions will not operate correctly for multiprocessor configurations. Specifically, the hardware will not snoop the reservation register if AS=00, so both processors in an SMP configuration may conclude that they own a locked resource.
4:5		Reserved	
6:7	TAA	Tag Array Access	This field is set to the number of CPU cycles it takes to access the L2C Tag Array. All binary values greater than 0 will reduce performance and power consumption. This field is ignored if L2CR0[AS] == 00.
8:9			

Figure 7-15. L2C Configuration Register 0 (L2CR0) (continued)

Bits	Mnemonic	Description	Notes
10:11	DAA	Data Array Access	This field is set to the number of CPU cycles it takes to access the L2 Data Array. All binary values are valid, provided value of this field is equal to or larger than the value of L2CR0[TAA]. Binary values greater than 0 will reduce performance and power consumption. This field is ignored if L2CR0[AS] == 00.
12:16		Reserved	
17	TTWEN	Touch Target Way Enable	This bit indicates that touch requests targeting the L2 should be written into the way specified by L2CR0[TTW]. This field is ignored if L2CR0[AS] == 00.
18:19	TTW	Touch Target Way 00 Way 0 01 Way 1 10 Way 2 11 Way 3	This field specifies which way of the L2 array a touch targeting the L2 should be written into when L2CR0[TTWEN] is set. This field is ignored if L2CR0[AS] == 00.
20:22	LKWAY	Locked Ways	This field specifies which ways in the L2 are currently locked. Each bit of this field corresponds to a way in the L2 array. Bit 0 corresponds to Way 0, bit 1 to Way 1, and bit 2 to Way 2. If a way is locked as indicated by this field, it can only be written to by setting L2CR0[TTWEN] and L2CR0[TTW]. This field is ignored if L2CR0[AS] = 00.
23:27		Reserved	
28	DECC	Disable ECC 0 ECC logic enabled 1 ECC logic disabled	A value of 1 for this bit disables the L2's ECC logic meaning that no array errors will be detected or corrected. This field is ignored if L2CR0[AS] == 00.
29	DECA	Disable Error Correction in Array 0 Error correction in array enabled 1 Error correction in array disabled	A value of 1 on this bit prevents the L2C from auto-updating the L2C array with corrected data after detecting a single bit error. This field is ignored if L2CR0[AS] = 00.
30	TAI	Tag Array Invalidate	This bit indicates that the hardware Tag Array Invalidate is in progress. This bit is set by software to start the tag array invalidation and is cleared by hardware once the invalidation is complete. This bit cannot be set if L2CR0[AS] = 00.
31		Reserved	

7.1.12.5 L2C Configuration Register 1 (L2CR1)

L2CR1 register described in *Figure 7-16* contains the configuration directives for L2C. The definitions are described in the figure below. This is one of four Configuration Register, L2CR0 - L2CR3. Some bits need to be set in conjunction with L2CR0 bits.

L2 Indirect DCR number: 0x01
Privileged: Yes
Access: R/W
Reset Value: All bits are reset to zero.

Figure 7-16. L2C Configuration Register 1 (L2CR1)

Bits	Mnemonic	Description	Notes
0:1	FAMES	Fixed Address Mode Enable / Size 00 FAM is off 01 Half of the array is in FAM 10 All of the array is in FAM 11 Reserved	This field enables the L2C array to operate in Fixed Address Mode. This field is ignored if L2CR0[AS] = 00.
2	PUBFAM	Public Fixed Address Mode 0 FAR is private 1 FAR is public	This field indicates whether or not the L2C's Fixed Address Region is public or private. This field is ignored if L2CR0[AS] = 00 or if L2CR1[FAMES] = 00.
3	FDIOD	Force D-side In Order Data 0 Return data in the order received 1 Always return data in order	This field indicates whether or not the L2C will always return data in order to the L1. This field is ignored if L2CR0[AS] = 00 or if L2CR1[FAMES] = 00.
4	SL1ILU	Serialize L1 I Side Lookup 0 Array accesses are parallel 1 Array accesses are serial	This field indicates whether or not L2C array accesses from the L1 Instruction Side should be serialized. A serial access indicates that the tag array result is known before the data array is accessed. A parallel access indicates that the tag and data arrays are accessed in parallel. This field is ignored if L2CR0[AS] = 00. Note: Setting SL1ILU and SL1DLU bits to '1' will reduce dynamic power dissipation by avoiding reading all four cache ways for each L2 access. The cost of setting these bits to '1' is reduced performance, because each cache access takes one additional cycle. This bit should be configured at startup and not changed during normal operations
5	SL1DLU	Serialize L1 D Side Lookup 0 Array accesses are parallel 1 Array accesses are serial	This field indicates whether or not L2C array accesses from the L1 Data Side should be serialized. A serial access indicates that the tag array result is known before the data array is accessed. A parallel access indicates that the tag and data arrays are accessed in parallel. This field is ignored if L2CR0[AS] = 00. Note: Setting SL1ILU and SL1DLU bits to '1' will reduce dynamic power dissipation by avoiding reading all four cache ways for each L2 access. The cost of setting these bits to '1' is reduced performance, because each cache access takes one additional cycle. This bit should be configured at startup and not changed during normal operations Note: If the entire L2 array is placed into FAM mode atomic accesses using lwarx/stwcx instructions will not operate correctly for multiprocessor configurations. Specifically, the hardware will not snoop the reservation register if FAMES=10, so both processors in an SMP configuration may conclude that they own a locked resource.
6	LRUUM	LRU Update Mode 0 Set to LRU if any granule in the line is invalidated 1 Set to LRU if all granules in the line are invalidated, else no LRU update	This field indicates how the LRU array should be updated when granules within the L2C array are invalidated. This field is ignored if L2CR0[AS] = 00.

Figure 7-16. L2C Configuration Register 1 (L2CR1) (continued)

Bits	Mnemonic	Description	Notes
7	DSG	Disable Store Gathering 0 Gather 1 No gathering	This field indicates whether or not the L2 will perform store gathering on L1 data. This field is ignored if L2CR0[AS] = 00
8	TAS	Touch Allocation Size 0 Get 128 Bytes on touch 1 Get only requested granule on touch	This field is ignored if L2CR0[AS] = 00. Note: If L2CR1[RAS] == '0', setting the L2CR1[TAS] to '1' has no effect. In this case all touch instructions will cause a full 128 byte line to be read.
9	RAS	Read Allocation Size 0 Get up to 128 Bytes on read 1 Get only requested granule on read	This field is ignored if L2CR0[AS] = 00.
10:31		Reserved	

7.1.12.6 L2C Configuration Register 2 (L2CR2)

L2CR2 register described in *Figure 7-17* contains the configuration directives for snoop functions in L2C. The definitions are described in the figure below. This is one of four Configuration Register, L2CR0 - L2CR3.

L2 Indirect DCR number: 0x02
Privileged: Yes
Access: R/W
Reset Value: All bits are reset to zero.

Figure 7-17. L2C Configuration Register 2 (L2CR2)

Bits	Mnemonic	Description	Notes
0	SNPME	Snooping Master Enable 0 Snooping master is disabled 1 Snooping master is enabled	This field enables the L2C snooper port.
1	SSNPLU	Serialize Snoop Lookup 0 Array accesses are parallel 1 Array accesses are serial	This field indicates whether or not L2C array accesses from the snooper port should be serialized. A serial access indicates that the tag array result is known before the data array is accessed. A parallel access indicates that the tag and data arrays are accessed in parallel.
2	SNPCM	Snoop Castout Mode 0 Perform castouts in low power mode 1 Perform castouts in high performance mode	This field indicates how the L2C should perform snoop castouts.
3	MEI	MEI Mode 0 Use MESI 1 Use MEI	This field indicates whether or not the L2C should use MESI or MEI for hardware enforced coherency.If the L2 is configured to use MEI mode, snoop pushes will be converted to snoop flushes and L1 reads with M == 1 will be converted to RWITM (read with intent to modify)when sent to the PLB. This MEI bit and FRS [7] bit should not be set at the same time.
4	CSNPKF	Convert Snoop Kills to Flushes 0 Do not convert snoop kills to flushes 1 Convert snoop kills to flushes	This field indicates whether or not the L2C should convert snoop kills to flushes.

Figure 7-17. L2C Configuration Register 2 (L2CR2) (continued)

Bits	Mnemonic	Description	Notes
5	CSNPPEP	Convert Snoop PushE's to Pushes 0 Do not convert snoop pushE's to pushes 1 Convert snoop pushE's to pushes	This field indicates whether or not the L2 should convert snoop pushE's to pushes.
6	CSNPPF	Convert Snoop Pushes to Flushes 0 Do not convert snoop push(E) to flushes 1 Convert snoop push(E) to flushes	This field indicates whether or not the L2 should convert snoop push and pushE's to flushes. When this bit is set, lwarx instruction will kill the reservation of other processor(s).
7	FRS	Force Read-Shared 0 Take granule exclusive 1 Take granule shared	This field indicates whether the L2 takes a granule exclusive or shared for a read w/push if all snoopers respond miss. This FRS bit and MEI [3] bit should not be set at the same time
8:9		Reserved	
10:11	SNPRQPD	Snoop Request Pipeline Depth	This field indicates the depth of the L2's snoop request queue. It is suggested that a value of b11 (max depth of 4) be used to improve performance in multiprocessor uses.
12	FMSTNS	Force Master Non-snoopable 0 Do not convert PLB requests to nonsnoopable 1 Convert all PLB requests to non-snoopable	This field indicates whether the L2 should make all requests PLB non-snoopable.
13:15		Reserved	
16	L1CE	L1 Coherency Enabled 0 L1 coherency support is not enabled 1 L1 coherency support is enabled	This field indicates whether or not the L2 should support L1 coherency. If this bit is set, the L2 will send snoops to the L1. The L2 will also maintain inclusiveness with coherent paged in the L1 Data Cache. If this bit is not set and coherency is desired in the system, all M pages must be marked L1 Inhibited.
17:31		Reserved	

7.1.12.7 L2C Configuration Register 3 (L2CR3)

L2CR3 register described in *Figure 7-18* contains the configuration directives for PLB interface in L2C. The definitions are described in the figure below. This is one of four Configuration Register, L2CR0 - L2CR3.

L2 Indirect DCR number: 0x03
Privileged: Yes
Access: R/W
Reset Value: All bits are reset to zero.

Figure 7-18. L2C Configuration Register 3 (L2CR3)

Bits	Mnemonic	Description	Notes
0	PLBPE	PLB Parity Enable 0 Do not generate / check parity 1 Reserved	This field indicates whether or not parity should be generated for requests to the PLB and checked for requests from the PLB. For the PPC465 core, this bit should be set to 0.
1	MSTOORE	PLB Master Out-of-Order Read Enable 0 Out of Order Read Data Disabled 1 Out of Order Read Data Enabled	This field indicates whether PLB Read requests should be made with Out Of Order Read Data Enabled or not. Note: The PPC465 does not support out of order reads. MSTOORE should always be set to 0.

Figure 7-18. L2C Configuration Register 3 (L2CR3) (continued)

Bits	Mnemonic	Description	Notes
2:3		Reserved	
4:7	MSTPD	PLB Master Port Pipeline Depth L2 open address tenure = (MSTPD +1)	This field indicates the number of PLB master requests that can have an open address tenure. At present the only valid value is 0011, which is 4 open address tenures.
8:9	MSTIRP	PLB Master Instruction Side Request Priority Must be set to 00.	This field is used by the L2 when mastering a PLB request that was initiated by the Instruction side L1 interface. For the PPC465, this field must be set to 00.
10:26		Reserved	
27	SLVTWF	PLB Slave Target Word First	Set to indicate that the slave should send its read data out to the bus in target word first order. Note: It is recommended that this be set to 1 to increase performance.
28:29		Reserved	
30:31	SLVPD	PLB Slave Port Pipeline Depth 0011 4 open address tenures All other settings are reserved.	This field indicates the depth of the L2's slave port request queue.

7.1.12.8 L2 Cache Op Register (L2COPR)

L2COPR register described in *Figure 7-19* is used to perform either a L2ccci invalidate or a L2read. Note that it is not possible to set L2ccci if also setting L2read. L2CR0[AS] must be non-zero in order to set L2ccci or L2read. The DCR bus will not Ack an L2read until it has completed. An L2ccci is Acked as soon as the array accepts the request.

L2 Indirect DCR number: 0x0C
Privileged: Yes
Access: R/W
Reset Value: All bits are reset to zero.

Figure 7-19. L2 Cache Op Register (L2COPR)

Bits	Mnemonic	Description	Notes
0	L2CCCI		
1	L2READ		
2:11		Reserved	
12:24	WAY/CC	See tables below for bit description	
25:27	QWADDR	Quadword Address (L2 Read only)	
28:31		Reserved	

Table 7-13. L2COPR fields based for L2CCCI

Array Size	L2ccci Bit	Reserved	Congruence Class	Reserved

Table 7-13. L2COPR fields based for L2CCCI

256 KB	0	1 : 15	16 : 24	25 : 31

Table 7-14. L2COPR format for L2READ

Array Size	Reserved	L2read	Reserved	Way	Congruence Class	Quadword Select	Reserved
256 KB	0	1	2 : 13	14 : 15	16 : 24	25 : 27	28 : 31

7.1.12.9 L2 Extended Real Address Prefix Register (L2ERAPR)

L2ERAPR register described in *Figure 7-20* contains the high order [0:27] address bits that the L2C is currently mapping all of its addresses to. In the PPC465 Processor Complex, the real address is 36 bits, therefore, the extended address is not used. This register should be set to a value of zero.

L2 Indirect DCR number: 0x08
Privileged: Yes
Access: R/W
Reset Value: Undefined

Figure 7-20. L2 Extended Real Address Prefix Register (L2ERAPR)

Bits	Mnemonic	Description	Notes
0:27	HOA	High Order Address Bits of the 64-bit Real Address	Must set to 0.
28:31		Reserved	

7.1.12.10 PLB Slave Extended Real Address Prefix Register (L2SLVERAPR)

L2SLVERAPR register described in *Figure 7-21* contains the high order [0:27] address bits that the L2C is currently mapping its PLB slave port address to for the public Fixed Address Mode range. In the PPC465 Processor Complex, the real address is 36 bits, therefore, the extended address is not used. This register should be set to a value of zero.

L2 Indirect DCR number: 0x09
Privileged: Yes
Access: R/W
Reset Value: Undefined

Figure 7-21. PLB Slave Extended Real Prefix Register (L2SLVERAPR)

Bits	Mnemonic	Description	Notes
0:27		Bits 0 - 27 of the 64 bit real address	Must set to 0.
28:31		Reserved	

7.1.12.11 Fixed Address Mode Address Register (L2FAMAR)

FAMAR register described in *Figure 7-22* contains 18-bits of the starting (base) address (bits 28 to 45 of the 64 bit address) of the L2C's Fixed Address Mode (FAM) range. Fixed Address Mode is enabled via the FAM bits in the L2CR1[FAMES]. If FAM is disabled, this DCR value is not used by the L2. The upper address (bits 0 to 27) of the 64-bit address for the FAM range supplied by the L2SLVERAPR register for are not used in the PPC465 Processor Complex.

L2 Indirect DCR number: 0x0A
Privileged: Yes
Access: R/W
Reset Value: Undefined

Figure 7-22. Fixed Address Mode Address Register (L2FAMAR)

Bits	Mnemonic	Description	Notes
0:17	FAM	Fixed Address Mode Value	The FAM is bits 28 to 45 of the 64-bit address where the ERPN is address bits 28:31 and the RPN is address bits 32:45.
18:31		Reserved	

The FAM base value must be aligned to the size of the L2C fixed address region as shown in *Table 7-15.*

Table 7-15. FAMAR Bits Setting

L2 Array Size	FAM Size	Bits that must be 0
256KB	1/2 array (256KB)	-
256KB	full array (256KB)	[18]

7.1.12.12 ECC Index Register (L2ECCIDX)

L2ECCIDX register described in *Figure 7-23* is updated by the L2C when it reports a single or double bit error and is locked until the error status bit in L2MCSR is cleared. The value in the L2ECCIDX corresponds to the way and set of the cache location in error. There are no registers to indicate the exact dword or bit.

L2 Indirect DCR number: 0x12
Privileged: Yes
Access: Read Only
Reset Value: undefined

Figure 7-23. ECC Index Register (L2ECCIDX)

Bits	Mnemonic	Description	Notes
0:15		Reserved	
16:24	CCE	Congruence Class of the Error	
25:31		Reserved	

7.1.12.13 L2THRR (L2 Throttle Register)

Separate throttling mechanisms are provided for CPU instruction cache interface, CPU data cache interface, L2 cache snoop interface and L2 cache slave port interface to save L2 cache power by reducing the frequency of those interface responses. The 8-bit counter value specifies the number of L2 cache clock cycles to delay or insertion of "wait" state between requests of the corresponding interface.

L2 Indirect DCR Number: 0x0D
Valid Bits: 0-31
Privileged: Yes
Access: RW
Reset Value: All bits are reset to zero

Figure 7-24. L2 Throttle Register (L2THRR)

Bits	Mnemonic	Description	Notes
0:7	L2ITHR	L2 I Side Throttle Value	
8:15	L2DTHR	L2 DSide Throttle Value	
16:23	SNPTHR	Snoop Throttle Value	
24:31	SLVTHR	Slave Throttle Value	

7.1.12.14 L2REVID (L2 Revision ID Register)

L2REVID contains the L2 controller revision ID.

Figure 7-25. L2 Revision ID Register (L2REVID)

Bits	Mnemonic	Description	Notes
0:11		Reserved	
12:23	RVNUM	Revision Number	
24:31	BRN	Branch Revision Number	

7.2 L2 Debug Facilities

The L2C supports various debug facilities to allow for easier hardware debug and state initialization by software. There are three debug operations that the L2C supports: L2read of the cache array contents, forcing ecc errors, and port access testing to create debug events.

These are explained in the following three sections.

7.2.1 L2read of Cache Array Contents

The L2C is able to read a particular way, congruence class, and quadword via a mtdcr to L2COPR with bit 1 set to a '1'. On detecting this mtdcr, the L2 performs an L2read to the way, congruence class, and quadword specified by bits [n:27] of the L2COPR, where n is based on cache size. *Table 7-16* illustrates which bits of the L2COPR are used based on cache size.

Table 7-16. L2COPR format for L2read

Array Size	Reserved	L2read	Reserved	Way	Congruence Class	Quadword Select	Reserved
256K	0	1	2:13	14:15	16:24	25:27	28:31

The L2C will with-hold the DCR acknowledge signal from the PPC465 core, L1core, until the L2read has completed.

Each L2C read instruction to the L2C can read out one quadword of data from one way of the L2C. Based on this, it takes (8) L2read's to read out one L2C cache line from one way. The data read out of the array is stored into L2C DCR's (L2DBDR0 - L2DBDR4).

When an L2read is performed with the Quadword select equal to zero, the Tag information is also read out and is stored into L2C DCR's (L2DBTR0 - L2DBTR2).

7.2.1.1 L2 Debug Data Register 0 (L2DBDR0)

L2DBDR0 register described in *Figure 7-26* is updated with the data from the L2C data array following an L2read.

L2 Indirect DCR number: 0x20
Privileged: Yes
Access: Read Only
Reset Value: Undefined

Figure 7-26. L2 Debug Data Register 0 (L2DBDR0)

Bits	Mnemonic	Description	Notes
0:31		Word 0 data from targeted quadword	

7.2.1.2 L2 Debug Data Register 1 (L2DBDR1)

L2DBDR1 register described in *Figure 7-27* is updated with the data from the L2C data array following an L2read.

L2 Indirect DCR number: 0x21
Privileged: Yes
Access: Read Only
Reset Value: Undefined

Figure 7-27. L2 Debug Data Register 1 (L2DBDR1)

Bits	Mnemonic	Description	Notes
0:31		Word 0 data from targeted quadword	

7.2.1.3 L2 Debug Data Register 2 (L2DBDR2)

L2DBDR2 register described in *Figure 7-28* is updated with the data from the L2C data array following an L2read.

L2 Indirect DCR number: 0x22
Privileged: Yes
Access: Read Only
Reset Value: Undefined

Figure 7-28. L2 Debug Data Register 2 (L2DBDR2)

Bits	Mnemonic	Description	Notes
0:31		Word 0 data from targeted quadword	

7.2.1.4 L2 Debug Data Register 3 (L2DBDR3)

L2DBDR3 register described in *Figure 7-29* is updated with the data from the L2C data array following an L2read.

L2 Indirect DCR number: 0x23
Privileged: Yes
Access: Read Only
Reset Value: Undefined

Figure 7-29. L2 Debug Data Register 3 (L2DBDR3)

Bits	Mnemonic	Description	Notes
0:31		Word 0 data from targeted quadword	

7.2.1.5 L2C Debug Data Register 4 (L2DBDR4)

L2DBDR4 register described in *Figure 7-30* is updated with the ECC from the L2 data array following an L2read.

L2 Indirect DCR number: 0x24
Privileged: Yes
Access: Read Only
Reset Value: Undefined

Figure 7-30. L2 Debug Data Register 4 (L2DBDR4)

Bits	Mnemonic	Description	Notes
0:7	CCE	Doubleword 0 ECC from targeted quadword	
8:15	CCE	Doubleword 1 ECC from targeted quadword	
16:31		Reserved	

7.2.1.6 L2 Debug Tag Register 0 (L2DBTR0)

L2DBTR0 register described in *Figure 7-31* is updated with the address from the L2C tag array following an L2read targeting quadword 0 of the L2C line.

L2 Indirect DCR number: 0x25
Privileged: Yes
Access: Read Only
Reset Value: Undefined

Figure 7-31. L2 Debug Tag Register 0 (L2DBTR0)

Bits	Mnemonic	Description	Notes
0:20	ADDR	Address from the tag array	
21:31		Reserved	

7.2.1.7 L2 Debug Tag Register 1 (L2DBTR1)

LDBTR1 register described in *Figure 7-32* is updated with the attributes and ECC from the L2C tag array following an L2read targeting quadword 0 of the L2 line.

L2 Indirect DCR number: 0x26
Privileged: Yes
Access: Read Only
Reset Value: Undefined

Figure 7-32. L2 Debug Tag Register 1 (L2DBTR1)

Bits	Mnemonic	Description	Notes
0		Reserved	
1	M	M bit	
2	G	G bit	
3		Reserved	
4:7	USER	User bits	
8		Reserved	
9:11	GRAN0	Granule 0 state bits (Valid, Shared, Modified)	
12		Reserved	
13:15	GRAN1	Granule 1 state bits (Valid, Shared, Modified)	
16		Reserved	
17:19	GRAN2	Granule 2 state bits (Valid, Shared, Modified)	
20		Reserved	
21:23	GRAN3	Granule 3 state bits (Valid, Shared, Modified)	
24:31	ECC	Tag ECC bits	

7.2.1.8 L2 Debug Tag Register 2 (L2DBTR2)

L2DBTR2 register described in *Figure 7-33* is updated with the information from the L2C LRU array following an L2read targeting quadword 0 of the L2C line.

L2 Indirect DCR number: 0x27
Privileged: Yes
Access: Read Only
Reset Value: Undefined

Figure 7-33. L2 Debug Tag Register 2 (L2DBTR2)

Bits	Mnemonic	Description	Notes
0:1		Reserved	
2:7	LRU	LRUs	
8:31		Reserved	

7.2.2 Forcing ECC Errors

Since the L2C supports single bit error correction and double bit error detection, it is helpful to provide a mechanism for testing that this logic performs correctly. Since errors are uncommon and not deterministic in nature, the hardware needs to provide a way for software to induce errors into the array for testing purposes. The L2C provides a mechanism for software to force an erroneous ECC code into the array via DCRs (L2FER0 - L2FER1).

A bit set to 1 in either but not both of these registers will force a single bit error on the field that the asserted bit corresponds to. The same bit set to 1 in both of these registers will force a double bit error on the field that the asserted bits correspond to. The bits in these registers are not mutually exclusive meaning that a error could be forced on all 17 fields at the same time by setting all bits in one or both of these registers to 1. The specified bit(s) are inverted and loaded into the designated dword or tag location of the next occurrence of a cache miss in the array. This will cause the ECC Machine Check error condition when this location is read from the cache during normal operation. The L2FER0 and L2FER1 register contents are automatically cleared each time the error is injected into a cache location. Only use L2FER0 and L2FER1 to inject errors during special testing procedures so that erroneous errors are not created during normal operations. There is no way to selectively set a particular bit within the dword. Register L2ECCIDX will capture the cache index location of the ECC error and the L2MCRER and L2MCSR are used to enable and show ECC error status. See the Interrupts and Exceptions chapter for Cache Error handling.

7.2.2.1 Force Error Register 0 (L2FER0)

Force Error Register 0 described in *Figure 7-34* is used to inject ECC errors into the L2C array. Bits in this register are set to 1 by software to indicate that the next load into the L2C array and read should have an error forced on that line. After the load and read the L2C will clear this register.

L2 Indirect DCR number: 0x0E
Privileged: Yes
Access: R/W
Reset Value: Reset to zero

Figure 7-34. Force Error Register 0 (L2FER0)

Bits	Mnemonic	Description	Notes
0:14		Reserved	

Figure 7-34. Force Error Register 0 (L2FER0) (continued)

Bits	Mnemonic	Description	Notes
15	FETA	Force error in tag array	
16	FED0	Force error on doubleword 0 in data array	
17	FED1	Force error on doubleword 1 in data array	
18	FED2	Force error on doubleword 2 in data array	
19	FED3	Force error on doubleword 3 in data array	
20	FED4	Force error on doubleword 4 in data array	
21	FED5	Force error on doubleword 5 in data array	
22	FED6	Force error on doubleword 6 in data array	
23	FED7	Force error on doubleword 7 in data array	
24	FED8	Force error on doubleword 8 in data array	
25	FED9	Force error on doubleword 9 in data array	
26	FED10	Force error on doubleword 10 in data array	
27	FED11	Force error on doubleword 11 in data array	
28	FED12	Force error on doubleword 12 in data array	
29	FED13	Force error on doubleword 13 in data array	
30	FED14	Force error on doubleword 14 in data array	
31	FED15	Force error on doubleword 15 in data array	

7.2.2.2 Force Error Register 1 (L2FER1)

Force Error Register 1 described in *Figure 7-17* is used to inject ECC errors into the L2C array. Bits in this register are set to 1 by software to indicate that the next load into the L2C array and read should have an error forced on that line. After the load and read the L2C will clear this register.

L2 Indirect DCR number: 0x0F
Privileged: Yes
Access: R/W
Reset Value: Reset to zero

Table 7-17. Force Error Register 1 (L2FER1)

Bits	Mnemonic	Description	Notes
0:14		Reserved	
15	FETA	Force error in tag array	
16	FED0	Force error on doubleword 0 in data array	
17	FED1	Force error on doubleword 1 in data array	
18	FED2	Force error on doubleword 2 in data array	
19	FED3	Force error on doubleword 3 in data array	

Table 7-17. Force Error Register 1 (L2FER1) (continued)

Bits	Mnemonic	Description	Notes
20	FED4	Force error on doubleword 4 in data array	
21	FED5	Force error on doubleword 5 in data array	
22	FED6	Force error on doubleword 6 in data array	
23	FED7	Force error on doubleword 7 in data array	
24	FED8	Force error on doubleword 8 in data array	
25	FED9	Force error on doubleword 9 in data array	
26	FED10	Force error on doubleword 10 in data array	
27	FED11	Force error on doubleword 11 in data array	
28	FED12	Force error on doubleword 12 in data array	
29	FED13	Force error on doubleword 13 in data array	
30	FED14	Force error on doubleword 14 in data array	
31	FED15	Force error on doubleword 15 in data array	

7.2.2.3 L2 Machine Check Reporting Enable Register (L2MCRER)

L2C machine check reporting enable register described in *Figure 7-35* contains the reporting enable bits for L2C machine checks.

L2 Indirect DCR number: 0x11
Privileged: Yes
Access: R/W
Reset Value: Reset to zero

Figure 7-35. L2 Machine Check Reporting Enable Register (L2MCRER)

Bits	Mnemonic	Description	Notes
0		Reserved	
1	ITSBEN	Instruction Side Tag Array Single Bit Enable	
2	ITDBEN	Instruction Side Tag Array Double Bit Enable	
3	IDSBEN	Instruction Side Data Array Single Bit Enable	
4	IDDBEN	Instruction Side Data Array Double Bit Enable	
5	DTSBEN	Data Side Tag Array Single Bit Enable	
6	DTDBEN	Data Side Tag Array Double Bit Enable	
7	DDSBEN	Data Side Data Array Single Bit Enable	
8	DDDBEN	Data Side Data Array Double Bit Enable	
9	SNPTSBEN	Snoop Side Tag Array Single Bit Enable	
10	SNPTDBEN	Snoop Side Tag Array Double Bit Enable	
11	SNPDSBEN	Snoop Side Data Array Single Bit Enable	

Figure 7-35. L2 Machine Check Reporting Enable Register (L2MCRER) (continued)

Bits	Mnemonic	Description	Notes
12	SNPDDBEN	Snoop Side Data Array Double Bit Enable	
13	FARMEN	Fixed Address Region Mismatch Enable	
14:15		Reserved	
16	MSTRQEN	PLB Master Port Request Enable	
17	MSTRDEN	PLB Master Port Read Data Enable	
18	MSTRDPEN	PLB Master Port Read Data Parity Enable	
19	SNPAPEN		
20		Reserved	
21	SLVWDEN	PLB Slave Port Write Data Enable	
22	SLVWDPEN	PLB Slave Port Write Data Parity Enable	
23	SLVAPEN	PLB Slave Port Address Parity Enable	
24	SLVBEPEN	PLB Slave Port Byte Enable Parity Enable	
25	SLVDSBEN	PLB Slave Port Data Array Single Bit Enable	
26	SLVDDBEN	PLB Slave Port Data Array Double Bit Enable	
27	SLVIAEEN	PLB Slave Port Invalid Address Error Enable	
28	SLVICEEN	PLB Slave Port Invalid Command Error Enable	
29:30		Reserved	
31	DCRTOEN	DCR Timeout Enable	

7.2.2.4 L2 Machine Check Status Register (L2MCSR)

L2C machine check status register described in *Figure 7-36* contains the syndrome bits generated as a result of an L2C or PLB Machine Check. The summary bit is only set is an event bit and its corresponding reporting enable are set.

L2 Indirect DCR number: 0x10
Privileged: Yes
Access: Read Clear/Set
Reset Value: Reset to zero

Figure 7-36. L2 Machine Check Status Register (L2MCSR)

Bits	Mnemonic	Description	Notes
0	MCS	Machine Check Summary	
1	ITSBE	Instruction Side Tag Array Single Bit Error	
2	ITDBE	Instruction Side Tag Array Double Bit Error	
3	IDSBE	Instruction Side Data Array Single Bit Error	

Figure 7-36. L2 Machine Check Status Register (L2MCSR) (continued)

Bits	Mnemonic	Description	Notes
4	IDDBE	Instruction Side Data Array Double Bit Error	
5	DTSBE	Data Side Tag Array Single Bit Error	
6	DTDBE	Data Side Tag Array Double Bit Error	
7	DDSBE	Data Side Data Array Single Bit Error	
8	DDDBE	Data Side Data Array Double Bit Error	
9	SNPTSBE	Snoop Side Tag Array Single Bit Error	
10	SNPTDBE	Snoop Side Tag Array Double Bit Error	
11	SNPDSBE	Snoop Side Data Array Single Bit Error	
12	SNPDDBE	Snoop Side Data Array Double Bit Error	
13	FARM	Fixed Address Region Mismatch	
14:15		Reserved	
16	MSTRQE	PLB Master Port Request Error	
17	MSTRDE	PLB Master Port Read Data Error	
18	MSTRDPE	PLB Master Port Read Data Parity Error	
19	SNPAPE	PLB Snoop Port Address Parity Error	
20		Reserved	
21	SLVWDE	PLB Slave Port Write Data Error	
22	SLVWDPE	PLB Slave Port Write Data Parity Error	
23	SLVAPE	PLB Slave Port Address Parity Error	
24	SLVBEPE	PLB Slave Port Byte Enable Parity Error	
25	SLVDSBE	PLB Slave Port Data Array Single Bit Error	
26	SLVDDBE	PLB Slave Port Data Array Double Bit Error	
27	SLVIAE	PLB Slave Port Invalid Address Error Error	
28	SLVICE	PLB Slave Port Invalid Command Error Error	
29:30		Reserved	
31	DCRTO	DCR Timeout	

7.2.3 Debug Events and L2 Port Testing

The L2C core supports the following debug events.

- Single Bit ECC error debug event - Event will trigger if L2DBCR[SB] is set and the L2C detects a single bit ECC error in the array.
- Double Bit ECC error debug event - Event will trigger if L2DBCR[DB] is set and the L2C detects a double bit ECC error in the array.
- Snoop Address Compare 0 Kill debug event - Event will trigger if L2DBCR[SNPAC0K] is set and the L2C

receives a snoop kill request that matches the address contained in ERAPR and L2SNPAC0, the conditions specified by L2DBCR[SNPA0C] and the size specified by L2DBCR[SNPA0S].

- Snoop Address Compare 0 Flush debug event - Event will trigger if L2DBCR[SNPAC0F] is set and the L2C receives a snoop flush request that matches the address contained in ERAPR and L2SNPAC0, the conditions specified by L2DBGCR[SNPA0C] and the size specified by L2DBCR[SNPA0S].
- Snoop Address Compare 0 Push debug event - Event will trigger if L2DBCR[SNPAC0P] is set and the L2C receives a snoop push request that matches the address contained in ERAPR and L2SNPAC0, the conditions specified by L2DBCR[SNPA0C] and the size specified by L2DBCR[SNPA0S].
- Snoop Address Compare 0 PushE debug event - Event will trigger if L2DBCR[SNPAC0PE] is set and the L2C receives a snoop pushE request that matches the address contained in ERAPR and L2SNPAC0, the conditions specified by L2DBCR[SNPA0C] and the size specified by L2DBCR[SNPA0S].
- Snoop Address Compare 1 Kill debug event - Event will trigger if L2DBCR[SNPAC1K] is set and the L2C receives a snoop kill request that matches the address contained in ERAPR and L2SNPAC1, the conditions specified by L2DBCR[SNPA1C] and the size specified by L2DBCR[SNPA1S].
- Snoop Address Compare 1 Flush debug event - Event will trigger if L2DBCR[SNPAC1F] is set and the L2C receives a snoop flush request that matches the address contained in ERAPR and L2SNPAC1, the conditions specified by L2DBCR[SNPA1C] and the size specified by L2DBCR[SNPA1S].
- Snoop Address Compare 1 Push debug event - Event will trigger if L2DBCR[SNPAC1P] is set and the L2C receives a snoop push request that matches the address contained in ERAPR and L2SNPAC1, the conditions specified by L2DBCR[SNPA1C] and the size specified by L2DBCR[SNPA1S].
- Snoop Address Compare 1 PushE debug event - Event will trigger if L2DBCR[SNPAC1PE] is set and the L2C receives a snoop pushE request that matches the address contained in ERAPR and L2SNPAC1, the conditions specified by L2DBCR[SNPA1C] and the size specified by L2DBCR[SNPA1S].
- Slave Address Compare Read debug event - Event will trigger if L2DBCR[SLVAR] is set and the L2C receives a slave port read that falls within the L2C cache line specified by L2SLVERAPR and L2SLVAC0.
- Slave Address Compare Write debug event - Event will trigger if L2DBCR[SLVAR] is set and the L2C receives a slave port write that falls within the L2C cache line specified by L2SLVERAPR and L2SLVAC0.

7.2.3.1 L2 Debug Address Compare Mask Register (L2DBACMR)

This register, described in *Figure 5-48*, indicates which bits of the L2DBACR should be compared for L2 address compare debug events. At least one bit of this register must be set for any compares to occur.

L2 Indirect DCR number: 0x17

Privileged: Yes

Figure 7-37. L2 Debug Status Register (L2DBSR)

Bits	Mnemonic	Description	Notes
0:30	DBACM	Mask for bits 0-30 of the 36-bit real address to be compared	
31		Reserved	

7.2.3.2 L2 Debug Status Register (L2DBSR)

L2C debug status register described in *Figure 7-38* contains the status bits for L2C debug events. These bits are all set by hardware and set or cleared by software. The debug summary bit is set if any of the debug event bits are set.

L2 Indirect DCR number: 0x18
Privileged: Yes

Access: Read Clear or Read Set
Reset Value: Reset to zero

Figure 7-38. L2 Debug Status Register (L2DBSR)

Bits	Mnemonic	Description	Notes
0	DBGS	Debug Summary	
1	SB	Single Bit Error Debug Event	
2	DB	Double Bit Error Debug Event	
3:7		Reserved	
8	SNPA0K	Snoop Address Compare 0 Kill Event	
9	SNPA0F	Snoop Address Compare 0 Flush Event	
10	SNPA0P	Snoop Address Compare 0 Push Event	
11	SNPA0PE	Snoop Address Compare 0 PushE Event	
12:15		Reserved	
16	SNPA1K	Snoop Address Compare 1 Kill Event	
17	SNPA1F	Snoop Address Compare 1 Flush Event	
18	SNPA0P	Snoop Address Compare 1 Push Event	
19	SNPA1PE	Snoop Address Compare 1 PushE Event	
20:23		Reserved	
24	SLVAR	Slave Address Compare Read Event	
25	SLVAW	Slave Address Compare Write Event	
26:31		Reserved	

7.2.3.3 L2 Debug Control Register (L2DBCR)

L2 Debug Control Register described in *Figure 7-39* contains the enable bits for L2 debug events.

L2 Indirect DCR number: 0x19
Privileged: Yes
Access: R/W
Reset Value: All bits are reset to zero.

Figure 7-39. L2 Debug Control Register 1 (L2DBCR)

Bits	Mnemonic	Description	Notes
0		Reserved	
1	SB	Single Bit Error Debug Enable	
2	DB	Double Bit Error Debug Enable	
3:7		Reserved	
8	SNPA0K	Snoop Address Compare 0 Kill Enable	

Figure 7-39. L2 Debug Control Register 1 (L2DBCR) (continued)

Bits	Mnemonic	Description	Notes
9	SNPA0F	Snoop Address Compare 0 Flush Enable	
10	SNPA0P	Snoop Address Compare 0 Push Enable	
11	SNPA0PE	Snoop Address Compare 0 Push E Enable	
12	SNPA0S	Snoop Address Compare 0 Size 0 Snoop request matched coherency granule (32 Bytes) 1 Snoop request matched L2 line (128 Bytes)	
13		Reserved	
14:15	SNPA0C	Snoop Address Compare 0 Condition 00 - PLB Snoop request L2C hit not modified 01 - PLB Snoop request L2C hit modified 10 - PLB Snoop request L2C hit 11 - PLB Snoop request L2C miss (entire line if size = 1)	Note: The behavior of the debug logic for the values “00” and “01” is not always as expected. Snooping is initiated through the PLB for accesses from other CPUs. The snoop command can be either a snoop push or a snoop kill. In the event that the snoop command is a “snoop kill”, the L2 cache which is being snooped will report an unmodified type snoop hit regardless of whether the L2 had the line in a modified state at the time of receiving the snoop. Thus if L2DBCR[SNPA0C] == 00, a debug event will occur, while if L2DBCR[SNPA0C] == 01, a debug event will not occur.
16	SNPA1K	Snoop Address Compare 1 Kill Enable	
17	SNPA1F	Snoop Address Compare 1 Flush Enable	
18	SNPA1P	Snoop Address Compare 1 Push Enable	
19	SNPA1PE	Snoop Address Compare 1 PushE Enable	
20	SNPA1S	Snoop Address Compare 1 Size 0 - Snoop request matched coherency granule (32 Bytes) 1 - Snoop request matched L2 line (128 Bytes)	
21		Reserved	
22:23	SNPA1C	Snoop Address Compare 1 Condition 00 - PLB Snoop request L2 hit not modified 01 - PLB Snoop request L2 hit modified 10 - PLB Snoop request L2 hit 11 - PLB Snoop request L2 miss (entire line if size = 1)	**Note:** For any of these debug events trigger bits 0-27 of the snoop address must match the ERAPR
24	SLVAR	Slave Address Compare Read Enable	
25	SLVAW	Slave Address Compare Write Enable	
26:27		Reserved	
28	DBACEN	Debug Address Compare Enable	
29:31		Reserved	

7.2.3.4 Slave Address Compare 0 Register (L2SLVAC0)

Slave address compare register 0 described in *Figure 7-40* has the [43 to 56] address bits to be compared for Slave Address Compare 0 debug events.

L2 Indirect DCR number: 0x1C
Privileged: Yes
Access: R/W
Reset Value: Undefined

Figure 7-40. Slave Address Compare 0 Register (L2SLVAC0)

Bits	Mnemonic	Description	Notes
0:10		Reserved	
11:24	RAC	Bits 43-56 of the 64 bit real address (bits 0:29 of the 36 bit real address) to be compared	
25:31		Reserved	

7.2.3.5 Snoop Address Compare 0 Register (L2SNPAC0)

Snoop address compare register 0 described in *Figure 7-41* has the [28 to 58] address bits to be compared for Snoop Address Compare 0 debug events.

L2 Indirect DCR number: 0x1E
Privileged: Yes
Access: R/W
Reset Value: Undefined

Figure 7-41. Snoop Address Compare 0 Register (L2SNPAC0)

Bits	Mnemonic	Description	Notes
0:30	SNPRA	Bits 28-58 of the 64 bit real address (bits 0:29 of the 36 bit real address) to be compared	
31		Reserved	

7.2.3.6 Snoop Address Compare 1 Register (L2SNPAC1)

Snoop address compare register 1 described in *Figure 7-42* has the [28 to 58] address bits to be compared for Snoop Address Compare 1 debug events.

L2 Indirect DCR number: 0x1F
Privileged: Yes
Access: R/W
Reset Value: Undefined

Figure 7-42. Snoop Address Compare 1 Register (L2SNPAC1)

Bits	Mnemonic	Description	Notes
0:30	SNPRA	Bits 28-58 of the 64 bit real address to be compared	
31		Reserved	

7.2.3.7 L2 Debug Address Compare Status Register (L2DBACSR)

The L2 Debug Address Compare Status Register described in Figure 7-43 contains address match detection bits.

Figure 7-43. L2 Debug Address Compare Status Register (L2DBACSR)

Bits	Mnemonic	Description	Notes
0	ISRDQM	Instruction Side Read Queue Match	
1	DSRDQM	Data Side Read Queue Match	
2	DSWRQ0M	Data Side Write Queue 0 Match	
3	DSWRQ1M	Data Side Write Queue 1 Match	
4	LOOKUPM	Lookup Match	
5	STOBUFM	Store Buffer Match	
6	CASTOUTM	Castout Buffer Match	
7	FB0M	Fill Buffer 0 Match	
8	FB1M	Fill Buffer 1 Match	
9	FB2M	Fill Buffer 2 Match	
10	FB3M	Fill Buffer 3 Match	
11	SB0M	Snoop Buffer 0 Match	
12	SB1M	Snoop Buffer 1 Match	
13:31		Reserved	

7.2.3.8 L2 Debug Address Compare Control Register (L2DBACCR)

The L2 Debug Address Compare Control Register described in Figure 7-44 contains address match detection bits.

Figure 7-44. L2 Debug Address Compare Control Register (L2DBACCR)

Bits	Mnemonic	Description	Notes
0	ISRDQEN	Instruction Side Read Queue Enable	
1	DSRDQEN	Data Side Read Queue Enable	
2	DSWRQ0EN	Data Side Write Queue 0 Enable	
3	DSWRQ1EN	Data Side Write Queue 1 Enable	
4	LOOKUPEN	Lookup Enable	
5	STOBUFEN	Store Buffer Enable	
6	CAS-TOUTEN	Castout Buffer Enable	
7	FB0EN	Fill Buffer 0 Enable	
8	FB1EN	Fill Buffer 1 Enable	

Figure 7-44. L2 Debug Address Compare Control Register (L2DBACCR) (continued)

Bits	Mnemonic	Description	Notes
9	FB2EN	Fill Buffer 2 Enable	
10	FB3EN	Fill Buffer 3 Enable	
11	SB0EN	Snoop Buffer 0 Enable	
12	SB1EN	Snoop Buffer 1 Enable	
13:31		Reserved	

7.2.3.9 L2 Debug Address Compare Register (L2DBACR)

This register, described in *Figure 7-45* contains the address bits to be compared for L2 address compare debug events. Only bits selected for comparison by the mast register L2DBACMR are compared.

Figure 7-45. L2 Debug Address Compare Register (L2DBACR)

Bits	**Mnemonic**	**Description**	**Notes**
0:30	DBAC	Bits 0-30 of the 36 bit real address to be compared	
31		Reserved	

7.3 L2C Processor Local Bus (PLB) Ports

The sections that follow describe the PLB master port, snoop port, and slave port.

7.3.1 L2 PLB Master Port

The L2 master interface provides a separate read address/data and a separate write address/data port on the PLB bus to initiate transactions from the PPC465 processor core to other devices and to the memory system. The L2C PLB master port is considered a split transaction bus and is designed to allow both read and write requests and data to proceed across the PLB nearly simultaneously to access data from/to the memory unit or other devices. The master port also supports snoop protocol in that it initiates the transactions and supplies the castout or write-back data as a result of snoop port hits. The L2C also uses the master port to generate address-only snoop requests to other snooping masters.

A Memory Coherent read request from the PPC465 processor with L2 cache when attached to the a PLBv5 core-connect compliant arbitration unit will proceed as follows: The PLB Arbiter receives the request and issues a request to the memory slave location to access the data and also issues a snoop request to all other snooping masters except the original requestor. The slave can access the data but is not allowed to transfer it back to the requestor until the snoop is complete and any castout data has been taken care of. Further details on PLB Memory coherency sequence will be found in PLB(v5) Architectural specifications.

7.3.1.1 Request Pipelining

The L2C PLB master logic is capable of pipelining to a depth of 16 total requests out of a maximum of 32 requests handling capability that the PLB(v5) architecture supports for its masters. Types of those requests and combinations are listed below:

1. Up to 4 burst or line fills in which the L2C is requesting data,
2. Up to 16 address only flushes generated by a Promote to Exclusive request,
3. One mbar or msync command,
4. Up to 16 PLB coherent and non-snoopable write requests,
   PLB coherent and non-snoopable write requests are:
   - L2C castouts,
   - Snoop castouts,
   - Stores marked by the L1D as internally snoopable,
   - DCBF / DBCST with L1D data to M = 1 storage space,
   - L2C store with “Write-Thru Hit Exclusive” case.
5. One PLB snoopable or non-coherent write request
   PLB snoopable or non-coherent write requests are:
   - Stores to M= 0 storage space,
   - Stores to M= 1 storage space that are marked internally non-snoopable,
   - Cache op generated address only Ops with M= 1 storage space,
   - DCBF / DBCST with L1D data to M = 0 storage space,
   - tlbivax / tlbsync (Not supported in PPC465 core).

7.3.1.2 Request Types

Table 7-18 describes valid request types from the L2C to the PLB.

Table 7-18. Valid PLB Request Types

Read / Write	Type	Size(s) Bytes	Command	PostponeEn	OutOfOrder-DataEn	CastoutReq
Read (always target word first)	Burst	1 2, 3 (will never cross 4 B boundary) 4 (4B aligned) 8 (8B aligned) 16 (16B aligned)	Non-snoopable	1	1	0
			Read with no intent to cache	1	1	0
			Read with intent to modify	1	1	0
			Read with push	1	1	0
	Line	32 64 128	Non-snoopable	1	1	0
			Read with no intent to cache	1	1	0
			Read with intent to modify	1	1	0
			Read with push	1	1	0
Write	Burst	1-16,32 (will never cross a 32 Byte boundary)	Non-snoopable	0	—	0 / 1
			Write with flush	1	—	0
		64 96 128	Non-snoopable	0	—	0 / 1

Table 7-18. Valid PLB Request Types (continued)

Read / Write	Type	Size(s) Bytes	Command	PostponeEn	OutOfOrder-DataEn	CastoutReq
Address-Only	—	32	Kill	1	—	-
	—		Flush	1	—	-
	—		Push	1	—	-
Command	—	—	MSYNC	1	—	-
Note: The reorder enable feature of PLB(v5) is not implemented in L2C. Thus, “Mn_reorderEn” should be tied to zero at the input to the PLB core. **Note:** The L2C will generate parity for outgoing data if the PLB Parity Enable bit, L2CR3[PLBPE], is set. **Note:** The L2C will check parity for incoming data if the PLB Parity Enable bit, L2CR3[PLBPE], is set, and if parity is enabled in the PLB core.						

7.3.1.3 Read Request Size

The L2C will allocate its cache line and determine PLB read request size based on L1 cache command type. Table 7-19 the L1 cache command type and PLB read request size relations.

Table 7-19. L1 Cache Command Type and PLB Read Request Size Relations

L1 Cache Command	IL2	TLB M bit	L2CR1[RAS or TAS]	PLB Read Request Size (Data Bits)
L1core I-cache Read	1	—	—	2
	0			Based on *Table*
	0	—	1	2
L1core I-cache Touch	1		—	2
	0	—		8
	0		1	2
L1core D-cache Read	1	—	—	1 or 2 (L1 request size)
	0			Based on *Table*
	0	—	1	2
L1core D-cache Touch	1		—	2
	0	—		8
	0		1	2
L1core D-cache Store (See notes below)	1	—	—	n/a
	0	0		Based on *Table*
	0	—	1	2
	0		—	2
Note: L1core D-cache stores that are without allocate will not make a PLB read request if they miss the L2C. **Note:** L1core D-cache stores that are for a 32-Byte granule (L1core D-cache castout) and are allocating will never make a PLB read, but will just write the 32-Bytes of data to the target granule. Table 5-16. Size of Read Requests from L2C to PLB for Allocating Ops				

Table 7-20. Size of Read Requests from L2C to PLB for Allocating Ops

Target Granule	Granule 0 Valid	Granule 1 Valid	Granule 2 Valid	Granule 3 Valid	Target Address (Granule)	Request Size (Bytes)	Granules Filled
0	0	0	0	0	0	128	0, 1, 2, 3
0	0	0	0	1	0	64	0, 1
0	0	0	1	0	0	64	0, 1
0	0	0	1	1	0	64	0, 1
0	0	1	0	0	0	32	0
0	0	1	0	1	0	32	0
0	0	1	1	0	0	32	0
0	0	1	1	1	0	32	0
0	1	0	0	0	No Request		
0	1	0	0	1			
0	1	0	1	0			
0	1	0	1	1			
0	1	1	0	0			
0	1	1	0	1			
0	1	1	1	0			
0	1	1	1	1			
1	0	0	0	0	1	128	0, 1, 2, 3
1	0	0	0	1	1	64	0, 1
1	0	0	1	0	1	64	0, 1
1	0	0	1	1	1	64	0, 1
1	0	1	0	0	No Request		
1	0	1	0	1			
1	0	1	1	0			
1	0	1	1	1			
1	1	0	0	0	1	32	1
1	1	0	0	1	1	32	1
1	1	0	1	0	1	32	1
1	1	0	1	1	1	32	1
1	1	1	0	0	No Request		
1	1	1	0	1			
1	1	1	1	0			
1	1	1	1	1			
2	0	0	0	0	2	128	0, 1, 2, 3

Table 7-20. Size of Read Requests from L2C to PLB for Allocating Ops (continued)

Target Granule	Granule 0 Valid	Granule 1 Valid	Granule 2 Valid	Granule 3 Valid	Target Address (Granule)	Request Size (Bytes)	Granules Filled
2	0	0	0	1	2	32	2
2	0	0	1	0			
2	0	0	1	1			
2	0	1	0	0	2	64	2, 3
2	0	1	0	1	2	32	2
2	0	1	1	0	No Request		
2	0	1	1	1			
2	1	0	0	0	2	64	2, 3
2	1	0	0	1	2	32	2
2	1	0	1	0	No Request		
2	1	0	1	1			
2	1	1	0	0	2	64	2, 3
2	1	1	0	1	2	32	2
2	1	1	1	0	No Request		
2	1	1	1	1			
3	0	0	0	0	3	128	0, 1, 2, 3
3	0	0	0	1	No Request		
3	0	0	1	0	3	32	3
3	0	0	1	1	No Request		
3	0	1	0	0	3	64	2, 3
3	0	1	0	1	No Request		
3	0	1	1	0	3	32	3
3	0	1	1	1	No Request		
3	1	0	0	0	3	64	2, 3
3	1	0	0	1	No Request		
3	1	0	1	0	3	32	3
3	1	0	1	1	No Request		
3	1	1	0	0	3	64	2, 3
3	1	1	0	1	No Request		
3	1	1	1	0	3	32	3
3	1	1	1	1	No Request		

7.3.1.4 Number and size of PLB Write Requests: Replacement Castouts

It is possible to cause an L2 castout with 1, 2, 3, or 4 dirty 32-byte segments because the L2C cache lines are segmented. Thus, the number, size and starting address of PLB requests varies for castouts. The L2C will never castout data if none of the four granules are modified. *Table 7-21* describes how the castout is presented to the PLB.

Table 7-21. Number, Size and Address of PLB Requests from Castouts

Granule 0	Granule 1	Granule 2	Granule 3	Number of Requests	Size of Request(s) (Bytes)	Granule Address(es) of Requests
0	0	0	0	no castout request		
0	0	0	M	1	32	3
0	0	M	0	1	32	2
0	0	M	M	1	64	2
0	M	0	0	1	32	1
0	M	0	M	2	32, 32	1, 3
0	M	M	0	1	64	1
0	M	M	M	1	96	1
M	0	0	0	1	32	0
M	0	0	M	2	32, 32	0, 3
M	0	M	0	2	32, 32	0, 2
M	0	M	M	2	32, 64	0, 2
M	M	0	0	1	64	0
M	M	0	M	2	64, 32	0, 3
M	M	M	0	1	96	0
M	M	M	M	1	128	0
Note: M denotes the segment is modified, “0” denotes segment is not modified.						

7.3.1.5 Number and size of PLB Write Requests: Snoop Castouts

It is possible to have a snoop castout with 1, 2, 3, or 4 dirty 32-byte granules because the L2C cache lines have segmented modified bits, and the L1core D-cache can deliver snoop castout data to the L2C. Thus, the number, size, and starting address of PLB requests varies for snoop castouts. Table 7-22 on page 211 describes how the castout is presented to the PLB.

Table 7-22. Number, Size, and Address of PLB Requests from Snoop Castouts

Granule	Granule 1	Granule2	Granule3	# of Requests	Size of request(s)	Address of Request(s)
0	0	0	0	0	n/a	n/a
0	0	0	M	1	32	3
0	0	M	0	1	32	2
0	0	M	M	1	64	2
0	M	0	0	1	32	1
0	M	0	M	2	32, 32	1, 3
0	M	M	0	1	64	1
0	M	M	M	1	96	1
M	0	0	0	1	32	0
M	0	0	M	2	32, 32	0, 3
M	0	M	0	2	32, 32	0, 2
M	0	M	M	2	32, 64	0, 2
M	M	0	0	1	64	0
M	M	0	M	2	64, 32	0, 3
M	M	M	0	1	96	0
M	M	M	M	1	128	0

Note: M indicates segment is modified in the L2, 0 indicates segment is not modified or exclusive.
Note: This table should be applied on all snoop castouts after the L1 and L2 data have been merged.

There will be two modes for snoop castouts: high performance and low power. The mode is chosen by L2CR2[SNPCM], bit-2. In high performance mode, if the snoop request is wholly contained within an L2C cache line, the L2C has at least one granule modified, and the L2C contains valid granules for the entire snoop, then the L2C will make one castout request matching the size of the snoop request. If the above conditions are not satisfied or the L2C is in low power snoop castout mode, then the L2C will only castout modified granules.

7.3.1.6 Request Priority

The L2C has a time based priority to process the requests to the PLB for the snoop castout buffers, replacement castout buffer, and Store-Buffer or Fill-Buffers.

The L2C will assign bus priorities to PLB requests based on the following fields in L2CR3[MSTIRP, MSTDRP, MSTDWP, MSTCOP]. The L2C will choose the value to assign to the request based on the request type. The four types of request priorities are for L1core I-cache reads, L1core D-cache reads, L1core D-cache writes, and L2C castouts.

7.3.2 L2 PLB Snoop Port

The L2C has a PLB(v5) compliant snooping port that works with the L2C Master PLB port to provide memory coherency for the L1/L2 caches. This snoop port logic accepts snoop requests from the PLB and tracks the PLB snoop activities by checking the L2C states and forwarding the snoop requests to the L1core data cache for cache line invalidation or for reservation stucx) granule invalidation. The results of the snoop is reported back onto the PLB by snoop port status signals on whether it is a miss, a hit, or a modified line. If castout data is required from the snoop, then it is internally queued up and it is the L2’s master port that initiates the castout transaction and puts the castout or pushed data onto the PLB. The snoop port receives its snoop requests from other devices in the system. Since a locally generated L2 cache miss request for data already missed the L2 (and L1Data due to inclusiveness) there is no reason to snoop back into this L2. The L2 snoop can result in forwarding the snoop to the L1 Data-side cache for line invalidation. The L1 invalidation from the snoop can be for a single cache line or up to 8 cache lines depending on the snoop command and qualifiers. Because snoops occur on coherence granules and a coherence granule, which is 32 bytes, the lower 5 bits are not used. Thus a snoop address is 59 bits (64-5) in the 64-bit space. In the PPC465 Processor Complex, only 36 address bits are used, so that a snoop address is 31 bits (real address 0:31).

The L2C snoop port is comprised of inputs such as a 59-bit real address, 4-bit command, 3-bit count, 6-bit master_id, and a 4-bit tag (plus other signals) to help track the PLB transaction and identify any castout data back to the initiating master. Snoop output results include 8-bits of response (snoop completed), 8-bits of hit status, and 8-bits of castout/push status needed to identify the state of up to 8 cache line granules (depending if it is a line or burst transaction and the count). The L2 snoop port is capable of accepting up to two pending snoop requests. There are more details on snooping in Section 7.1.6 , “Memory Coherence” on page 170. The PPC465 Core Support Manual has a very detailed section on snooping and its related commands and signal functions. Also, please refer to the PLB(v5) specifications for memory coherency operation and signal definitions and sequences.

7.3.3 L2 PLB Slave Port

The L2C has a PLB(v5) compliant slave port integrated within it. The L2 slave port is enabled only when the L2 is in Fixed Address Mode, FAM, (half or full) and the Fixed Address Mode is configured as public accessable. This is to allow external (PLB) reads and writes to access the FAM memory. Note that the PPC465 core accesses to the FAM are only generated on the L1 to L2 internal buses and do not get broadcast onto the PLB bus. By providing this dual-ported structure, the FAM provides low latency access from PPC465 core to system memory by having a local memory area that does not require snoop delays. The public access to the FAM memory through the PLB Slave port allows rapid data transfer capability to the local on-chip storage by DMA devices such as a Ethernet or PCI masters.

There are two main requirements for use of this memory in public mode: the local L1 is the only device in the system allowed to cache this memory space, and the TLB storage attribute M must be zero. This allows the PPC465 core and any masters on the PLB to access this array without snooping on the PLB. It is designed to benefit the system performance by providing a local memory without bus snooping.

7.3.3.1 PLB Slave Requests

The slave port logic has a 3 entry deep FIFO queue (address tenure queue) which takes requests from the PLB. The slave port can operate on both Intra-Address Pipelined (IAP) and non-IAP ways. See the table below for this port behaviors on PLB requests. The queue is always processed in order. The queue may contain any mix of PLB reads and writes. PLB address tenure responses are only made from Queue entry 0; Address-Acknowledged (AAcked) read and write requests then move to the data tenure queue, which is 2 entries deep.

Table 7-23. Slave Port Behaviors (responses) on PLB Requests

IAP Enable	Reset	Array Size == 0	FAM Enable	Public FAM	FAR Match	Request Snoopable	Req Parity Error	Write Request	Write Buffer Available	Behaviors (responses)
0	1	—	—	—	—	—	—	—	—	No PLB Response No-op request
0	—	1	—	—	—	—	—	—	—	No PLB Response No-op request
0	—	—	0	—	—	—	—	—	—	No PLB Response No-op request
0	—	—	-	0	—	—	—	—	—	No PLB Response No-op request
0	—	—	-	-	0	—	—	—	—	No PLB Response No-op request
0	0	0	1	1	1	1	—	—	—	Respond ReqErr to PLB No-op request
0	0	0	1	1	1	0	1	—	—	Respond ReqErr to PLB No-op request
0	0	0	1	1	1	0	0	0	—	AddrAck to PLB Perform read
0	0	0	1	1	1	0	0	1	0	AddrAck to PLB WrDRdy when buffer available, Perform write when all data received
0	0	0	1	1	1	0	0	1	1	AddrAck to PLB, WrDRdy, Perform write when all data received
1	1	—	—	—	—	—	—	—	—	No Response to PLB No-op request
1	—	0	—	—	—	—	—	—	—	Respond ReqErr to PLB No-op request
1	—	—	0	—	—	—	—	—	—	Respond ReqErr to PLB No-op request
1	—	—	—	0	—	—	—	—	—	Respond ReqErr to PLB No-op request
1	—	—	—	—	0	—	—	—	—	Respond ReqErr to PLB No-op request
1	—	—	—	—	—	1	—	—	—	Respond ReqErr to PLB No-op request
1	—	—	—	—	—	—	1	—	—	Respond ReqErr to PLB No-op request
1	0	0	1	1	1	0	0	0	—	AddrAck to PLB Perform read

Table 7-23. Slave Port Behaviors (responses) on PLB Requests (continued)

IAP Enable	Reset	Array Size == 0	FAM Enable	Public FAM	FAR Match	Request Snoopable	Req Parity Error	Write Request	Write Buffer Available	Behaviors (responses)
1	0	0	1	1	1	0	0	1	0	AddrAck to PLB WrDRdy when buffer available, Perform write when all data received
1	0	0	1	1	1	0	0	1	1	AddrAck to PLB WrDRdy Perform write when all data received

Descriptions of columns of above table:

- IAP Enable - This is a primary input from the PLB macro to the slave port that tells the slave port if it is on an Intra-address Pipelined (IAP) way or a non-IAP way. A slave device on an IAP way must respond to all requests it sees. A slave device on a non-IAP way must only respond to requests that fall within its address region.
- Reset - Indicates that the L2C is currently being reset.
- Array Size == 0 - This describes whether or not the L2C array size is set to something other than zero. An array size of zero means that no array is present.
- FAM Enable - This describes whether or not the L2C array is configured for Fixed Address Mode or not. There are 2 bits in the L2CR1[FAMES] that enable the FAM and its size. In this table all we care about is if FAM is enabled so this check is the OR of the 2 bits in the L2CR1[0:1].
- Public FAM - There is a bit in the L2CR1[PUBFAM] that indicates whether the Fixed Address Region (if enabled) is public or private. Devices in the system are only allowed to access the FAR through the slave port when the FAR is public.
- FAR Match - This describes whether the PLB request matches the Fixed Address Region or not. The FAR is the slave port's memory region.
- Request Snoopable - This is a transfer qualifier from the PLB that is sent with the request to the slave. Data from the L2C FAR cannot be cached and modified by any other device in the system, so all requests to this memory range must be marked as non-snoopable.
- Req Parity Error - When a request is made by the PLB to the slave port, it comes with parity bits for the address and byte enables. This parity is used to detect bit flips in these buses between the requesting master and the slave.
- Write Request - This describes whether or not the current request is a write or a read.
- Write Buffer Available - This describes whether or not the slave port's write buffer able to accept data from a new write or if it is still in use from the last write.

Figure 7-24 describes each of the three responses by signals on a PLB Slave request made, shown in *Figure 7-23*

Table 7-24. Slave Address Tenure PLB Slave Response Signals

PLB Signal	Reason
slavPlbReqErr	Non-IAP, if L2SLVERAPR & FAMAR match and: - Address Parity Error - Byte Enable Parity Error - Command Type Error IAP: - L2SLVERAPR doesn't match - FAMAR doesn't match - Address Parity Error - Byte Enable Parity Error - Command Type Error
slavPlbRearbitrate	Non-IAP, if not a reqErr and: - Msync/Mbar with a write in the data Q - Both data tenure queue entries valid IAP, if not a reqErr and: - Msync/Mbar with a write in the data Q
slavPlbAddrAck	Non-IAP, if not a reqErr or reArb case and: - Not an Mbar or MSync and there is at least 1 free data tenure Q entry IAP, if not a reqErr or reArb case and: - There is at least 1 free data tenure Q entry

7.3.3.2 PLB Slave Port Processes for Request Types

This section describes the PLB slave port processes for each PLB slave request type.

Reads

PLB reads may be up to 256 bytes in length. When the L2C slave port receives a read, it is placed in the first available entry of the PLB Slave Address Tenure Queue. When it reaches the head of the Address Tenure Queue, the read will respond to the PLB slave bus.

If it responds with AAck, the request will move into the first available entry of the PLB Slave Data Tenure Queue. When the request reaches the head of the Data Tenure Queue, it will allocate its read buffers and an entry in the Read Buffer Queue. The PLB slave port logic 'subunit' has two read buffers of 128 bytes each.

After the buffers are allocated, the read will make its request to the L2C array. The second array request is made the cycle after the first is accepted in a consecutive case. The data from the reads is captured in the buffers. Once a buffer has its data, the buffer will start sending its data out to the PLB.

If the Target Word First is enabled, the two read requests may be performed out of order.

Once the final read request has been made into the L2C array, the read request will leave the PLB slave request queue. The read buffer queue is responsible for sending the data out.

Writes

PLB writes may be up to 256 bytes in length. When the L2C slave port logic receives a write, it is placed in the first available entry of the PLB Slave Address Tenure Queue. When it reaches the head of the Address Tenure Queue, the write will respond to the bus.

If it responds with AAck, the request will move into the first available entry of the PLB Slave Data Tenure Queue. It then allocates its write buffers as soon as enough buffers are available. The slave port logic ‘subunit’ has two write buffers, each 128 bytes in size. After allocating write buffers, it will respond to the PLB slave with “Write-Data-accept-Ready”, wrDRdy.

After the buffers have received all their data, and the write is at the head of the Data Tenure Queue, it makes a request to the L2C array. If needed, and second request is made for the 2nd line of the request. A write buffer deallocates when its data has been captured by the L2C array. The write deallocates from Queue Entry 0 once the final request has been accepted by the L2C array.

mbar** and **msync

When the L2C slave port logic receives an mbar or msync, it is placed in the first available entry of the PLB Slave Address Tenure Queue. When it reaches the head of the Address Tenure Queue, the **mbar** or **msync** will respond to the bus. It will respond with a “re-try”, Re-arb, if there are any writes in the Data Tenure Queue, otherwise it will send Address-Acknowledge, AddrAck.

After responding to the PLB slave, the mbar or msync deallocates the Queue; An mbar or msync can never enter the Data Tenure Queue.

PLB Slave Address Tenure Queue

The PLB Slave Port Logic subunit contains a three entry Address Tenure Queue. This queue is used to hold requests from the PLB slave bus until their address tenure is complete. This is the time between request and acknowledge (Addr Ack, Error Ack, or Re-try/Re-arb). The addresses requested are latched into the lowest numbered available queue entry. Responses are made to the PLB slave only from queue entry 0. After responding to the PLB slave, AddrAck reads and writes move to the Data Tenure Queue.

PLB Slave Data Tenure Queue

The PLB Slave Logic ‘subunit’ contains a two entry deep Data Tenure Queue. This queue is used to hold data of read and write requests that have completed their address tenure but have not yet accessed the L2C array. Writes will collect their data from the PLB slave. Data of Requests are latched into the lowest numbered queue entry first. Requests are made to the L2C array from entry zero. After the final request is made to the array, writes are deallocated and reads move to the read data queue.

Write Buffers

There are two write buffers of 128 bytes wide each. These buffers are used to hold up to 256 bytes of write data for writes in the Data Tenure Queue. They are allocated when a write request arrives in the Data Tenure Queue. These buffers are aligned on a 128 byte boundary. This means that two different write requests (each contained within a 128 byte line) could be using the buffers simultaneously. These buffers are deallocated after their contents are written into the L2C array.

Read Data Queue

The PLB Slave Logic has a 2 entry deep FIFO queue for read data. This queue keeps track of read requests that have left the PLB slave Data Tenure Queue but not yet sent all their data out to the bus. This queue is allocated when the read request reaches the head of the PLB slave Data Tenure Queue.

Read Buffers

There are two read buffers, each 128 bytes wide. These buffers are used to hold up to 256 bytes of read data for a read in the Data Tenure Queue and Read Data Queue. They are allocated when a read make a request to the L2C array. These buffers are aligned on a 128 byte boundary. This means that two different read requests (each contained within a 128 byte line) could be using the buffers simultaneously. These buffers are deallocated after their contents are written to the PLB slave.

7.3.3.3 L2C PLB Slave Error Reporting Registers

The following section details L2 DCR registers associated with the error reporting for the L2 PLB slave port. The error status is captured (and cleared) in the L2MCSR and enabled in the L2MCRER registers.

PLB Slave Port Error Address Register 0 (L2SLVEAR0)

PLB slave port error address register 0 is updated with the address of the slave port request that caused an error to be signaled to the PLB. *Figure 7-46* describes L2SLVEAR0 register bits.

Possible errors are:
- Write Data Error
- Write Data Parity Error
- Request Address Parity Error
- Request Byte Enable Parity Error
- Request Invalid Address Error
- Request Invalid Command Error

Figure 7-46. PLB Slave Port Error Address Register 0 (L2SLVEAR0)

Bits	Mnemonic	Description	Notes
0:31		Bits 0-31 of the 64 bit real address	

PLB Slave Port Error Address Register 1 (L2SLVEAR1)

PLB slave port error address register 1 is updated with the address of the slave port request that caused an error to be signaled to the PLB. *Figure 7-47* describes L2SLVEAR1 register bits.

Possible errors are:
- Write Data Error
- Write Data Parity Error
- Request Address Parity Error
- Request Byte Enable Parity Error
- Request Invalid Address Error
- Request Invalid Command Error

Figure 7-47. PLB Slave Port Error Address Register 1 (L2SLVEAR1)

Bits	Mnemonic	Description	Notes
0:31		Bits 32-63 of the 64 bit real address.	This address will be aligned with the request size for line types.

PLB Slave Port Error Master ID Register (L2SLVMIDR)

PLB slave port error master ID register is updated with the master ID of the slave port request that caused an error to be signaled to the PLB. *Figure 7-48* describes L2SLVMIDR register bits.

Possible errors are:
- Write Data Error

- Write Data Parity Error
- Request Address Parity Error
- Request Byte Enable Parity Error
- Request Invalid Address Error
- Request Invalid Command Error

Figure 7-48. Slave Port Error Master ID Register (L2SLVMIDR)

Bits	Mnemonic	Description	Notes
0:1		Reserved	
2:7	MID	Master ID	
8:11		Reserved	
12:15	RTAG	Request Tag	
16	RNW	Request RNW	
17:31		Reserved	

8. Memory Management

The PPC465 supports a uniform, 4 GB effective address (EA) space, and a 64 GB (36-bit) real address (RA) space. The PPC465 memory management unit (MMU) performs address translation between effective and real addresses, as well as protection functions. With appropriate system software, the MMU supports:

- Translation of effective addresses into real addresses
- Software control of the page replacement strategy
- Page-level access control for instruction and data accesses
- Page-level storage attribute control

8.1 MMU Overview

The PPC465 generates effective addresses for instruction fetches and data accesses. An effective address is a 32-bit address formed by adding an index or displacement to a base address (see *Effective Address Calculation* on page 42). Instruction effective addresses are for sequential instruction fetches, and for fetches caused by changes in program flow (branches and interrupts). Data effective addresses are for load, store and cache management instructions. The MMU expands effective addresses into virtual addresses (VAs) and then translates them into real addresses (RAs); the instruction and data caches use real addresses to access memory.

The PPC465 MMU supports demand-paged virtual memory and other 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. The hardware provides sufficient information to correct the fault and restart the faulting instruction.

The MMU divides storage into pages. The page represents the granularity of address translation, access control, and storage attribute control. PowerPC Book-E architecture defines 16 page sizes, of which the PPC440H6 MMU supports nine. These nine page sizes (1KB, 4KB, 16KB, 64KB, 256KB, 1MB, 16MB, 256MB, and 1GB) are simultaneously supported. The various page sizes supported are supported simultaneously. A valid entry for a page referenced by an effective address must be in the translation look aside buffer (TLB) in order for the address to be accessed. An attempt to access an address for which no TLB entry exists causes an Instruction or Data TLB Error interrupt, depending on the type of access (instruction or data). See *Interrupts and Exceptions* on page 247 for more information on these and other interrupt types.

The TLB is parity protected against soft errors. If such errors are detected, the CPU can be configured to vector to the machine check interrupt handler, where software can take appropriate action. The details of parity checking and suggested interrupt handling are described below.

8.1.1 Support for PowerPC Book-E MMU Architecture

The Book-E Enhanced PowerPC Architecture defines specific requirements for MMU implementations, but also leaves the details of several features implementation-dependent. The PPC465 is fully compliant with the required MMU mechanisms defined by PowerPC Book-E, but a few optional mechanisms are not supported. These are:

- TLB Invalidate virtual address (**tlbiva**) instruction

  The **tlbiva** instruction is used to support the invalidation of TLB entries in a multiprocessor environment with hardware-enforced coherency, which is not supported by the PPC465. Consequently, the attempted execution of this instruction will cause an Illegal Instruction exception type Program interrupt. The **tlbwe** instruction may be used to invalidate TLB entries in a uniprocessor environment.

- TLB Synchronize (**tlbsync**) instruction

  The **tlbsync** instruction is used to synchronize software TLB management operations in a multiprocessor environment with hardware-enforced coherency, which is not supported by the PPC465. Consequently, this instruction is treated as a no-op.

- Page Sizes

  PowerPC Book-E defines sixteen different page sizes, but does not require that an implementation support all of them. Furthermore, some of the page sizes are only applicable to 64-bit implementations, as they are larger than a 32-bit effective address space can support (4GB). Accordingly, the PPC465 supports nine of the sixteen page sizes, from 1KB up to 1GB, as mentioned above and as listed in *Table 8-2 Page Size and Effective Address to EPN Comparison* on page 227.

- Address Space

  Since the PPC465 is a 32-bit implementation of the 64-bit PowerPC Book-E architecture, there are differences in the sizes of some of the TLB fields. First, the Effective Page Number (EPN) field varies from 4 to 22 bits, depending on page size. Second, the page number portion of the real address is made up of a concatenation of two TLB fields, rather than a single Real Page Number (RPN) field as described in PowerPC Book-E. These fields are the RPN field (which can vary from 4 to 22 bits, depending on page size), and the Extended Real Page Number (ERPN) field, which is 4 bits, for a total of 36 bits of real address, when combined with the page offset portion of the real address. See *Address Translation* on page 227 for a more detailed explanation of these fields and the formation of the real address.

8.2 Translation Look Aside Buffer

The Translation Look Aside Buffer (TLB) is the hardware resource that controls translation, protection, and storage attributes. A single unified 64-entry, fully-associative TLB is used for both instruction and data accesses. In addition, the PPC465 implements two separate, smaller "shadow" TLB arrays, one for instruction fetch accesses and one for data accesses. These shadow TLBs improve performance by lowering the latency for address translation, and by reducing contention for the main unified TLB between instruction fetching and data storage accesses. See *Shadow TLB Arrays* on page 239 for additional information on the operation of the shadow TLB arrays.

Maintenance of TLB entries is under software control. System software determines the TLB entry replacement strategy and the format and use of any page table information. A TLB entry contains all of the information required to identify the page, to specify the address translation, to control the access permissions, and to designate the storage attributes.

A TLB entry is written by copying information from a GPR and the MMUCR[STID] field, using a series of three **tlbwe** instructions. A TLB entry is read by copying the information into a GPR and the MMUCR[STID] field, using a series of three **tlbre** instructions. Software can also search for specific TLB entries using the **tlbsx[.]** instruction. See *TLB Management Instructions* on page 240 for more information on these instructions.

Each TLB entry identifies a page and defines its translation, access controls, and storage attributes. Accordingly, fields in the TLB entry fall into these categories:

- Page identification fields (information required to identify the page to the hardware translation mechanism).
- Address translation fields
- Access control fields
- Storage attribute fields
- Cacheability

Table 8-1 summarizes the TLB entry fields for each of the categories.

Table 8-1. TLB Entry Fields

TLB Word	Bit	Field	Description
		Page Identification Fields	
0	0:21	EPN	**Effective Page Number** (variable size, from 4 - 22 bits) Bits 0:n–1 of the EPN field are compared to bits 0:n–1 of the effective address (EA) of the storage access (where n = 32–$\log_2$(page size in bytes) and page size is specified by the SIZE field of the TLB entry). See *Table 8-2* on page 227.
0	22	V	**Valid** (1 bit) This bit indicates that this TLB entry is valid and may be used for translation. The Valid bit for a given entry can be set or cleared with a **tlbwe** instruction.
0	23	TS	Translation Address Space (1 bit) This bit indicates the address space this TLB entry is associated with. For instruction fetch accesses, MSR[IS] must match the value of TS in the TLB entry for that TLB entry to provide the translation. Likewise, for data storage accesses (including instruction cache management operations), MSR[DS] must match the value of TS in the TLB entry. For the **tlbsx**[.] instruction, the MMUCR[STS] field must match the value of TS.
0	24:27	SIZE	**Page Size** (4 bits) The SIZE field specifies the size of the page associated with the TLB entry as 4^{SIZE}KB, where SIZE $\in$ {0, 1, 2, 3, 4, 5, 7, 9, 10}. Since this field is the encoded value of a field in the UTLB, if there is a parity error in the decoded size (DSIZ) field of the UTLB entry, the value of this fields is UNDE-FINED following a **tlbre**. See *Table 8-2* on page 227.
0	28:31	TPAR	**Tag Parity** (4 bits) These four bits are the parity read from the TLB array; they corresponds to the parity written into the array as tag parity[0:3]. Set conditionally based on CCR0[CRPE].
0	32:39	TID	**Translation ID** (8 bits) Field used to identify a globally shared page (TID=0) or the process ID of the owner of a private page (TID<>0). See *Page Identification* on page 224.
		Address Translation Fields	
1	0:21	RPN	**Real Page Number** (variable size, from 4 - 22 bits) Bits 0:n–1 of the RPN field are used to replace bits 0:n–1 of the effective address to produce a por-tion of the real address for the storage access (where n = 32–$\log_2$(page size in bytes) and page size is specified by the SIZE field of the TLB entry). Software must set unused low-order RPN bits (that is, bits n:21) to 0. See *Address Translation* on page 227 and *Table 8-3* on page 228.
1	22:23	PAR1	**Parity for TLB word 1** (2 bits) These two bits are the parity read from the TLB array; they corresponds to the parity written into the array as dataPar1[0:1]. Set conditionally based on CCR0[CRPE]. See section *TPAR, PAR1 and PAR2 Parity Calculation* on page 243.
1	28:31	ERPN	**Extended Real Page Number** (4 bits) The 4-bit ERPN field are prepended to the rest of the translated address to form a total of a 36-bit real address. See *Address Translation* on page 227 and *Table 8-3* on page 228.
		Storage Attribute Fields	
2	0:1	PAR2	**Parity for TLB word 2** (2 bits) These two bits are the parity read from the TLB array; they corresponds to the parity written into the array as dataPar2[0:1]. Set conditionally based on CCR0[CRPE]. See section *TPAR, PAR1 and PAR2 Parity Calculation* on page 243.
2	2:9	_	**Reserved**

Table 8-1. TLB Entry Fields (continued)

TLB Word	Bit	Field	Description
2	10	FAR	**Fixed Address Region Attribute** (1 bit) Specifies the FAR storage attribute for the page associated with the TLB entry. The L2 cache array can be configured to use none, half, or all, of its memory array as a fixed-address, on-chip memory, attached as a slave device to the PLB5. Virtual addresses that map to the real addresses defined by the array configured as such in on-chip memory must be marked with the Fixed Address Region (FAR) attribute set to 1. See “L2 Cache Fixed Address Mode (FAM)” on page 177. **Note:** 1. FAR is not supported when CCR1[L2COBE] = 0 indicating that there is no L2 cache support. 2. Pages with FAR=1 can not also have M=1 (Memory Coherence Required).
2	11	WL1	**Write Through L1 Attribute** (1 bit) (See also the W attribute.) This attribute will be set to 1 by the hardware if the W attribute is set to 1. If the W bit is set to 0, this attribute may be set to 1 to force pages to be write-through in the L1 cache, but copy-back in the L2. 0 The page is not write-through in the L1 cache (i.e., the page is copy-back). 1 The page is write-through in the L1 cache.
2	12	IL1I	**Caching Inhibited L1 Instruction Cache Attribute** (1 bit) (See also the I attribute.) This attribute provides individual control over the L1 cacheability of instruction fetches to the page. Note that if the I attribute is set to 1, the hardware will set the IL1I attribute to 1 as well.
2	13	IL1D	**Caching Inhibited L1 Data Cache Attribute** (1 bit) (See also the I attribute.) This attribute provides individual control over the L1 cacheability of data accesses to the page. Note that if the I attribute is set to 1, the hardware will set the IL1D attribute to 1 as well.
2	14	IL2I	**Caching Inhibited L2 Instruction Cache Attribute** (1 bit) (See also the I attribute.) If L2 I-cache is implemented, this attribute provides individual control over the L2 cacheability of instruction fetches to the page. Note that if the I attribute is set to 1, the hardware will set the IL2I attribute to 1 as well.
2	15	IL2D	**Caching Inhibited L2 Data Cache Attribute** (1 bit) (See also the I attribute.) If L2 D-cache is implemented, this attribute provides individual control over the L2 cacheability of data accesses to the page. Note that if the I attribute is set to 1, the hardware will set the IL2D attribute to 1 as well.
2	16	U0	**User-Definable Storage Attribute 0** (1 bit) See *User-Definable (U0–U3)* on page 234. Specifies the U0 storage attribute for the page associated with the TLB entry. The function of this storage attribute is system-dependent, and has no effect within the PPC465.
2	17	U1	**User-Definable Storage Attribute 1** (1 bit) See *User-Definable (U0–U3)* on page 234. Specifies the U1 storage attribute for the page associated with the TLB entry. The function of this storage attribute is system-dependent, but the PPC465 can be programmed to use this attribute to designate a memory page as containing *transient* data and/or instructions (see *Level 1 Cache* on page 125). See the definition of MMUCR[U1TE].
2	18	U2	**User-Definable Storage Attribute 2** (1 bit) See *User-Definable (U0–U3)* on page 234. Specifies the U2 storage attribute for the page associated with the TLB entry. The function of this storage attribute is system-dependent, but the PPC465 core can be programmed to use this attribute to specify whether or not stores that miss in the L1 data cache should allocate the line in the L1 data cache. See the definition of MMUCR[U2SWOAE].
2	19	U3	**User-Definable Storage Attribute 3** (1 bit) See *User-Definable (U0–U3)* on page 234. Specifies the U3 storage attribute for the page associated with the TLB entry. The function of this storage attribute is system-dependent, but the PPC465 core can be programmed to use this attribute to specify whether or not stores that miss in the L2 data cache should allocate the line in the L2 data cache. See the definition of MMUCR[U3L2SWOAE].

Table 8-1. TLB Entry Fields (continued)

TLB Word	Bit	Field	Description	
2	20	W	**Write-Through** (1 bit) See *Write-Through (W)* on page 231. (See also the WL1 attribute).	
			0	The page is not write-through (that is, the page is copy-back) in the L2, and the L1 is controlled by the WL1 attribute.
			1	The page is write-through in both the L1 and L2 cache.
2	21	I	**Caching Inhibited** (1 bit) (See the IL1I, IL1D, IL2I, IL2D attributes).	
			0	The page is not caching inhibited (that is, the page may be cacheable). Cacheability in the L1 instruction and data caches, and the unified L2 cache, is also controlled by the IL1I, IL1D, IL2I, IL2D attributes.
			1	The page is caching inhibited in all cache levels.
2	22	M	**Memory Coherence Required** (1 bit) See *Memory Coherence Required (M)* on page 233.	
			0	The page is not memory coherence required.
			1	The page is memory coherence required.
			Note: Memory Coherency for the data is supported in a PPC465 system if the page attribute WL1=1 or IL1D=1, and M=1 with L2 cache in copy-back. L2 cache provides MESI support [that is write through OR not cacheable in the L1 cache, IL2D=0 (cacheable in the L2 cache), I=0 (architecturally cacheable), and M=1 (Memory Coherence Required)].	
2	23	G	**Guarded** (1 bit) See *Guarded (G)* on page 233.	
			0	The page is not guarded.
			1	The page is guarded.
2	24	E	**Endian** (1 bit) See *Endian (E)* on page 234.	
			0	All accesses to the page are performed with big-endian byte ordering, which means that the byte at the effective address is considered the most-significant byte of a multi-byte scalar (see *Byte Ordering* on page 44).
			1	All accesses to the page are performed with little-endian byte ordering, which means that the byte at the effective address is considered the least-significant byte of a multi-byte scalar (see *Byte Ordering* on page 44).
2	25	–	**Reserved**	
			Access Control Fields	
2	26	UX	**User State Execute Enable** (1 bit) See *Execute Access* on page 229.	
			0	Instruction fetch is not permitted from this page while MSR[PR]=1 and the attempt to execute an instruction from this page while MSR[PR] =1 will cause an Execute Access Control exception type Instruction Storage interrupt.
			1	Instruction fetch and execution is permitted from this page while MSR[PR]=1.
2	27	UW	**User State Write Enable** (1 bit) See *Write Access* on page 229.	
			0	Store operations and the **dcbz** instruction are not permitted to this page when MSR[PR]=1 and will cause a Write Access Control exception type Data Storage interrupt.
			1	Store operations and the **dcbz** instruction are permitted to this page when MSR[PR]=1.

Table 8-1. TLB Entry Fields (continued)

TLB Word	Bit	Field	Description	
2	28	UR	**User State Read Enable** (1 bit) See *Read Access* on page 229.	
			0	Load operations and the **dcbt**, **dcbtst**, **dcbst**, **dcbf**, **icbt**, and **icbi** instructions are not permitted from this page when MSR[PR]=1 and will cause a Read Access Control exception. Except for the **dcbt**, **dcbtst**, and **icbt** instructions, a Data Storage interrupt will occur (see *Table 8-4* on page 231).
			1	Load operations and the **dcbt**, **dcbtst**, **dcbst**, **dcbf**, **icbt**, and **icbi** instructions are permitted from this page when MSR[PR]=1.
2	29	SX	**Supervisor State Execute Enable** (1 bit) See *Execute Access* on page 229.	
			0	Instruction fetch is not permitted from this page while MSR[PR]=0 and the attempt to execute an instruction from this page while MSR[PR] =0 will cause an Execute Access Control exception type Instruction Storage interrupt.
			1	Instruction fetch and execution is permitted from this page while MSR[PR]=0.
2	30	SW	**Supervisor State Write Enable** (1 bit) See *Write Access* on page 229.	
			0	Store operations and the **dcbz** and **dcbi** instructions are not permitted to this page when MSR[PR]=0 and will cause a Write Access Control exception type Data Storage interrupt.
			1	Store operations and the **dcbz** and **dcbi** instructions are permitted to this page when MSR[PR]=0.
2	31	SR	**Supervisor State Read Enable** (1 bit) See *Read Access* on page 229.	
			0	Load operations and the **dcbt**, **dcbtst**, **dcbst**, **dcbf**, **icbt**, and **icbi** instructions are not permitted from this page when MSR[PR]=0 and will cause a Read Access Control exception. Except for the **dcbt**, **dcbtst**, and **icbt** instructions, a Data Storage interrupt will occur (see *Table 8-4* on page 231).
			1	Load operations and the **dcbt**, **dcbtst**, **dcbst**, **dcbf**, **icbt**, and **icbi** instructions are permitted from this page when MSR[PR]=0.

During reset, the ITLB and DTLB are loaded with an initial entry that maps the reset code and data page, and sets W=0, I=1, as in previous versions of the core. For PPC465, the new attributes are reset to: WL1=0, and IL1I=IL1D=IL2I=IL2D=1.

Note: Setting I to 1, automatically sets IL1I, IL1D, IL2I and IL2D to 1. Likewise, setting W to 1, automatically sets WL1 to 1.

8.3 Page Identification

The Valid (V), Effective Page Number (EPN), Translation Space Identifier (TS), Page Size (SIZE), and Translation ID (TID) fields of a particular TLB entry identify the page associated with that TLB entry. Except as noted, all comparisons must succeed to validate this entry for subsequent translation and access control processing. Failure to locate a matching TLB entry based on this criteria for instruction fetches will result in a TLB Miss exception type Instruction TLB Error interrupt. Failure to locate a matching TLB entry based on this criteria for data storage accesses will result in a TLB Miss exception which may result in a Data TLB Error interrupt, depending on the type of data storage access (certain cache management instructions do not result in an interrupt if they cause an exception; they simply no-op).

8.3.1 Virtual Address Formation

The first step in page identification is the expansion of the effective address into a virtual address. Again, the effective address is the 32-bit address calculated by a load, store, or cache management instruction, or as part of an instruction fetch. The virtual address is formed by prepending the effective address with a 1-bit address space identifier and an 8-bit process identifier. The process identifier is contained in the Process ID (PID) register. The address space identifier is provided by MSR[IS] for instruction fetches, and by MSR[DS] for data storage accesses and cache management operations, including instruction cache management operations. The resulting 41-bit value forms the virtual address, which is then compared to the virtual addresses contained in the TLB entries.

Note that the **tlbsx**[**.**] instruction also forms a virtual address, for software controlled search of the TLB. This instruction calculates the effective address in the same manner as a data access instruction, but the process identifier and address space identifier are provided by fields in the MMUCR, rather than by the PID and MSR, respectively (see *TLB Search Instruction (tlbsx[.])* on page 240).

8.3.2 Address Space Identifier Convention

The address space identifier differentiates between two distinct virtual address spaces, one generally associated with interrupt-handling and other system-level code and/or data, and the other generally associated with application-level code and/or data.

Typically, user mode programs will run with MSR[IS,DS] both set to 1, allowing access to application-level code and data memory pages. Then, on an interrupt, MSR[IS,DS] are both automatically cleared to 0, so that the interrupt handler code and data areas may be accessed using system-level TLB entries (that is, TLB entries with the TS field = 0). It is also possible that an operating system could set up certain system-level code and data areas (and corresponding TLB entries with the TS field = 1) in the application-level address space, allowing user mode programs running with MSR[IS,DS] set to 1 to access them (system library routines, for example, which may be shared by multiple user mode and/or supervisor mode programs). System-level code wishing to use these areas would have to first set the corresponding MSR[IS,DS] field in order to use the application-level TLB entries, or there would have to be alternative system-level TLB entries set up.

The net of this is that the notion of application-level code running with MSR[IS,DS] set to 1 and using corresponding TLB entries with the TS=1, and conversely system-level code running with MSR[IS,DS] set to 0 and using corresponding TLB entries with TS=0, is by convention. It is possible to run in user mode with MSR[IS,DS] set to 0, and conversely to run in supervisor mode with MSR[IS,DS] set to 1, with the corresponding TLB entries being used. The only fixed requirement in this regard is the fact that MSR[IS,DS] are cleared on an interrupt, and thus there *must* be a TLB entry for the system-level interrupt handler code with TS=0 in order to be able to fetch and execute the interrupt handler itself. Whether or not other system-level code switches MSR[IS,DS] and creates corresponding system-level TLB entries depends upon the operating system environment.

Programming Note: Software must ensure that there is always a valid TLB entry with TS=0 and with supervisor mode execute access permission (SX=1) corresponding to the effective address of the interrupt handlers. Otherwise, an Instruction TLB Error interrupt could result upon the fetch of the interrupt handler for some other interrupt type, and the registers holding the state of the routine which was executing at the time of the original interrupt (SRR0/SRR1) could be corrupted. See *Interrupts and Exceptions* on page 247 for more information.

8.3.3 TLB Match Process

This virtual address is used to locate the associated entry in the TLB. The address space identifier, the process identifier, and a portion of the effective address of the storage access are compared to the TS, TID, and EPN fields, respectively, of each TLB entry.

The virtual address matches a TLB entry if:

- The valid (V) field of the TLB entry is 1, and
- The value of the address space identifier is equal to the value of the TS field of the TLB entry, and
- Either the value of the process identifier is equal to the value of the TID field of the TLB entry (private page), or the value of the TID field is 0 (globally shared page), and
- The value of bits 0:n–1 of the effective address is equal to the value of bits 0:n-1 of the EPN field of the TLB entry (where n = 32–$\log_2$ (page size in bytes) and page size is specified by the value of the SIZE field of the TLB entry). See *Table 8-2 Page Size and Effective Address to EPN Comparison* on page 227.

A TLB Miss exception occurs if there is no matching entry in the TLB for the page specified by the virtual address (except for the **tlbsx**[**.**] instruction, which simply returns an undefined value to the GPR file and (for **tlbsx.**) sets CR[CR0]$_2$ to 0). See *TLB Search Instruction (tlbsx[.])* on page 240.

Programming Note: Although it is possible for software to create multiple TLB entries that match the same virtual address, doing so is a programming error and the results are undefined.

Figure 8-1 illustrates the criteria for a virtual address to match a specific TLB entry, while *Table 8-2* defines the page sizes associated with each SIZE field value, and the associated comparison (==) of the effective address to the EPN field.

Figure 8-1. Virtual Address to TLB Entry Match Process

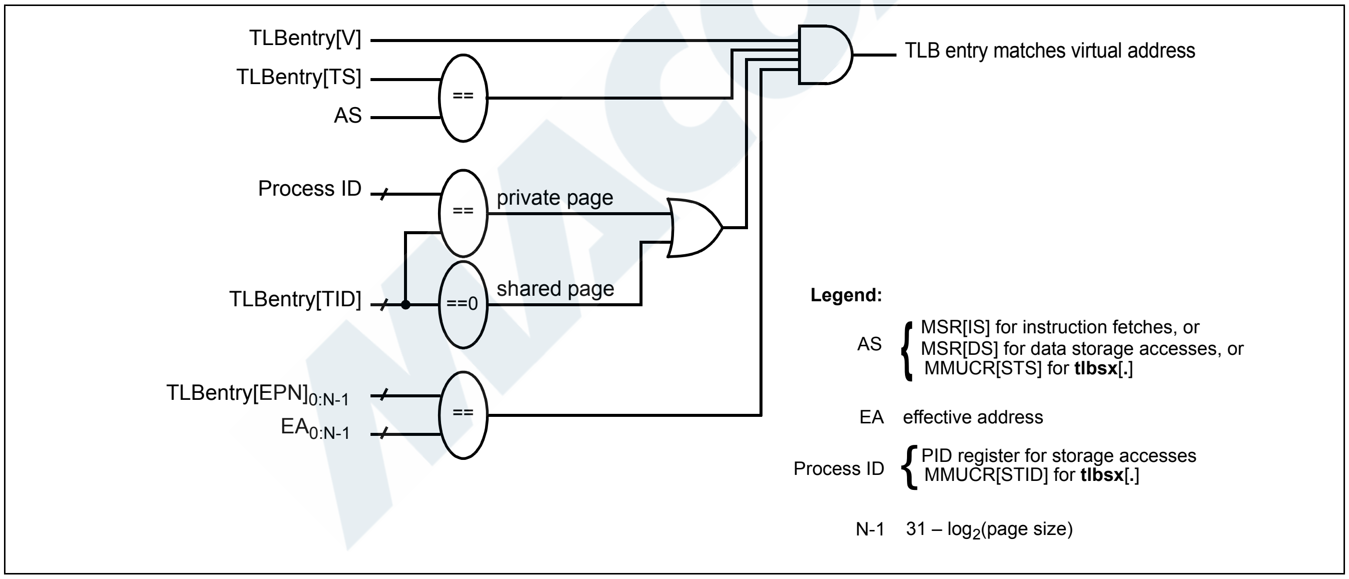


Table 8-2. Page Size and Effective Address to EPN Comparison

Size	Page Size	EA to EPN Comparison
0b0000	1KB	$EPN_{0:21}$ == $EA_{0:21}$
0b0001	4KB	$EPN_{0:19}$ == $EA_{0:19}$
0b0010	16KB	$EPN_{0:17}$ == $EA_{0:17}$
0b0011	64KB	$EPN_{0:15}$ == $EA_{0:15}$
0b0100	256KB	$EPN_{0:13}$ == $EA_{0:13}$
0b0101	1MB	$EPN_{0:11}$ == $EA_{0:11}$
0b0110	not supported	not supported
0b0111	16MB	$EPN_{0:7}$ == $EA_{0:7}$
0b1000	not supported	not supported
0b1001	256MB	$EPN_{0:3}$ == $EA_{0:3}$
0b1010	1GB	$EPN_{0:1}$ == $EA_{0:1}$
0b1011	not supported	not supported
0b1100	not supported	not supported
0b1101	not supported	not supported
0b1110	not supported	not supported
0b1111	not supported	not supported

8.4 Address Translation

Once a TLB entry is found which matches the virtual address associated with a given storage access, as described in “Page Identification” on page 224, the virtual address is translated to a real address according to the procedures described in this section.

The Real Page Number (RPN) and Extended Real Page Number (ERPN) fields of the matching TLB entry provide the page number portion of the real address. Let n=32–$\log_2$(*page size* in bytes) where *page size* is specified by the SIZE field of the matching TLB entry. Bits n:31 of the effective address (the “page offset”) are appended to bits 0:n–1 of the RPN field, and bits 0:3 of the ERPN field are prepended to this value to produce the 36-bit real address (that is, RA = $ERPN_{0:3}$ || $RPN_{0:n-1}$ || $EA_{n:31}$).

Figure 8-2 illustrates the address translation process, while *Table 8-3* defines the relationship between the different page sizes and the real address formation.

Figure 8-2. Effective-to-Real Address Translation Flow

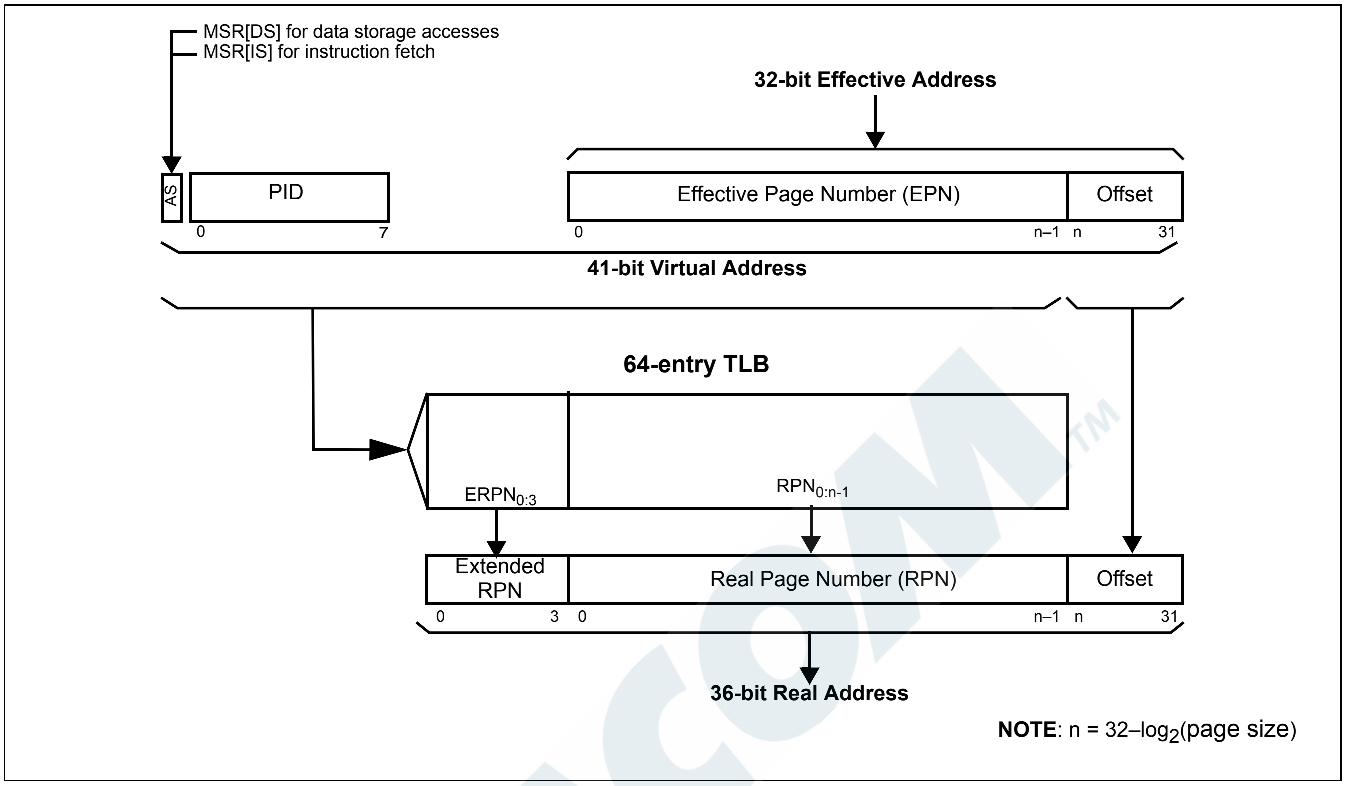


Table 8-3. Page Size and Real Address Formation

Size	Page Size	RPN bits required to be 0	Real Address
0b0000	1KB	none	$RPN_{0:21}$ || $EA_{22:31}$
0b0001	4KB	$RPN_{20:21}=0$	$RPN_{0:19}$ || $EA_{20:31}$
0b0010	16KB	$RPN_{18:21}=0$	$RPN_{0:17}$ || $EA_{18:31}$
0b0011	64KB	$RPN_{16:21}=0$	$RPN_{0:15}$ || $EA_{16:31}$
0b0100	256KB	$RPN_{14:21}=0$	$RPN_{0:13}$ || $EA_{14:31}$
0b0101	1MB	$RPN_{12:21}=0$	$RPN_{0:11}$ || $EA_{12:31}$
0b0110	not supported	not supported	not supported
0b0111	16MB	$RPN_{8:21}=0$	$RPN_{0:7}$ || $EA_{8:31}$
0b1000	not supported	not supported	not supported
0b1001	256MB	$RPN_{4:21}=0$	$RPN_{0:3}$ || $EA_{4:31}$
0b1010	1GB	$RPN_{2:21}=0$	$RPN_{0:1}$ || $EA_{2:31}$
0b1011	not supported	not supported	not supported
0b1100	not supported	not supported	not supported
0b1101	not supported	not supported	not supported
0b1110	not supported	not supported	not supported
0b1111	not supported	not supported	not supported

8.5 Access Control

Once a matching TLB entry has been identified and the address has been translated, the access control mechanism determines whether the program has execute, read, and/or write access to the page referenced by the address, as described in the following sections.

8.5.1 Execute Access

The UX or SX bit of a TLB entry controls *execute* access to a page of storage, depending on the operating mode, user (MSR[PR]=1) or supervisor (MSR[PR]=0), of the processor.

Instructions may be fetched and executed from a page in storage while in supervisor mode if the SX access control bit for that page is equal to 1. If the SX access control bit is equal to 0, then instructions from that page will not be fetched, and will not be placed into any cache as the result of a fetch request to that page while in supervisor mode.

Furthermore, if the sequential execution model calls for the execution in supervisor mode of an instruction from a page that is not enabled for execution in supervisor mode (that is, SX=0 when MSR[PR]=0), an Execute Access Control exception type Instruction Storage interrupt is taken (See “Interrupts and Exceptions” on page 247 for more information).

8.5.2 Write Access

The UW or SW bit of a TLB entry controls *write* access to a page, depending on the operating mode (user or supervisor) of the processor.

- User mode (MSR[PR] = 1)

  Store operations (including the store-class cache management instruction **dcbz**) are permitted to a page in storage while in user mode if the UW access control bit for that page is equal to 1. If execution of a store operation is attempted in user mode to a page for which the UW access control bit is 0, then a Write Access Control exception occurs. If the instruction is an **stswx** with string length 0, then no interrupt is taken and no operation is performed (see “Access Control Applied to Cache Management Instructions” on page 230). For all other store operations, execution of the instruction is suppressed and a Data Storage interrupt is taken.

  Note that although the **dcbi** cache management instruction is a store-class instruction, its execution is privileged and thus will not cause a Data Storage interrupt if execution of it is attempted in user mode (a Privileged Instruction exception type Program interrupt will occur instead).

- Supervisor mode (MSR[PR] = 0)

  Store operations (including the store-class cache management instructions **dcbz** and **dcbi**) are permitted to a page in storage while in supervisor mode if the SW access control bit for that page is equal to 1. If execution of a store operation is attempted in supervisor mode to a page for which the SW access control bit is 0, then a Write Access Control exception occurs. If the instruction is an **stswx** with string length 0, then no interrupt is taken and no operation is performed (see *Access Control Applied to Cache Management Instructions* on page 230). For all other store operations, execution of the instruction is suppressed and a Data Storage interrupt is taken.

8.5.3 Read Access

The UR or SR bit of a TLB entry controls *read* access to a page, depending on the operating mode (user or supervisor) of the processor.

- User mode (MSR[PR] = 1)

  Load operations (including the load-class cache management instructions **dcbst**, **dcbf**, **dcbt**, **dcbtst**, **icbi**, and **icbt**) are permitted from a page in storage while in user mode if the UR access control bit for that page is equal to 1. If execution of a load operation is attempted in user mode to a page for which the UR access control bit is 0, then a Read Access Control exception occurs. If the instruction is a load (not including **lswx** with string length 0) or is a **dcbst**, **dcbf**, or **icbi**, then execution of the instruction is suppressed and a Data Storage interrupt is taken. On the other hand, if the instruction is an **lswx** with string length 0, or is a **dcbt**, **dcbtst**, or **icbt**, then no interrupt is taken and no operation is performed (see *Access Control Applied to Cache Management Instructions* below).

- Supervisor mode (MSR[PR] = 0)

  Load operations (including the load-class cache management instructions **dcbst**, **dcbf**, **dcbt**, **dcbtst**, **icbi**, and **icbt**) are permitted from a page in storage while in supervisor mode if the SR access control bit for that page is equal to 1. If execution of a load operation is attempted in supervisor mode to a page for which the SR access control bit is 0, then a Read Access Control exception occurs. If the instruction is a load (not including **lswx** with string length 0) or is a **dcbst**, **dcbf**, or **icbi**, then execution of the instruction is suppressed and a Data Storage interrupt is taken. On the other hand, if the instruction is an **lswx** with string length 0, or is a **dcbt**, **dcbtst**, or **icbt**, then no interrupt is taken and no operation is performed (see *Access Control Applied to Cache Management Instructions* below).

8.5.4 Access Control Applied to Cache Management Instructions

This section summarizes how each of the cache management instructions is affected by the access control mechanism.

- **dcbz** instructions are treated as *stores* with respect to access control since they actually change the data in a cache block. As such, they can cause Write Access Control exception type Data Storage interrupts.
- **dcbi** instructions are treated as *stores* with respect to access control since they can change the value of a storage location by invalidating the “current” copy of the location in the data cache, effectively “restoring” the value of the location to the “former” value which is contained in memory. As such, they can cause Write Access Control exception type Data Storage interrupts.
- **dcba** instructions are treated as no-ops by the PPC465 under all circumstances, and thus can not cause any form of Data Storage interrupt.
- **icbi** instructions are treated as *loads* with respect to access control. As such, they can cause Read Access Control exception type Data Storage interrupts. Note that this instruction may cause a *Data* Storage interrupt (and not an *Instruction* Storage interrupt), even though it otherwise would perform its operation on the *instruction* cache. Instruction storage interrupts are associated with exceptions which occur upon the *fetch* of an instruction, whereas Data storage interrupts are associated with exceptions which occur upon the *execution* of a storage access or cache management instruction.
- **dcbt**, **dcbtst**, and **icbt** instructions are treated as *loads* with respect to access control. As such, they can cause Read Access Control exceptions. However, because these instructions are intended to act merely as “hints” that the specified cache block will likely be accessed by the processor in the near future, such exceptions will not result in a Data Storage interrupt. Instead, if a Read Access Control exception occurs, the instruction is treated as a no-op.
- **dcbf** and **dcbst** instructions are treated as *loads* with respect to access control. As such, they can cause Read Access Control exception type Data Storage interrupts. Flushing or storing a dirty line from the cache is not considered a store since an earlier store operation has already updated the cache line, and the **dcbf** or **dcbst** instruction is simply causing the results of that earlier store operation to be propagated to memory.
- **dccci** and **iccci** instructions do not even generate an address, nor are they affected by the access control mechanism. They are privileged instructions, and if executed in supervisor mode they will flash invalidate the entire associated cache.

Table 8-4 summarizes the effect of access control on each of the cache management instructions.

Table 8-4. Access Control Applied to Cache Management Instructions

Instruction	Read Protection Violation Exception	Write Protection Violation Exception
dcba	No	No
dcbf	Yes	No
dcbi	No	Yes
dcbst	Yes	No
dcbt	Yes[1]	No
dcbtst	Yes[1]	No
dcbz	No	Yes
dccci	No	No
icbi	Yes	No
icbt	Yes[1]	No
iccci	No	No
Note: dcbt, **dcbtst**, or **icbt** may cause a Read Access Control exception but will not result in a Data Storage interrupt		

8.6 Storage Attributes

Each TLB entry specifies a number of storage attributes for the memory page with which it is associated. Storage attributes affect the manner in which storage accesses to a given page are performed. The storage attributes (and their corresponding TLB entry fields) are:

- Write-through (W and WL1)
- Caching inhibited (I, IL1I, IL1D, IL2I and IL2D)
- Memory coherence required (M)
- Guarded (G)
- Endianness (E)
- User-definable (U0, U1, U2, U3)

All combinations of these attributes are supported except combinations which simultaneously specify a region as write-through and caching inhibited.

8.6.1 Write-Through (W)

If a memory page is marked as write-through (W=1), then the data for all store operations to that page are written to memory, as opposed to only being written into the data cache. If the referenced line also exists in the data cache (that is, the store operation is a "hit"), then the data will also be written into the data cache, although the cache line will *not* be marked as having been modified (that is, the "dirty" bit(s) will not be set).

8.6.1.1 Expanded Write-Through Storage Attributes, WL1

A write-through L1 storage attribute identified as WL1 has been added for independent write-through control of the L1 data cache. There are two main purposes for the attribute:

1. In order to recover from parity errors in the PPC465 L1 data cache, the L1 cache must operate in writethrough mode. Parity errors in the L1 data cache cause an exception if enabled. Software for handling the exception will typically invalidate the line in the L1 and return to the application, which will retry the access and miss the L1, but get the data from the ECC-protected memory.
2. PowerPC architecture prohibits aliasing of the W attribute - that is, multiple translations of virtual addresses that resolve to the same physical address must have the same value for all W attributes in all translation table entries. The same requirement is made of the new WL1 attribute. *Table 8-5* lists the combinations of the architected W bit and the implementation-specific WL1 bit:

Table 8-5. W and WL1 Attributes

W	WL1	**Description**
0	0	L1 data cache in copy-back mode.
0	1	L1 data cache is write-through.
1	0	Invalid combination
1	1	L1 data cache in write-through mode.

8.6.2 Caching Inhibited (I)

If a memory page is marked as caching inhibited (I=1), then all load, store, and instruction fetch operations perform their access in memory, as opposed to in the respective cache. If I=0, then the page is cacheable and the operations may be performed in the cache.

It is a programming error for the target location of a load, store, **dcbz**, or fetch access to caching inhibited storage to be in the respective cache; the results of such an access are undefined. It is *not* a programming error for the target locations of the other cache management instructions to be in the cache when the caching inhibited storage attribute is set. The behavior of these instructions is defined for both I=0 and I=1 storage. See the instruction descriptions in *Instruction Set* on page 343 for more information.

8.6.2.1 Expanded Cacheability Storage Attributes (IL1I, IL1D)

Additional cacheability bits have been added for independent control of L1: IL1I and IL1D respectively. When the architectural I bit (of WIMG) indicates that caching is inhibited, IL1I and IL1D must all also indicate that caching is inhibited (I = IL1I = IL1D = 1). When a memory reference must access the PLB, the I bit is presented as the cacheability attribute on the PLB bus. *Table 8-5* lists the combinations of the I and IL1 bits:

Table 8-6. I and IL1 Attributes

I	IL1*	PLB	L1 Caches
0	0	PLB requests are presented as Cacheable	L1 Cacheable
0	1		L1 Cache Inhibited
1	0	Invalid Combinations	
1	1	PLB Requests are presented as Cache Inhibited	L1 Cache Inhibited

Note: *There are two IL1 bits, for independent control of the instruction side accesses (IL1I) and data side accesses (IL1D). The IL1 column in the above table applies to both the IL1I and LI1D bits, as appropriate for the type of access.

The IL1I and IL1D bits were added to allow independent L1 cacheability of instructions and data residing on the same page. If there are instructions on the same page that are not really required to be memory-coherent (i.e. software ensures that the instructions on the page do not overlap with the coherent data), the instructions on the page can be cached by setting IL1I = 0, and the memory translation for both instructions and data can be handled by a single page table (and TLB) entry. If there were not independent IL1I and IL1D attributes, this functionality would require two virtual addresses that map to the same physical page with different L1 cacheability attributes, consuming two TLB entries.

Note: IL1D=1 and WL1=1 is not supported for the same page in the same manner that PowerPC Book E specifies I=1 and W=1 (non-cacheable and write-through) is not supported for the same page. (Either is a programming error.)

8.6.3 Memory Coherence Required (M)

The memory coherence required (M) storage attribute is defined by the architecture to support cache and memory coherency. If this bit is set (M=1) and the Write Thru Level 1 bit is also set (WL1=1), memory accesses to that page will be snooped by the PLB in all other caches in the system, maintaining memory coherence.

8.6.4 Guarded (G)

The guarded storage attribute is provided to control “speculative” access to “non-well-behaved” memory locations. Storage is said to be “well-behaved” if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. As such, data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

In general, storage that is not well-behaved should be marked as guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a Machine Check exception. For example, if the input buffer of a serial I/O device is memory-mapped, then an out-of-order or speculative access to that location could result in the loss of an item of data from the input buffer, if the instruction execution is interrupted and later re-attempted.

A data access to a guarded storage location is performed only if either the access is caused by an instruction that is known to be required by the sequential execution model, or the access is a load and the storage location is already in the data cache. Once a guarded data storage access is initiated, if the storage is also caching inhibited then only the bytes specifically requested are accessed in memory, according to the operand size for the instruction type. Data storage accesses to guarded storage which is marked as cacheable may access the entire cache block, either in the cache itself or in memory.

Instruction fetch is not affected by guarded storage. While the architecture does not prohibit instruction fetching from guarded storage, system software should generally prevent such instruction fetching by marking all guarded pages as “no-execute” (UX/SX = 0). Then, if an instruction fetch is attempted from such a page, the memory access will not occur and an Execute Access Control exception type Instruction Storage interrupt will result if and when execution is attempted for an instruction at any address within the page.

See “Level 1 Cache” on page 125 for more information on the handling of accesses to guarded storage. Also see “Level 2 Cache” on page 157 for information on the relationship between the guarded storage attribute and instruction restart and partially executed instructions.

8.6.5 Endian (E)

The endian (E) storage attribute controls the *byte ordering* with which load, store, and fetch operations are performed. Byte ordering refers to the order in which the individual bytes of a multiple-byte scalar operand are arranged in memory. The operands in a memory page with E=0 are arranged with *big-endian* byte ordering, which means that the bytes are arranged with the *most*-significant byte at the lowest-numbered memory address. The operands in a memory page with E=1 are arranged with *little-endian* byte ordering, which means that the bytes are arranged with the *least*-significant byte at the lowest-numbered address.

See *Byte Ordering* on page 44 for a more detailed explanation of big-endian and little-endian byte ordering.

8.6.6 User-Definable (U0–U3)

The PPC465 provides four user-definable (U0–U3) storage attributes which can be used to control system-dependent behavior of the storage system. By default, these storage attributes do not have any effect on the operation of the PPC465, although all storage accesses indicate to the memory subsystem the values of U0–U3 using the corresponding transfer attribute interface signals.

On the other hand, the PPC465 can be programmed to make specific use of two of the four user-definable storage attributes. Specifically, by enabling the function using a control bit in the MMUCR (see *Memory Management Unit Control Register (MMUCR)* on page 236), the U1 storage attribute can be used to designate whether storage accesses to the associated memory page should use the “normal” or “transient” region of the respective cache. Similarly, another control bit in the MMUCR can be set to enable the U2 storage attribute to be used to control whether or not store accesses to the associated memory page which miss in the data cache should allocate the line in the cache. The U1 or U2 storage attributes do not affect PPC465 operation unless they are enabled using the MMUCR to perform these specific functions. See “Level 1 Cache” on page 125 and “Level 2 Cache” on page 157 or more information on the mechanisms that can be controlled by the U1 and U2 storage attributes.

The U0 and U3 storage attributes have no such mechanism that enables them to control any specific function within the PPC465.

8.6.7 Supported Storage Attribute Combinations

Storage modes where both W = 1 and I = 1 (which would represent write-through but caching inhibited storage) are not supported. For all supported combinations of the W and I storage attributes, the G, E, and U0-U3 storage attributes may used in any combination.

8.6.8 Aliasing

For multiple pages that are mapped to the same real address the following rules apply:

2. If the multiple pages exist on a single processor, then:

   The I bits (I, IL1I, IL1D) must match the corresponding I bits on all pages (see note below).

   The W bits do not need to match on all pages.

   The WL1 bits must match on all pages.

   The M bits do not need to match on all pages; however, it will then be software's responsibility to maintain data coherency.

3. If the multiple pages exist on multiple processors, then:

   The I bits (I, IL1I, IL1D) do not need to match on all pages.

   The W bit must match on all pages. (Book E requirement).

   The WL1 bit does not need to match on all pages. (When W =0, WL1 can be 0 or 1, however, when W=1, WL1 must be 1.)

   The M bits do not need to match on all pages, however, it will then be software's responsibility to maintain data coherency.

Note: For multiple pages that exist on a single processor that map to the same real address, the I bits (I, IL1I, IL1D) do not need to match under the following conditions that must be guaranteed by software:

1. For those pages where the I bit is zero, the page must be marked as Guarded and no execute to prevent speculative accesses.

2. For those addresses where the cacheability attributes are different software must ensure that only those pages where all I bits are the same access the overlapped real address. (Or software could manage the cache appropriately between different cacheability accesses to guarantee that an access to any I = 1 is not found in the associated cache. When the architected I bit is a one, the data must not be in any level of cache.)

For example consider a cacheable 64K page and a non-cacheable 1K page that both map to the same real address (e.g. the 1K page maps to the last 1K of real addresses that the 64K page maps to). In this case the 64K page is marked as Guarded as well as Cacheable. In addition, software must ensure that when operating in the 64K page no accesses are performed to the last 1K addresses.

8.7 Storage Control Registers

In addition to the two registers described below, the MSR[IS,DS] bits specify which of the two address spaces the respective instruction or data storage accesses are directed towards. Also, the MSR[PR] bit is used by the access control mechanism. See *Machine State Register (MSR)* on page 253 for more detailed information on the MSR and the function of each of its bits.

8.7.1 Memory Management Unit Control Register (MMUCR)

The MMUCR is written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**. In addition, the MMUCR[STID] is updated with the TID field of the selected TLB entry when a **tlbre** instruction is executed. Conversely, the TID field of the selected TLB entry is updated with the value of the MMUCR[STID] field when a **tlbwe** instruction is executed. Other functions associated with the STID and other fields of the MMUCR are described in more detail in the sections that follow.

Figure 8-3. Memory Management Unit Control Register (MMUCR)

0:5		Reserved	
6	L2SWOA	L2 Store Without Allocate 0 Cacheable store misses allocates a line in the L2 cache. 1 Cacheable store misses do not allocate a line in the L2 cache.	If MMUCR[U3L2SWOAE] = 1, this bit is ignored.
7	SWOA	Store Without Allocate 0 Cacheable store misses allocate a line in the data cache. 1 Cacheable store misses do not allocate a line in the data cache.	If MMUCR[U2SWOAE] = 1, this field is ignored.
8		Reserved	
9	U1TE	U1 Transient Enable 0 Disable U1 storage attribute as transient storage attribute. 1 Enable U1 storage attribute as transient storage attribute.	
10	U2SWOAE	U2 Store without Allocate Enable 0 Disable U2 storage attribute control of store without allocate. 1 Enable U2 storage attribute control of store without allocate.	If MMUCR[U2SWOAE] = 1, the U2 storage attribute overrides MMUCR[SWOA].
11	U3L2SWOAE	U3 L2 Store without Allocate Enable 0 Disable U3 L2 storage attribute control of store without allocate. 1 Enable U3 L2 storage attribute control of store without allocate.	If MMUCR[U3L2SWOAE] = 0, it honors MMUCR[L2SWOA], and U3 attribute is ignored. If MMUCR[U3L2SWOAE] = 1, the U3 storage attribute overrides the value allowing programmer to set the L2SWOA value individually for each page.
12	DULXE	Data Cache Unlock Exception Enable 0 Data cache unlock exception is disabled. 1 Data cache unlock exception is enabled.	**dcbf** in user mode will cause Cache Locking exception type Data Storage interrupt when MMUCR[DULXE] is 1.
13	IULXE	Instruction Cache Unlock Exception Enable 0 Instruction cache unlock exception is disabled. 1 Instruction cache unlock exception is enabled.	**icbi** in user mode will cause Cache Locking exception type Data Storage interrupt when MMUCR[IULXE] is 1.
14		Reserved	
15	STS	Search Translation Space	Specifies the value of the translation space (TS) field for the **tlbsx[.]** instruction
16:23		Reserved	
24:31	STID	Search Translation ID	Specifies the value of the process identifier to be compared against the TLB entry's TID field for the **tlbsx[.]** instruction; also used to transfer a TLB entry's TID value for the **tlbre** and **tlbwe** instructions.

Store Without Allocate (SWOA) Field

Performance for certain applications can be affected by the allocation of cache lines on store misses. If the store accesses for a particular application are distributed sparsely in memory, and if the data is typically not re-used after having been stored, then performance may be improved by avoiding the latency and bus bandwidth associated with filling the entire cache line containing the bytes being stored. On the other hand, if an application typically stores to contiguous locations, or tends to store repeatedly to the same locations or to re-access data after it has been stored, then performance would likely be improved by allocating the line in the cache upon the first miss so that subsequent accesses will hit in the cache.

The SWOA field is one of two MMUCR fields which can control the allocation of cache lines upon store misses. The other is the U2SWOAE field, and if U2SWOAE is 1 then the U2 storage attribute controls the allocation and the SWOA field is ignored (see *User-Definable (U0–U3)* on page 234). However, if the U2SWOAE field is 0, then the SWOA field controls cache line allocation for all cacheable store misses. Specifically, if a cacheable store access misses in the data cache, then if SWOA is 0, then the cache line will be filled into the data cache, and the store data will be written into the cache (as well as to memory if the associated memory page is also marked as write-through; see *Write-Through (W)* on page 231). Conversely, if SWOA is 1, then cacheable store misses will *not* allocate the line in the data cache, and the store data will be written to memory only, whether or not the write-through attribute is set.

See “Level 1 Cache” on page 125 and “Level 2 Cache” on page 157 for more information on cache line allocation on store misses.

U1 Transient Enable (U1TE) Field

When U1TE is 1, then the U1 storage attribute is enabled to control the *transient* mechanism of the instruction and data caches (see “User-Definable (U0–U3)” on page 234). If the U1 field of the TLB entry for the memory page being accessed is 0, then the access will use the *normal* portion of the cache. If the U1 field is 1, then the transient portion of cache will be used.

If the U1TE field is 0, then the transient cache mechanism is disabled and all accesses use the normal portion of the cache.

See “Level 1 Cache” on page 125 and “Level 2 Cache” on page 157 for more information on the transient cache mechanism.

U2 Store Without Allocate Enable (U2SWOAE) Field

An explanation of the allocation of cache lines on store misses is provided in the section on the SWOA field above. The U2SWOAE field is the other mechanism which can control such allocation. If U2SWOAE is 0, then the SWOA field determines whether or not a cache line is allocated on a store miss.

When U2SWOAE is 1, then the U2 storage attribute is enabled to control the allocation on a memory page basis, and the SWOA field is ignored (see “User-Definable (U0–U3)” on page 234). If the U2 field of the TLB entry for the memory page containing the bytes being stored is 0, then the cache line is allocated in the data cache on a store miss. If the U2 field is 1, a store miss does not cause the cache line to be allocated.

See “Level 1 Cache” on page 125 and “Level 2 Cache” on page 157 for more information on cache line allocation on store misses.

Data Cache Unlock Exception Enable (DULXE) Field

The DULXE field can be used to force a Cache Locking exception type Data Storage interrupt to occur if a **dcbf** instruction is executed in user mode (MSR[PR]=1). Since **dcbf** can be executed in user mode and since it causes a cache line to be flushed from the data cache, it has the potential for allowing an application program to remove a locked line from the cache. The locking and unlocking of cache lines is generally a supervisor mode function, as the supervisor has access to the various mechanisms which control the cache locking mechanism (e.g., the Data Cache Victim Limit (DVLIM) and Instruction Cache Victim Limit (IVLIM) registers, and the MMUCR). Therefore, the DULXE field provides a means to prevent any **dcbf** instructions executed while in user mode from flushing any cache lines.

Note that with the PPC465, the Cache Locking exception occurs independent of whether the target line is truly locked or not. This behavior is necessary because the instruction execution pipeline is such that the exception determination must be made before it is determined whether or not the target line is actually locked (or whether it is even a hit).

Software at the Data Storage interrupt handler can determine whether the target line is locked, and if so whether or not the application should be allowed to unlock it.

If DULXE is 0, or if **dcbf** is executed while in supervisor mode, then the instruction execution is allowed to proceed and flush the target line, independent of whether it is locked or not.

See “Level 1 Cache” on page 125 and “Level 2 Cache” on page 157 for more information on cache locking.

Instruction Cache Unlock Exception Enable (IULXE) Field

The IULXE field can be used to force a Cache Locking exception type Data Storage interrupt to occur if an **icbi** instruction is executed in user mode (MSR[PR]=1). Since **icbi** can be executed in user mode and since it causes a cache line to be removed from the instruction cache, it has the potential for allowing an application program to remove a locked line from the cache. The locking and unlocking of cache lines is generally a supervisor mode function, as the supervisor has access to the various mechanisms which control the cache locking mechanism (e.g., the DVLIM and IVLIM registers, and the MMUCR). Therefore, the IULXE field provides a means to prevent any **icbi** instructions executed while in user mode from flushing any cache lines.

Note that with the PPC465, the Cache Locking exception occurs independent of whether the target line is truly locked or not. This behavior is necessary because the instruction execution pipeline is such that the exception determination must be made before it is determined whether or not the target line is actually locked (or whether it is even a hit).

Software at the Data Storage interrupt handler can determine whether the target line is locked, and if so whether or not the application should be allowed to unlock it.

If IULXE is 0, or if **icbi** is executed while in supervisor mode, then the instruction execution is allowed to proceed and flush the target line, independent of whether it is locked or not.

See “Level 1 Cache” on page 125 and “Level 2 Cache” on page 157 for more information on cache locking.

Search Translation Space (STS) Field

The STS field is used by the **tlbsx**[**.**] instruction to designate the value against which the TS field of the TLB entries is to be matched. For instruction fetch and data storage accesses, the TS field of the TLB entries is compared with the MSR[IS] bit or the MSR[DS] bit, respectively. For **tlbsx**[**.**] however, the MMUCR[STS] field is used, allowing the TLB to be searched for entries with a TS field which is references an address space other than the one being used by the currently executing process.

See “Address Space Identifier Convention” on page 225 for more information on the TLB entry TS field.

Search Translation ID (STID) Field

The STID field is used by the **tlbsx**[**.**] instruction to designate the process identifier value to be compared with the TID field of the TLB entries. For instruction fetch and data storage accesses and cache management operations, the TID field of the TLB entries is compared with the value in the PID register (see “Process ID (PID)” on page 239). For **tlbsx**[**.**] however, the MMUCR[STID] field is used, allowing the TLB to be searched for entries with a TID field which does not match the Process ID of the currently executing process.

The MMUCR[STID] field is also used to transfer the TLB entry’s TID field on **tlbre** and **tlbwe** instructions which target TLB word 0, as there are not enough bits in the GPR used for transferring the other fields such that it could hold this field as well.

See “TLB Match Process” on page 225 for more information on the TLB entry TID field and the address matching process. Also see “TLB Read/Write Instructions (tlbre/tlbwe)” on page 241 for more information on how the MMUCR[STID] field is used by these instructions.

8.7.2 Process ID (PID)

The Process ID (PID) is a 32-bit register, although only the lower 8 bits are defined in the PPC465. The 8-bit PID value is used as a portion of the virtual address for accessing storage (see “Virtual Address Formation” on page 225). The PID value is compared against the TID field of a TLB entry to determine whether or not the entry corresponds to a given virtual address. If an entry’s TID field is 0 (signifying that the entry defines a “global” as opposed to “private” page), then the PID value is ignored when determining whether the entry corresponds to a given virtual address. See “TLB Match Process” on page 225 for a more detailed description of the use of the PID value in the TLB match process.

The PID is written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 8-4. Process ID (PID)

0:23		Reserved	
24:31	PID	Process ID	

8.8 Shadow TLB Arrays

The PPC465 implements two shadow TLB arrays, one for instruction fetches and one for data accesses. These arrays “shadow” the value of a subset of the entries in the main, unified TLB (the UTLB in the context of this discussion). The purpose of the shadow TLB arrays is to reduce the latency of the address translation operation, and to avoid contention for the UTLB array between instruction fetches and data accesses.

The instruction shadow TLB (ITLB) contains four entries, while the data shadow TLB (DTLB) contains eight. There is no latency associated with accessing the shadow TLB arrays, and instruction execution continues in a pipelined fashion as long as the requested address is found in the shadow TLB. If the requested address is not found in the shadow TLB, the instruction fetch or data storage access is automatically stalled while the address is looked up in the UTLB. If the address is found in the UTLB, the penalty associated with the miss in the shadow array is three cycles. If the address is also a miss in the UTLB, then an Instruction or Data TLB Miss exception is reported.

The replacement of entries in the shadow TLB’s is managed by hardware, in a round-robin fashion. Upon a shadow TLB miss which leads to a UTLB hit, the hardware will automatically cast-out the oldest entry in the shadow TLB and replace it with the new translation.

The hardware will also automatically invalidate all of the entries in both of the shadow TLB's upon any context synchronization (see "Context Synchronization" on page 79). Context synchronizing operations include the following:

- Any interrupt (including Machine Check)
- Execution of **isync**
- Execution of **rfi, rfci,** or **rfmci**
- Execution of **sc**

Note that there are other "context changing" operations which do not cause automatic context synchronization in the hardware. For example, execution of a **tlbwe** instruction changes the UTLB contents but does not cause a context synchronization and thus does not invalidate or otherwise update the shadow TLB entries. In order for changes to the entries in the UTLB (or to other address-related resources such as the PID) to be reflected in the shadow TLB's, software must ensure that a context synchronizing operation occurs prior to any attempt to use any address associated with the updated UTLB entries (either the old or new contents of those entries). By invalidating the shadow TLB arrays, a context synchronizing operation forces the hardware to refresh the shadow TLB entries with the updated information in the UTLB as each memory page is accessed.

Note: Of the items in the preceding list of shadow TLB invalidating operations, the Machine Check interrupt is not architecturally required to be context synchronizing, and thus is not guaranteed to cause invalidation of any shadow TLB arrays on implementations other than those using the PPC465 processor. Consequently, software which is intended to be portable to other implementations should not depend on this behavior, and should insert the appropriate architecturally-defined context synchronizing operation as necessary for desired operation.

8.9 TLB Management Instructions

The processor does not imply any format for the page tables or the page table entries. Software has significant flexibility in organizing the size, location, and format of the page table, and in implementing a custom TLB entry replacement strategy. For example, software can "lock" TLB entries that correspond to frequently used storage, so that those entries are never cast out of the TLB, and TLB Miss exceptions to those pages never occur.

In order to enable software to manage the TLB, a set of TLB management instructions is implemented within the PPC465. These instructions are described briefly in the sections which follow, and in detail in *Instruction Set* on page 343 In addition, the interrupt mechanism provides resources to assist with software handling of TLB-related exceptions. One such resource is Save/Restore Register 0 (SRR0), which provides the exception-causing address for Instruction TLB Error and Instruction Storage interrupts. Another resource is the Data Exception Address Register (DEAR), which provides the exception-causing address for Data TLB Error and Data Storage interrupts. Finally, the Exception Syndrome Register (ESR) provides bits to differentiate amongst the various exception types which may cause a particular interrupt type. See *Interrupts and Exceptions* on page 247 for more information on these mechanisms.

All of the TLB management instructions are privileged, in order to prevent user mode programs from affecting the address translation and access control mechanisms.

8.9.1 TLB Search Instruction (tlbsx[.])

The **tlbsx[.]** instruction can be used to locate an entry in the TLB which is associated with a particular virtual address. This instruction forms an effective address for which the TLB is to be searched, in the same manner by which data storage access instructions perform their address calculation, by adding the contents of registers RA (or the value 0 if RA=0) and RB together. The MMUCR[STID] and MMUCR[STS] fields then provide the process ID and address space portions of the virtual address, respectively. Next, the TLB is searched for this virtual address, with the searching process including the notion of disabling the comparison to the process ID if the TID field of a given TLB entry is 0 (see "TLB Match Process" on page 225). Finally, the TLB index of the matching entry is written into the target register (RT). This index value can then serve as the source value for a subsequent **tlbre** or **tlbwe** instruction, to read or update the entry. If no matching entry is found, then the target register contents are undefined.

The “record form” of the instruction (**tlbsx.**) updates $CR[CR0]_2$ with the result of the search: if a match is found, then $CR[CR0]_2$ is set to 1; otherwise it is set to 0.

When the TLB is searched using a **tlbsx** instruction, if a matching entry is found, the parity calculated for the tag is compared to the parity stored in the TPAR field. A mismatch causes a parity error exception. Parity errors in words 1 and 2 of the entry will not cause parity error exceptions when executing a **tlbsx** instruction.

8.9.2 TLB Read/Write Instructions (tlbre/tlbwe)

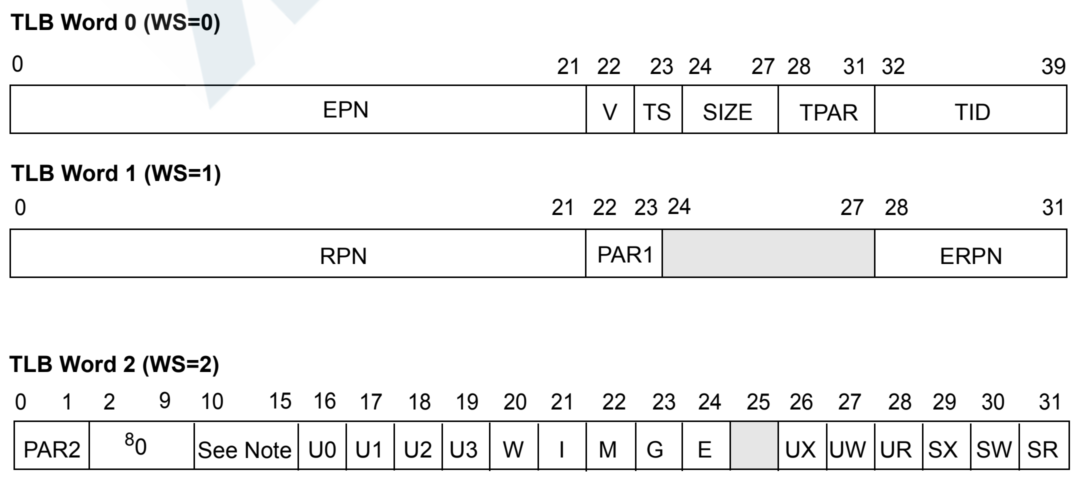
TLB entries can be read and written by the **tlbre** and **tlbwe** instructions, respectively. Since a TLB entry contains more than 32 bits, multiple **tlbre**/**tlbwe** instructions must be executed in order to transfer all of the TLB entry information. A TLB entry is divided into three portions, TLB word 0, TLB word 1, and TLB word 2. The RA field of the **tlbre** and **tlbwe** instructions designates a GPR from which the low-order six bits are used to specify the TLB index of the TLB entry to be read or written. An immediate field (WS) designates which word of the TLB entry is to be transferred (that is, WS=0 specifies TLB word 0, and so on). Finally, the contents of the selected TLB word are transferred to or from a designated target or source GPR (and the MMUCR[STID] field, for TLB word 0; see below), respectively.

The fields in each TLB word are illustrated in *Figure 8-5*. The bit numbers indicate which bits of the target/source GPR correspond to each TLB field. Note that the TID field of TLB word 0 is transferred to/from the MMUCR[STID] field, rather than to/from the target/source GPR.

When executing a **tlbre**, the parity fields (TPAR, PAR1, and PAR2) are loaded if and only if the CCR0[CRPE] bit is set. Otherwise those fields are loaded with zeros. When the **tlbre** is executed, If the parity bits stored for the particular word that is read by the **tlbre** indicate a parity error, the parity error exception *will* be generated regardless of the state of the CCR0[CRPE] bit.

When executing a **tlbwe**, bits in the source GPR that correspond to the parity fields are ignored, as the hardware calculates the parity to be recorded in those fields of the entry.

Figure 8-5. TLB Entry Word Definitions



Note:
Bit 10 - FAR

Bit 11 - WL1
Bit 13 - IL1D
Bit 14 - IL2I
Bit 15 - IL2D

8.9.3 TLB Sync Instruction (tlbsync)

The **tlbsync** instruction is used to synchronize software TLB management operations in a multiprocessor environment with hardware-enforced coherency, which is not supported by the PPC465. Consequently, this instruction is treated as a no-op. It is provided in support of software compatibility between PowerPC-based systems.

8.10 Page Reference and Change Status Management

When performing page management, it is useful to know whether a given memory page has been referenced, and whether its contents have been modified. Note that this may be more involved than determining whether a given TLB entry has been used to reference or change memory, since multiple TLB entries may translate to the same memory page. If it is necessary to replace the contents of some memory page with other contents, a page which has been referenced (accessed for any purpose) is more likely to be maintained than a page which has never been referenced. If the contents of a given memory page are to be replaced and the contents of that page have been changed, then the current contents of that page must be written to backup physical storage (such as a hard disk) before replacement.

Similarly, when performing TLB management, it is useful to know whether a given TLB entry has been referenced. When making a decision about which entry of the TLB to replace in order to make room for a new entry, an entry which has never been referenced is a more likely candidate to be replaced.

The PPC465 does not automatically record references or changes to a page or TLB entry. Instead, the interrupt mechanism may be used by system software to maintain reference and change information for TLB entries and their associated pages, respectively.

Execute, Read and Write Access Control exceptions may be used to allow software to maintain reference and change information for a TLB entry and for its associated memory page. The following description explains one way in which system software can maintain such reference and change information.

The TLB entry is originally written into the TLB with its access control bits (UX, SX, UR, SR, UW, and SW) off. The first attempt of application code to use the page will therefore cause an Access Control exception and a corresponding Instruction or Data Storage interrupt. The interrupt handler records the reference to the TLB entry and to the associated memory page in a software table, and then turns on the appropriate access control bit, thereby indicating that the particular TLB entry has been referenced. An initial read from the page is handled by only turning on the appropriate UR or SR access control bit, leaving the page “read-only”. Subsequent read accesses to the page via that TLB entry will proceed normally.

If a write access is later attempted, a Write Access Control exception type Data Storage interrupt will occur. The interrupt handler records the change status to the memory page in a software table, and then turns on the appropriate UW or SW access control bit, thereby indicating that the memory page associated with the particular TLB entry has been changed. Subsequent write accesses to the page via that TLB entry will proceed normally.

8.11 TLB Parity Operations

The TLB is parity protected against soft errors in the TLB memory array that are caused by alpha particle impacts. If such errors are detected, the CPU can be configured to vector to the machine check interrupt handler, which can restore the corrupted state of the TLB from the page tables in system memory.

The TLB is a 64-entry CAM/RAM with 40 tag bits, 41 data bits, and 8 parity bits per entry. Tag and data bits are parity protected with four parity bits for the 40-bit tag, two parity bits for 26 bits of data (i.e. those read and written as word 1 by the **tlbre** and **tlbwe** instructions), and two more parity bits for the remaining 15 bits of data (i.e. word 2). The parity bits are stored in the TLB entries in fields named TPAR, PAR1, and PAR2, respectively. See *Figure 8-5 TLB Entry Word Definitions*.

Unlike the instruction and data cache CAM/RAMs, the TLB does *not* detect multiple hits due to parity errors in the tags. The TLB is a relatively small memory array, and the reduction in Soft Error Rate (SER) provided by adding multi-hit detection to the circuit is small, and so, not worth the expense of the feature.

TLB parity bits are *set* any time the TLB is updated, which is always done via a **tlbwe** instruction. TLB parity is *checked* each time the TLB is searched or read, whether to re-fill the ITLB or DTLB, or as a result of a **tlbsx** or **tlbre** instruction. When executing an ITLB or DTLB refill, parity is checked for the tag and both data words. When executing a **tlbsx**, data output is not enabled for the translation and protection outputs of the TLB, so only the tag parity is checked. When executing a **tlbre**, parity is checked only for the word specified in the WS field of the **tlbre** instruction. Detection of a parity error causes a machine check exception. If MSR[ME] is set (which is the usual case), the processor takes a machine check interrupt.

8.11.1 TPAR, PAR1 and PAR2 Parity Calculation

Figure 8-7 through *Figure 8-10* indicate how parity is calculated for TLB Word 0, 1 and 2.

Table 8-7. DSIZ, EPN and RPN Bits Included in TPAR and PAR1 Parity Calculation

Page Size	Size	DSIZ	Even_EPN/Even_RPN	Odd_EPN/Odd_RPN
1KB	0b0000	0b00000000	$EPN_{1,3,5,7,9,11,13,15,17,19,21}$ $RPN_{1,3,5,7,9,11,13,15,17,19,21}$	$EPN_{0,2,4,6,8,10,12,14,16,18,20}$ $RPN_{0,2,4,6,8,10,12,14,16,18,20}$
4KB	0b0001	0b00000001	$EPN_{1,3,5,7,9,11,13,15,17,19}$ $RPN_{1,3,5,7,9,11,13,15,17,19}$	$EPN_{0,2,4,6,8,10,12,14,16,18}$ $RPN_{0,2,4,6,8,10,12,14,16,18}$
16KB	0b0010	0b00000011	$EPN_{1,3,5,7,9,11,13,15,17}$ $RPN_{1,3,5,7,9,11,13,15,17}$	$EPN_{0,2,4,6,8,10,12,14,16}$ $RPN_{0,2,4,6,8,10,12,14,16}$
64KB	0b0011	0b00000111	$EPN_{1,3,5,7,9,11,13,15}$ $RPN_{1,3,5,7,9,11,13,15}$	$EPN_{0,2,4,6,8,10,12,14}$ $RPN_{0,2,4,6,8,10,12,14}$
256KB	0b0100	0b00001111	$EPN_{1,3,5,7,9,11,13}$ $RPN_{1,3,5,7,9,11,13}$	$EPN_{0,2,4,6,8,10,12}$ $RPN_{0,2,4,6,8,10,12}$
1MB	0b0101	0b00011111	$EPN_{1,3,5,7,9,11}$ $RPN_{1,3,5,7,9,11}$	$EPN_{0,2,4,6,8,10}$ $RPN_{0,2,4,6,8,10}$
16MB	0b0111	0b00111111	$EPN_{1,3,5,7}$ $RPN_{1,3,5,7}$	$EPN_{0,2,4,6}$ $RPN_{0,2,4,6}$
256MB	0b1001	0b01111111	$EPN_{1,3}$ $RPN_{1,3}$	$EPN_{0,2}$ $RPN_{0,2}$
1GB	0b1010	0b11111111	EPN_1 RPN_1	EPN_0 RPN_0

Note: DSIZ is the decoded version of the Size field as seen internally by the MMU. Parity is calculated on DSIZ not the encoded Size setting.

Table 8-8. TPAR Calculation

TLB Word 0	TPAR[0:3]	Parity Calculation for TLB Word 0
bit 28	TPAR[0]	bitwise XOR(Even_EPN || TS || Even_TID)
bit 29	TPAR[1]	bitwise XOR(Odd_EPN || V || Odd_TID)
bit 30	TPAR[2]	bitwise XOR(Odd_DSIZ)
bit 31	TPAR[3]	bitwise XOR(Even_DSIZ || TD)

Formula Key for *Table 8-8 TPAR Calculation*

1. Even_EPN and Odd_EPN are listed by page size in *Table 8-8 DSIZ, EPN and RPN Bits Included in TPAR and PAR1 Parity Calculation.*
2. Even_TID is TLB Word 0 bits 32, 34, 36, 38. Only included in parity calculation when TD = 0.
3. Odd_TID is TLB Word 0 bits 33, 35, 37, 39. Only include in parity calculation when TD = 0.
4. TD = 0 if TID > 0 and TD = 1 if TID = 0.
5. Even_DSIZ is DSIZ bits 0, 2, 4, 6. See *Table 8-8 DSIZ, EPN and RPN Bits Included in TPAR and PAR1 Parity Calculation.*
6. Odd_DSIZ is DSIZ bits 1, 3, 5, 7. *Table 8-8 DSIZ, EPN and RPN Bits Included in TPAR and PAR1 Parity Calculation.*

Table 8-9. PAR1 Calculation

TLB Word 1	PAR1[0:1]	Parity Calculation for TLB Word 1
bit 0	PAR1[0]	bitwise XOR(Even_RPN || Even_ERPN)
bit 1	PAR1[1]	bitwise XOR(Odd_RPN || Odd_ERPN)

Formula Key for *Table 8-9 PAR1 Calculation*

1. Even_RPN and Odd_RPN are listed by page size in *Table 8-9 DSIZ, EPN and RPN Bits Included in TPAR and PAR1 Parity Calculation.*
2. Even_ERPN is TLB Word 1 bits 28 and 30.
3. Odd_ERPN is TLB Word 1 bits 29 and 31.

Table 8-10. PAR2 Calculation

TLB Word 2	PAR2[0:1]	Parity Calculation for TLB Word 2
bit 0	PAR2[0]	bitwise XOR(FAR, IL1I, IL2I, U0, U2, W, M, E, UX, UR, SW)
bit 1	PAR2[1]	bitwise XOR(WL1, IL1D, IL2D, U1, U3, I, G, UW, SX, SR)

8.11.2 Reading TLB Parity Bits with tlbre

If CCR0[CRPE] is set, execution of a **tlbre** instruction updates the target register with parity values as well as the tag or other data from the TLB. However, since a **tlbre** that detects a parity error will cause a machine check exception, the target register can only be updated with a “bad” parity value if the MSR[ME] bit is cleared, preventing the machine check interrupt. Thus the usual flow of code that detects a parity error in the TLB and then finds out which entry is erroneous would proceed as:

1. A **tlbre** instruction is executed from normal OS code, resulting in a parity exception. The exception sets MCSR[TLBE] and MCSR[MCS].
2. MSR[ME] = 1, so the CPU vectors to the machine check handler (i.e., takes the machine check interrupt) and resets the MSR[ME] bit. Note that even though the parity error causes an *asynchronous* interrupt, that interrupt is guaranteed to be taken before the **tlbre** instruction completes if the CCR0[PRE] (Parity Recovery Enable) is set, and so the target register (RT) of the **tlbre** will not be updated.

3. The Machine Check handler code includes a series of **tlbre** instructions to query the state of the TLB and find the erroneous entry. When a **tlbre** encounters an erroneous entry and MSR[ME] = 0, the parity exception still happens, setting the MCSR[MCS] and MCSR[TLBE] bits. Additionally, since MSR[ME] = 0, MCSR[IMCE] is set, indicating that an imprecise machine check was detected. Finally, the instruction completes, (since no interrupt is taken because MSR[ME] = 0), updating the target register with data from the TLB, including the parity information.

Note: The **tlbre** causes an exception when it detects a parity error, but the **icread** and **dcread** instructions do not. This inconsistency is explained because OS code commonly uses a sequence of **tlbsx** and **tlbre** instructions to update the “changed” bit in the page table entries (see *Page Reference and Change Status Management* on page 242). Forcing the software to check the parity manually for each **tlbre** would be a performance limitation. No such functional use exists for the **icread** and **dcread** instructions; they are used only in debugging contexts with no significant performance requirements.

As is the case for any machine check interrupt, after vectoring to the machine check handler, the MCSRR0 contains the value of the oldest “uncommitted” instruction in the pipeline at the time of the exception and MCSRR1 contains the old (MSR) context. The interrupt handler is able to query Machine Check Status Register (MCSR) to find out that it was called due to a TLB parity exception, and then use **tlbre** instructions to find the error in the TLB and restore it from a known good copy in main memory.

Note: A parity error on the TLB entry *which maps the machine check exception handler code* prevents recovery. In effect, one of the 64 TLB entries is unprotected, in that the machine cannot recover from an error in that entry. It is possible to add logic to get around this problem, but the reduction in SER achieved by protecting 63 out of 64 TLB entries is sufficient. Further, the software technique of simply dedicating a TLB entry to the page that contains the machine check handler and periodically refreshing that entry from a known good copy can reduce the probability that the entry will be used with a parity error to near zero.

As mentioned above, any **tlbre** or **tlbsx** instruction that causes a machine check interrupt will be flushed from the pipeline before it completes. Further, any instruction that causes a DTLB or ITLB refill which causes a TLB parity error will be flushed before it completes.

8.11.3 Simulating TLB Parity Errors for Software Testing

Because parity errors occur in the TLB infrequently and unpredictably, it is desirable to provide users with a way to simulate the effect of a TLB parity error so that interrupt handling software may be exercised. This is exactly the purpose of the 4-bit CCR1[MMUPEI] field.

Usually, parity is calculated as the even parity for each set of bits to be protected, which the checking hardware expects. This calculation is done as the TLB data is stored with a **tlbwe** instruction. However, if any of the CCR1[MMUPEI] bits are set, the calculated parity for the corresponding bits of the data being stored are inverted and stored as odd parity. Then, when the data stored with odd parity is subsequently used to refill the DTLB or ITLB, or by a **tlbsx** or **tlbre** instruction, it will cause a Parity exception type Machine Check interrupt and exercise the interrupt handling software. The following pseudo-code is an example of how to use the CCR1[MMUPEI] field to simulate a parity error on a TLB entry:

```
mtspr CCR1, Rx          ; Set some CCR1[MMUPEI] bits
isync                   ; wait for the CCR1 context to update
tlbwe Rs,Ra,0           ; write some data to the TLB with bad parity
tlbwe Rs,Ra,1           ; write some data to the TLB with bad parity
tlbwe Rs,Ra,2           ; write some data to the TLB with bad parity
isync                   ; wait for the tlbwe(s) to finish
mtspr CCR1, Rz          ; Reset CCR1[MMUPEI]
isync                   ; wait for the CCR1 context to update
tlbre RT,RA,WS          ; tlbre with bad parity causes interrupt
```

9. Interrupts and Exceptions

This chapter begins by defining the terminology and classification of interrupts and exceptions in *Overview* and *Interrupt Classes*.

Interrupt Processing on page 250 explains in general how interrupts are processed, including the requirements for partial execution of instructions.

Several registers support interrupt handling and control. *Interrupt Processing Registers* on page 253 describes these registers.

Table 9-2 Interrupt and Exception Types on page 261 lists the interrupts and exceptions handled by the PPC465, in the order of Interrupt Vector Offset Register (IVOR) usage. Detailed descriptions of each interrupt type follow, in the same order.

Finally, *Interrupt Ordering and Masking* on page 281 and *Exception Priorities* on page 284 define the priority order for the processing of simultaneous interrupts and exceptions.

9.1 Overview

An *interrupt* is the action in which the processor saves its old context (Machine State Register (MSR) and next instruction address) and begins execution at a pre-determined interrupt-handler address, with a modified MSR. *Exceptions* are the events that may cause the processor to take an interrupt, if the corresponding interrupt type is enabled.

Exceptions may be generated by the execution of instructions, or by signals from devices external to the PPC465, the internal timer facilities, debug events, or error conditions.

9.2 Interrupt Classes

All interrupts, except for Machine Check, can be categorized according to two independent characteristics of the interrupt:

- Asynchronous or synchronous
- Critical or non-critical

9.2.1 Asynchronous Interrupts

Asynchronous interrupts are caused by events that are independent of instruction execution. For asynchronous interrupts, the address reported to the interrupt handling routine is the address of the instruction that would have executed next, had the asynchronous interrupt not occurred.

9.2.2 Synchronous Interrupts

Synchronous interrupts are those that are caused directly by the execution (or attempted execution) of instructions, and are further divided into two classes, *precise* and *imprecise*.

Synchronous, precise interrupts are those that *precisely* indicate the address of the instruction causing the exception that generated the interrupt; or, for certain synchronous, precise interrupt types, the address of the immediately following instruction.

Synchronous, imprecise interrupts are those that may indicate the address of the instruction which caused the exception that generated the interrupt, or the address of some instruction after the one which caused the exception.

9.2.2.1 Synchronous, Precise Interrupts

When the execution or attempted execution of an instruction causes a synchronous, precise interrupt, the following conditions exist when the associated interrupt handler begins execution:

- SRR0 (see *Save/Restore Register 0 (SRR0)* on page 254) or CSRR0 (see *Critical Save/Restore Register 0 (CSRR0)* on page 255) addresses either the instruction which caused the exception that generated the interrupt, or the instruction immediately following this instruction. Which instruction is addressed can be determined from a combination of the interrupt type and the setting of certain fields of the ESR (see *Exception Syndrome Register (ESR)* on page 258).
- The interrupt is generated such that all instructions preceding the instruction which caused the exception appear to have completed with respect to the executing processor. However, some storage accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms.
- The instruction which caused the exception may appear not to have begun execution (except for having caused the exception), may have been partially executed, or may have completed, depending on the interrupt type (see *Partially Executed Instructions* on page 251).
- Architecturally, no instruction beyond the one which caused the exception has executed.

9.2.2.2 Synchronous, Imprecise Interrupts

When the execution or attempted execution of an instruction causes a synchronous, imprecise interrupt, the following conditions exist when the associated interrupt handler begins execution:

- SRR0 or CSRR0 addresses either the instruction which caused the exception that generated the interrupt, or some instruction following this instruction.
- The interrupt is generated such that all instructions preceding the instruction addressed by SRR0 or CSRR0 appear to have completed with respect to the executing processor.
- If the imprecise interrupt is forced by the context synchronizing mechanism, due to an instruction that causes another exception that generates an interrupt (for example, Alignment, Data Storage), then SRR0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction may have been partially executed (see *Partially Executed Instructions* on page 251).
- If the imprecise interrupt is forced by the execution synchronizing mechanism, due to executing an execution synchronizing instruction other than **msync** or **isync**, then SRR0 or CSRR0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction appears not to have begun execution (except for its forcing the imprecise interrupt). If the imprecise interrupt is forced by an **msync** or **isync** instruction, then SRR0 or CSRR0 may address either the **msync** or **isync** instruction, or the following instruction.
- If the imprecise interrupt is not forced by either the context synchronizing mechanism or the execution synchronizing mechanism, then the instruction addressed by SRR0 or CSRR0 may have been partially executed (see *Partially Executed Instructions* on page 251).
- No instruction following the instruction addressed by SRR0 or CSRR0 has executed.

The only synchronous, imprecise interrupts in the PPC465 are the “special cases” of “delayed” interrupts, which can result when certain kinds of exceptions occur while the corresponding interrupt type is disabled. The first of these is the Floating-Point Enabled exception type Program interrupt. For this type of interrupt to occur, a floating-point unit must be attached to the auxiliary processor interface of the PPC465, and the Floating-point Enabled Exception Summary bit of the Floating-Point Status and Control Register (FPSCR[FEX]) must be set while

Floating-point Enabled exception type Program interrupts are disabled due to MSR[FE0,FE1] both being 0. If and when such interrupts are subsequently enabled, by setting one or the other (or both) of MSR[FE0,FE1] to 1 while FPSCR[FEX] is still 1, then a synchronous, imprecise form of Floating-Point Enabled exception type Program interrupt will occur, and SRR0 will be set to the address of the instruction which would have executed next (that is, the instruction after the one which updated MSR[FE0,FE1]). If the MSR was updated by an **rfi**, **rfci,** or **rfmci** instruction, then SRR0 will be set to the address to which the **rfi**, **rfci,** or **rfmci** was returning, and not to the instruction address which is sequentially after the **rfi**, **rfci,** or **rfmci**.

The second type of delayed interrupt which may be handled as a synchronous, imprecise interrupt is the Debug interrupt. Similar to the Floating-Point Enabled exception type Program interrupt, the Debug interrupt can be temporarily disabled by an MSR bit, MSR[DE]. Accordingly, certain kinds of Debug exceptions may occur and be recorded in the DBSR while MSR[DE] is 0, and later lead to a delayed Debug interrupt if MSR[DE] is set to 1 while a Debug exception is still set in the DBSR. If and when this occurs, the interrupt will either be synchronous and imprecise, or it will be asynchronous, depending on the type of Debug exception causing the interrupt. In either case, CSRR0 is set to the address of the instruction which would have executed next (that is, the instruction after the one which set MSR[DE] to 1). If MSR[DE] is set to 1 by **rfi**, **rfci,** or **rfmci**, then CSRR0 is set to the address to which the **rfi**, **rfci,** or **rfmci** was returning, and not to the address of the instruction which was sequentially after the **rfi**, **rfci,** or **rfmci**.

Besides these special cases of Program and Debug interrupts, all other synchronous interrupts are handled precisely by the PPC465, including FP Enabled exception type Program interrupts even when the processor is operating in one of the architecturally-defined imprecise modes (MSR[FE0,FE1] = 0b01 or 0b10).

See *Program Interrupt* on page 270 and *Debug Interrupt* on page 278 for a more detailed description of these interrupt types, including both the precise and imprecise cases.

9.2.3 Critical and Non-Critical Interrupts

Interrupts can also be classified as critical or noncritical interrupts. Certain interrupt types demand immediate attention, even if other interrupt types are currently being processed and have not yet had the opportunity to save the state of the machine (that is, return address and captured state of the MSR). To enable taking a critical interrupt immediately after a non-critical interrupt has occurred (that is, before the state of the machine has been saved), two sets of Save/Restore Register pairs are provided. Critical interrupts use the Save/Restore Register pair CSRR0/CSRR1. Non-Critical interrupts use Save/Restore Register pair SRR0/SRR1.

9.2.4 Machine Check Interrupts

Machine Check interrupts are a special case. They are typically caused by some kind of hardware or storage subsystem failure, including L2 Cache /subsystem, or by an attempt to access an invalid address. A Machine Check may be caused indirectly by the execution of an instruction, but not be recognized and/or reported until after the processor has executed past the instruction that caused the Machine Check. As such, Machine Check interrupts cannot be classified as either synchronous or asynchronous, nor as precise or imprecise. They also do not belong to either the critical or the non-critical interrupt class, but instead have associated with them a unique pair of save/restore registers, Machine Check Save/Restore Registers 0/1 (MCSRR0/1).

Architecturally, the following general rules apply for Machine Check interrupts:

1. No instruction after the one whose address is reported to the Machine Check interrupt handler in MCSRR0 has begun execution.
2. The instruction whose address is reported to the Machine Check interrupt handler in MCSRR0, and all prior instructions, may or may not have completed successfully. All those instructions that are ever going to complete appear to have done so already, and have done so within the context existing prior to the Machine Check interrupt. No further interrupt (other than possible additional Machine Check interrupts) will occur as a result of those instructions.

With the PPC465, Machine Check interrupts can be caused by Machine Check exceptions on a memory access for an instruction fetch, for a data access, or for a TLB access. Some of the interrupts generated behave as synchronous, precise interrupts, while other are handled in an asynchronous fashion.

In the case of an Instruction Synchronous Machine Check exception, the PPC465 will handle the interrupt as a synchronous, precise interrupt, assuming Machine Check interrupts are enabled (MSR[ME] = 1). That is, if a Machine Check exception is detected during an instruction fetch, the exception will not be *reported* to the interrupt mechanism unless and until execution is attempted for the instruction address at which the Machine Check exception occurred. If, for example, the direction of the instruction stream is changed (perhaps due to a branch instruction), such that the instruction at the address associated with the Machine Check exception will not be executed, then the exception will not be reported and no interrupt will occur. If and when an Instruction Machine Check exception is reported, and if Machine Check interrupts are enabled at the time of the reporting of the exception, then the interrupt will be synchronous and precise and MCSRR0 will be set to the instruction address which led to the exception. If Machine Check interrupts are *not* enabled at the time of the reporting of an Instruction Machine Check exception, then a Machine Check interrupt will *not* be generated (*ever*, even if and when MSR[ME] is subsequently set to 1), although the ESR[MCI] field will be set to 1 to indicate that the exception has occurred and that the instruction associated with the exception has been executed.

Instruction Asynchronous Machine Check, Data Asynchronous Machine Check, and TLB Asynchronous Machine Check exceptions, on the other hand, are handled in an “asynchronous” fashion. That is, the address reported in MCSRR0 may not be related to the instruction which prompted the access which led, directly or indirectly, to the Machine Check exception. The address may be that of an instruction before or after the exception-causing instruction, or it may reference the exception causing instruction, depending on the nature of the access, the type of error encountered, and the circumstances of the instruction’s execution within the processor pipeline. If MSR[ME] is 0 at the time of a Machine Check exception that is handled in this asynchronous way, a Machine Check interrupt *will* subsequently occur if and when MSR[ME] is set to 1.

See *Machine Check Interrupt* on page 263 for more detailed information on Machine Check interrupts.

9.3 Interrupt Processing

Associated with each kind of interrupt is an *interrupt vector*, that is, the address of the initial instruction that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the processor state in certain registers, identifying the cause of the interrupt in another register, and continuing execution at the corresponding interrupt vector location. When an exception exists and the corresponding interrupt type is enabled, the following actions are performed, in order:

1. SRR0 (for non-critical class interrupts) or CSRR0 (for critical class interrupts) or MCSRR0 (for Machine Check interrupts) is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. The ESR is loaded with information specific to the exception type. Note that many interrupt types can only be caused by a single type of exception, and thus do not need nor use an ESR setting to indicate the cause of the interrupt. Machine Check interrupts load the MCSR
3. SRR1 (for non-critical class interrupts) or CSRR1 (for critical class interrupts) or MCSRR1 (for Machine Check interrupts) is loaded with a copy of the contents of the MSR.

4. The MSR is updated as described below. The new values take effect beginning with the first instruction following the interrupt.
   - MSR[WE,EE,PR,FP,FE0,DWE,FE1,IS,DS] are set to 0 by all interrupts.
   - MSR[CE,DE] are set to 0 by all critical class interrupts and left unchanged by all non-critical class interrupts.
   - MSR[ME] is set to 0 by Machine Check interrupts and left unchanged by all other interrupts.

   See *Machine State Register (MSR)* on page 253 for more detail on the definition of the MSR.
5. Instruction fetching and execution resumes, using the new MSR value, at the interrupt vector address, which is specific to the interrupt type, and is determined as follows:

   $\text{IVPR}_{0:15}$ || $\text{IVOR}n_{16:27}$ || 0b0000

   where n specifies the IVOR register to be used for a particular interrupt type (see *Interrupt Vector Offset Registers (IVOR0:IVOR15)* on page 257).

At the end of a non-critical interrupt handling routine, execution of an **rfi** causes the MSR to be restored from the contents of SRR1 and instruction execution to resume at the address contained in SRR0. Likewise, execution of an **rfci** performs the same function at the end of a critical interrupt handling routine, using CSRR0 instead of SRR0 and CSRR1 instead of SRR1. **rfmci** uses MCSRR0 and MCSRR1 in the same manner.

Programming Note: In general, at process switch, due to possible process interlocks and possible data availability requirements, the operating system needs to consider executing the following instructions.

- **stwcx.**, to clear the reservation if one is outstanding, to ensure that a **lwarx** in the "old" process is not paired with a **stwcx.** in the "new" process. See the instruction descriptions for **lwarx** and **stwcx.** in *Instruction Set* on page 343 for more information on storage reservations.
- **msync**, to ensure that all storage operations of an interrupted process are complete with respect to other processors before that process begins executing on another processor.
- **isync**, **rfi**, **rfci,** or **rfmci**, to ensure that the instructions in the "new" process execute in the "new" context.

9.3.1 Partially Executed Instructions

In general, the architecture permits load and store instructions to be partially executed, interrupted, and then to be restarted from the beginning upon return from the interrupt. In order to guarantee that a particular load or store instruction will complete without being interrupted and restarted, software must mark the storage being referred to as Guarded, and must use an elementary (not a string or multiple) load or store that is aligned on an operand-sized boundary.

In order to guarantee that load and store instructions can, in general, be restarted and completed correctly without software intervention, the following rules apply when an instruction is partially executed and then interrupted:

- For an elementary load, no part of the target register (GPR(RT), FPR(FRT), or auxiliary processor register) will have been altered.
- For the "update" forms of load and store instructions, the update register, GPR(RA), will not have been altered.

On the other hand, the following effects are permissible when certain instructions are partially executed and then restarted:

- For any store instruction, some of the bytes at the addressed storage location may have been accessed and/or updated (if write access to that page in which bytes were altered is permitted by the access control mechanism). In addition, for the **stwcx.** instruction, if the address is not aligned on a word boundary, then the value in CR[CR0] is undefined, as is whether or not the reservation (if one existed) has been cleared if CCR1[L2COBE] is set to 0. However, if CCR1[L2COBE] bit is set to 1, stwcx. instruction will generate an alignment exception in this case. CR[CR0] will not be changed.
- For any load, some of the bytes at the addressed storage location may have been accessed (if read access to that page in which bytes were accessed is permitted by the access control mechanism). In addition, for the **lwarx** instruction, if the address is not aligned on a word boundary, it is undefined whether or not a reservation has been set. if CCR1[L2COBE] is set to 0. However, if CCR1[L2COBE] is set to 1, lwarx instruction generates an alignment exception in this case.
- For load multiple and load string instructions, some of the registers in the range to be loaded may have been altered. Including the addressing registers (GPR(RA), and possibly GPR(RB)) in the range to be loaded is an invalid form of these instructions (and a programming error), and thus the rules for partial execution do not protect against overwriting of these registers. Such possible overwriting of the addressing registers makes these invalid forms of load multiple and load strings inherently non-restartable.

In no case will access control be violated.

As previously stated, the only load or store instructions that are guaranteed to not be interrupted after being partially executed are elementary, aligned, guarded loads and stores. All others may be interrupted after being partially executed. The following list identifies the specific instruction types for which interruption after partial execution may occur, as well as the specific interrupt types that could cause the interruption:

1. Any load or store (except elementary, aligned, guarded):

   Critical Input

   Machine Check

   External Input

   Program (Imprecise Mode Floating-Point Enabled)

      Note that this type of interrupt can lead to partial execution of a load or store instruction under the architectural definition only; the PPC465 handles the imprecise modes of the Floating-Point Enabled exceptions precisely, and hence this type of interrupt will not lead to partial execution.

   Decrementer

   Fixed-Interval Timer

   Watchdog Timer

   Debug (Unconditional Debug Event)

2. Unaligned elementary load or store, or any load or store multiple or string:

   All of the above listed under item 1, plus the following:

   Alignment

   Data Storage (if the access crosses a memory page boundary)

   Debug (Data Address Compare, Data Value Compare)

9.4 Interrupt Processing Registers

The interrupt processing registers include the Save/Restore Registers (SRR0–SRR1), Critical Save/Restore Registers (CSRR0–CSRR1), Data Exception Address Register (DEAR), Interrupt Vector Offset Registers (IVOR0–IVOR15), Interrupt Vector Prefix Register (IVPR), and Exception Syndrome Register (ESR). Also described in this section is the Machine State Register (MSR), which belongs to the category of processor control registers.

9.4.1 Machine State Register (MSR)

The MSR is a register of its own unique type that controls important chip functions, such as the enabling or disabling of various interrupt types.

The MSR can be written from a GPR using the **mtmsr** instruction. The contents of the MSR can be read into a GPR using the **mfmsr** instruction. The MSR[EE] bit can be set or cleared atomically using the **wrtee** or **wrteei** instructions. The MSR contents are also automatically saved, altered, and restored by the interrupt-handling mechanism.

Figure 9-1. 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 Input and Watchdog Timer interrupts are disabled. 1 Critical Input and Watchdog Timer interrupts are enabled.	
15		Reserved	
16	EE	External Interrupt Enable 0 External Input, Decrementer, and Fixed Interval Timer interrupts are disabled. 1 External Input, Decrementer, and Fixed Interval Timer interrupts are enabled.	
17	PR	Problem State 0 Supervisor state (privileged instructions can be executed) 1 Problem state (privileged instructions can not be executed)	
18	FP	Floating Point Available 0 The processor cannot execute floating-point instructions 1 The processor can execute floating-point instructions	
19	ME	Machine Check Enable 0 Machine Check interrupts are disabled 1 Machine Check interrupts are enabled.	
20	FE0	Floating-point exception mode 0 0 If MSR[FE1] = 0, ignore exceptions mode; if MSR[FE1] = 1, imprecise nonrecoverable mode 1 If MSR[FE1] = 0, imprecise recoverable mode; if MSR[FE1] = 1, precise mode	

21	DWE	Debug Wait Enable 0 Disable debug wait mode. 1 Enable debug wait mode.	
22	DE	Debug interrupt Enable 0 Debug interrupts are disabled. 1 Debug interrupts are enabled.	
23	FE1	Floating-point exception mode 1 0 If MSR[FE0] = 0, ignore exceptions mode; if MSR[FE0] = 1, imprecise recoverable mode 1 If MSR[FE0] = 0, imprecise non-recoverable mode; if MSR[FE0] = 1, precise mode	
24:25		Reserved	
26	IS	Instruction Address Space 0 All instruction storage accesses are directed to address space 0 (TS = 0 in the relevant TLB entry). 1 All instruction storage accesses are directed to address space 1 (TS = 1 in the relevant TLB entry).	
27	DS	Data Address Space 0 All data storage accesses are directed to address space 0 (TS = 0 in the relevant TLB entry). 1 All data storage accesses are directed to address space 1 (TS = 1 in the relevant TLB entry).	
28:31		Reserved	

9.4.2 Save/Restore Register 0 (SRR0)

SRR0 is an SPR that is used to save machine state on non-critical interrupts, and to restore machine state when an **rfi** is executed. When a non-critical interrupt occurs, SRR0 is set to an address associated with the process which was executing at the time. When **rfi** is executed, instruction execution returns to the address in SRR0.

In general, SRR0 contains the address of the instruction that caused the non-critical interrupt, or the address of the instruction to return to after a non-critical interrupt is serviced. See the individual descriptions under *Interrupt Definitions* on page 260 for an explanation of the precise address recorded in SRR0 for each non-critical interrupt type.

SRR0 can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 9-2. Save/Restore Register 0 (SRR0)

0:29		Return address for non-critical interrupts	
30:31		Reserved	

9.4.3 Save/Restore Register 1 (SRR1)

SRR1 is an SPR that is used to save machine state on non-critical interrupts, and to restore machine state when an **rfi** is executed. When a non-critical interrupt is taken, the contents of the MSR (prior to the MSR being cleared by the interrupt) are placed into SRR1. When **rfi** is executed, the MSR is restored with the contents of SRR1.

Bits of SRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note: An MSR bit that is reserved may be altered by **rfi**, consistent with the value being restored from SRR1.

SRR1 can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 9-3. Save/Restore Register 1 (SRR1)

0:31		Copy of the MSR at the time of a non-critical interrupt.	

9.4.4 Critical Save/Restore Register 0 (CSRR0)

CSRR0 is an SPR that is used to save machine state on critical interrupts, and to restore machine state when an **rfci** is executed. When a critical interrupt occurs, CSRR0 is set to an address associated with the process which was executing at the time. When **rfci** is executed, instruction execution returns to the address in CSRR0.

In general, CSRR0 contains the address of the instruction that caused the critical interrupt, or the address of the instruction to return to after a critical interrupt is serviced. See the individual descriptions under *Interrupt Definitions* on page 260 for an explanation of the precise address recorded in CSRR0 for each critical interrupt type.

CSRR0 can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 9-4. Critical Save/Restore Register 0 (CSRR0)

0:29		Return address for critical interrupts	
30:31		Reserved	

9.4.5 Critical Save/Restore Register 1 (CSRR1)

CSRR1 is an SPR that is used to save machine state on critical interrupts, and to restore machine state when an **rfci** is executed. When a critical interrupt is taken, the contents of the MSR (prior to the MSR being cleared by the interrupt) are placed into CSRR1. When **rfci** is executed, the MSR is restored with the contents of CSRR1.

Bits of CSRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note: An MSR bit that is reserved may be altered by **rfci**, consistent with the value being restored from CSRR1.

CSRR1 can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 9-5. Critical Save/Restore Register 1 (CSRR1)

0:31		Copy of the MSR when a critical interrupt is taken	

9.4.6 Machine Check Save/Restore Register 0 (MCSRR0)

MCSRR0 is an SPR that is used to save machine state on Machine Check interrupts, and to restore machine state when an **rfmci** is executed. When a machine check interrupt occurs, MCSRR0 is set to an address associated with the process which was executing at the time. When **rfmci** is executed, instruction execution returns to the address in MCSRR0.

In general, MCSRR0 contains the address of the instruction that caused the Machine Check interrupt, or the address of the instruction to return to after a machine check interrupt is serviced. See the individual descriptions under *Interrupt Definitions* on page 260 for an explanation of the precise address recorded in MCSRR0 for each Machine Check interrupt type.

MCSRR0 can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 9-6. Machine Check Save/Restore Register 0 (MCSRR0)

0:29		Return address for machine check interrupts	
30:31		Reserved	

9.4.7 Machine Check Save/Restore Register 1 (MCSRR1)

MCSRR1 is an SPR that is used to save machine state on Machine Check interrupts, and to restore machine state when an **rfmci** is executed. When a machine check interrupt is taken, the contents of the MSR (prior to the MSR being cleared by the interrupt) are placed into MCSRR1. When **rfmci** is executed, the MSR is restored with the contents of MCSRR1.

Bits of MCSRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note: An MSR bit that is reserved may be altered by **rfmci**, consistent with the value being restored from MCSRR1.

MCSRR1 can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 9-7. Machine Check Save/Restore Register 1 (MCSRR1)

0:31		Copy of the MSR at the time of a machine check interrupt.	

9.4.8 Data Exception Address Register (DEAR)

The DEAR contains the address that was referenced by a load, store, or cache management instruction that caused an Alignment, Data TLB Miss, or Data Storage exception.

The DEAR can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 9-8. Data Exception Address Register (DEAR)

0:31		Address of data exception for Data Storage, Alignment, and Data TLB Error interrupts	

9.4.9 Interrupt Vector Offset Registers (IVOR0:IVOR15)

An IVOR specifies the quad word (16 byte)-aligned interrupt vector offset from the base address provided by the IVPR (see *Interrupt Vector Prefix Register (IVPR)* on page 258) for its respective interrupt type. IVOR0:IVOR15 are provided for the defined interrupt types. The interrupt vector effective address is formed as follows:

$$IVPR_{0:15} \mathbin{||} IVORn_{16:27} \mathbin{||} 0b0000$$

where *n* specifies the IVOR register to be used for the particular interrupt type.

Any IVOR can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

The following figure shows the IVOR field definitions, while *Table 9-1* identifies the specific IVOR register associated with each interrupt type.

Figure 9-9. Interrupt Vector Offset Registers (IVOR0:IVOR15)

0:15		Reserved	
16:27	IVO	Interrupt Vector Offset	
28:31		Reserved	

Table 9-1. Interrupt Types Associated with each IVOR

IVOR	Interrupt Type
IVOR0	Critical Input
IVOR1	Machine Check
IVOR2	Data Storage
IVOR3	Instruction Storage
IVOR4	External Input
IVOR5	Alignment
IVOR6	Program
IVOR7	Floating Point Unavailable
IVOR8	System Call
IVOR9	Auxiliary Processor Unavailable
IVOR10	Decrementer
IVOR11	Fixed Interval Timer
IVOR12	Watchdog Timer
IVOR13	Data TLB Error
IVOR14	Instruction TLB Error
IVOR15	Debug

9.4.10 Interrupt Vector Prefix Register (IVPR)

The IVPR provides the high-order 16 bits of the effective address of the interrupt vectors, for all interrupt types. The interrupt vector effective address is formed as follows:

$IVPR_{0:15} || IVORn_{16:27} || 0b0000$

where *n* specifies the IVOR register to be used for the particular interrupt type.

The IVPR can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 9-10. Interrupt Vector Prefix Register (IVPR)

0:15	IVP	Interrupt Vector Prefix	
16:31		Reserved	

9.4.11 Exception Syndrome Register (ESR)

The ESR provides a *syndrome* to differentiate between the different kinds of exceptions that can generate the same interrupt type. Upon the generation of one of these types of interrupt, the bit or bits corresponding to the specific exception that generated the interrupt is set, and all other ESR bits are cleared. Other interrupt types do not affect the contents of the ESR. *Figure 9-11* provides a summary of the fields of the ESR along with their definitions. See the individual interrupt descriptions under “Interrupt Definitions” on page 260 for an explanation of the ESR settings for each interrupt type, as well as a more detailed explanation of the function of certain ESR fields.

The ESR can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 9-11. Exception Syndrome Register (ESR)

0	MCI	Machine Check—Instruction Fetch Exception 0 Instruction Machine Check exception did not occur. 1 Instruction Machine Check exception occurred.	This is an implementation-dependent field of the ESR and is not part of the PowerPC Book-E Architecture.
1:3		Reserved	
4	PIL	Program Interrupt—Illegal Instruction Exception 0 Illegal Instruction exception did not occur. 1 Illegal Instruction exception occurred.	
5	PPR	Program Interrupt—Privileged Instruction Exception 0 Privileged Instruction exception did not occur. 1 Privileged Instruction exception occurred.	
6	PTR	Program Interrupt—Trap Exception 0 Trap exception did not occur. 1 Trap exception occurred.	
7	FP	Floating Point Operation 0 Exception was not caused by a floating point instruction. 1 Exception was caused by a floating point instruction.	

8	ST	Store Operation 0 Exception was not caused by a store-type storage access or cache management instruction. 1 Exception was caused by a store-type storage access or cache management instruction.	
9		Reserved	
10:11	DLK	Data Storage Interrupt—Locking Exception 00 Locking exception did not occur. 01 Locking exception was caused by **dcbf**. 10 Locking exception was caused by **icbi**. 11 Reserved	
12	AP	AP Operation 0 Exception was not caused by an auxiliary processor instruction. 1 Exception was caused by an auxiliary processor instruction.	
13	PUO	Program Interrupt—Unimplemented Operation Exception 0 Unimplemented Operation exception did not occur. 1 Unimplemented Operation exception occurred.	
14	BO	Byte Ordering Exception 0 Byte Ordering exception did not occur. 1 Byte Ordering exception occurred.	
15	PIE	Program Interrupt—Imprecise Exception 0 Exception occurred precisely; SRR0 contains the address of the instruction that caused the exception. 1 Exception occurred imprecisely; SRR0 contains the address of an instruction after the one which caused the exception.	This field is only set for a Floating-Point Enabled exception type Program interrupt, and then only when the interrupt occurs imprecisely due to MSR[FE0,FE1] being set to a non-zero value when an attached floating-point unit is already signaling the Floating-Point Enabled exception (that is, FPSCR[FEX] is already 1).
16:26		Reserved	
27	PCRE	Program Interrupt—Condition Register Enable 0 Instruction which caused the exception is not a floating-point CR-updating instruction. 1 Instruction which caused the exception is a floating-point CR-updating instruction.	This is an implementation-dependent field of the ESR and is not part of the PowerPC Book-E Architecture. This field is only defined for a Floating-Point Enabled exception type Program interrupt, and then only when ESR[PIE] is 0.
28	PCMP	Program Interrupt—Compare 0 Instruction which caused the exception is not a floating-point compare type instruction 1 Instruction which caused the exception is a floating-point compare type instruction.	This is an implementation-dependent field of the ESR and is not part of the PowerPC Book-E Architecture. This field is only defined for a Floating-Point Enabled exception type Program interrupt, and then only when ESR[PIE] is 0.
29:31	PCRF	Program Interrupt—Condition Register Field If ESR[PCRE]=1, this field indicates which CR field was to be updated by the floating-point instruction which caused the exception.	This is an implementation-dependent field of the ESR and is not part of the PowerPC Book-E Architecture. This field is only defined for a Floating-Point Enabled exception type Program interrupt, and then only when ESR[PIE] is 0.

9.4.12 Machine Check Status Register (MCSR)

The MCSR contains status to allow the Machine Check interrupt handler software to determine the cause of a machine check exception. Any Machine Check exception that is handled as an asynchronous interrupt sets MCSR[MCS] and other appropriate bits of the MCSR. If MSR[ME] and MCSR[MCS] are both set, the machine will take a Machine Check interrupt. See *Machine Check Interrupt* on page 263.

The MCSR is read into a GPR using **mfspr**. Clearing the MCSR is performed using **mtspr** by placing a 1 in the GPR source register in all bit positions which are to be cleared in the MCSR, and a 0 in all other bit positions. The data written from the GPR to the MCSR is not direct data, but a mask. A 1 clears the bit and a 0 leaves the corresponding MCSR bit unchanged.

Figure 9-12. Machine Check Status Register (MCSR)

0	MCS	Machine Check Summary 0 No asynchronous machine check exception pending 1 Asynchronous machine check exception pending	Set when a machine check exception occurs that is handled in the asynchronous fashion. One of MCSR bits 1:7 will be set simultaneously to indicate the exception type. When MSR[ME] and this bit are both set, Machine Check interrupt is taken.
1	IB	Instruction PLB Error 0 Exception not caused by Instruction Read PLB interrupt request (IRQ) 1 Exception caused by Instruction Read PLB interrupt request (IRQ)	
2:3		Reserved	
4	TLBP	Translation Look Aside Buffer Parity Error 0 Exception not caused by TLB parity error 1 Exception caused by TLB parity error	
5	ICP	Instruction Cache Parity Error 0 Exception not caused by I-cache parity error 1 Exception caused by I-cache parity error	
6	DCSP	Data Cache Search Parity Error 0 Exception not caused by DCU Search parity error 1 Exception caused by DCU Search parity error	Set if and only If the DCU parity error was discovered during a DCU Search operation. See *Data Cache Parity Operations* on page 153.
7	DCFP	Data Cache Flush Parity Error 0 Exception not caused by DCU Flush parity error 1 Exception caused by DCU Flush parity error	Set if and only If the DCU parity error was discovered during a DCU Flush operation. See *Data Cache Parity Operations* on page 153.
8	IMPE	Imprecise Machine Check Exception 0 No imprecise machine check exception occurred. 1 Imprecise machine check exception occurred.	Set if a machine check exception occurs that sets MCSR[MCS] (or would if it were not already set) and MSR[ME] = 0.
9:31		Reserved	

9.5 Interrupt Definitions

Table 9-2 provides a summary of each interrupt type, in the order corresponding to their associated IVOR register. The table also summarizes the various exception types that may cause that interrupt type; the classification of the interrupt; which ESR bit(s) can be set, if any; and which mask bit(s) can mask the interrupt type, if any.

Detailed descriptions of each of the interrupt types follow the table.

.

Table 9-2. Interrupt and Exception Types

IVOR	Interrupt Type	Exception Type	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (See Note 4)	MSR Mask Bit(s)	DBCR0/TCR Mask Bit	Notes
IVOR0	Critical Input	Critical Input	x			x		CE		1
IVOR1	Machine Check	Instruction Machine Check					[MCI]	ME		2
		Data Machine Check						ME		2
		TLB Machine Check						ME		2
IVOR2	Data Storage	Read Access Control		x			[FP,AP]			
		Write Access Control		x			ST,[FP,AP]			
		Cache Locking		x			{DLK$_0$,DLK$_1$}			
		Byte Ordering		x			BO,[ST],[FP,AP]			5
IVOR3	Instruction Storage	Execute Access Control		x						
		Byte Ordering		x			BO			6
IVOR4	External Input	External Input	x					EE		1
IVOR5	Alignment	Alignment		x			[ST],[FP,AP]			
IVOR6	Program	Illegal Instruction		x			PIL			
		Privileged Instruction		x			PPR,[AP]			
		Trap		x			PTR			
		FP Enabled		x	x		FP,[PIE],[PCRE] {PCMP,PCRF}	FE0 FE1		8
		AP Enabled		x			AP			8
		Unimplemented Operations		x			PUO,[FP,AP]			7
IVOR7	FP Unavailable	FP Unavailable		x						8
IVOR8	System Call	System Call		x						
IVOR9	AP Unavailable	AP Unavailable		x						8
IVOR10	Decrementer	Decrementer	x					EE	DIE	
IVOR11	Fixed Interval Timer	Fixed Interval Timer	x					EE	FIE	
IVOR12	Watchdog Timer	Watchdog Timer	x			x		CE	WIE	
IVOR13	Data TLB Error	Data TLB Miss		x			[ST],[FP,AP]			
IVOR14	Instruction TLB Error	Instruction TLB Miss		x						

Table 9-2. Interrupt and Exception Types (continued)

IVOR	Interrupt Type	Exception Type	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (See Note 4)	MSR Mask Bit(s)	DBCR0/TCR Mask Bit	Notes
IVOR15	Debug	Trap		x	x	x		DE	IDM	3
		Instruction Address Compare		x	x	x		DE	IDM	3
		Data Address Compare	x	x	x	x		DE	IDM	3
		Data Value Compare	x	x	x	x		DE	IDM	3
		Instruction Complete		x	x	x		DE	IDM	3
		Branch Taken		x		x		DE	IDM	3
		Return		x	x	x		DE	IDM	3
		Interrupt	x			x		DE	IDM	
		Unconditional	x			x		DE	IDM	

Table Notes

1. Although it is not specified as part of Book E, it is common for system implementations to provide, as part of the interrupt controller, independent mask and status bits for the various sources of Critical Input and External Input interrupts.
2. Machine Check interrupts are not classified as asynchronous nor synchronous. They are also not classified as critical or non-critical, because they use their own unique set to Save/Restore Registers, MCSRR0/1. See *Machine Check Interrupts* on page 249, and *Machine Check Interrupt* on page 263.
3. Debug exceptions have special rules regarding their interrupt classification (synchronous or asynchronous, and precise or imprecise), depending on the particular debug mode being used and other conditions (see *Debug Interrupt* on page 278).
4. In general, when an interrupt causes a particular ESR bit or bits to be set as indicated in the table, it also causes all other ESR bits to be cleared. Special rules apply to the ESR[MCI] field; see *Machine Check Interrupt* on page 263. If no ESR setting is indicated for any of the exception types within a given interrupt type, then the ESR is unchanged for that interrupt type.

   The syntax for the ESR setting indication is as follows:

   [xxx] means ESR[xxx] *may* be set

   [xxx,yyy,zzz] means any *one* (or none) of ESR[xxx] or ESR[yyy] or ESR[zzz] *may* be set, but never more than one

   {xxx,yyy,zzz} means that any combination of ESR[xxx], ESR[yyy], and ESR[zzz] *may* be set, including all or none

   xxx means ESR[xxx] *will* be set
5. Byte Ordering exception type Data Storage interrupts can only occur when the PPC465 is connected to a floating-point unit or auxiliary processor, and then only when executing FP or AP load or store instructions. See *Data Storage Interrupt* on page 265 for more detailed information on these kinds of exceptions.
6. Byte Ordering exception type Instruction Storage interrupts are defined by the PowerPC Book-E architecture, but cannot occur within the PPC465. The core is capable of executing instructions from both big endian and little endian code pages.
7. Unimplemented Operation exception type Program interrupts can only occur when the PPC465 is connected to a floating-point unit or auxiliary processor, and then only when executing instruction opcodes which are recognized by the floating-point unit or auxiliary processor but are not implemented within the hardware.
8. Floating-Point Unavailable and Auxiliary Processor Unavailable interrupts, as well as Floating-Point Enabled and Auxiliary Processor Enabled exception type Program interrupts, can only occur when the PPC465 is connected to a floating-point unit or auxiliary processor, and then only when executing instruction opcodes which are recognized by the floating-point unit or auxiliary processor, respectively.

9.5.1 Critical Input Interrupt

A Critical Input interrupt occurs when no higher priority exception exists, a Critical Input exception is presented to the interrupt mechanism, and MSR[CE] = 1. A Critical Input exception is caused by the activation of an asynchronous input to the PPC465. Although the only mask for this interrupt type within the core is the MSR[CE] bit, system implementations typically provide an alternative means for independently masking the interrupt requests from the various devices which collectively may activate the processor core Critical Input interrupt request input.

Note: MSR[CE] also enables the Watchdog Timer interrupt.

When a Critical Input interrupt occurs, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR0[IVO] || 0b0000.

Critical Save/Restore Register 0 (CSRR0): Set to the effective address of the next instruction to be executed.

Critical Save/Restore Register 1 (CSRR1): Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR): ME: Unchanged. All other MSR bits set to 0.

Programming Note: Software is responsible for taking any action(s) that are required by the implementation in order to clear any Critical Input exception status (such that the Critical Input interrupt request input signal is deasserted) before reenabling MSR[CE], in order to avoid another, redundant Critical Input interrupt.

9.5.2 Machine Check Interrupt

A Machine Check interrupt occurs when no higher priority exception exists, a Machine Check exception is presented to the interrupt mechanism, and MSR[ME] = 1. The PowerPC architecture specifies Machine Check interrupts as neither synchronous nor asynchronous, and indeed the exact causes and details of handling such interrupts are implementation dependent. Regardless, for this particular processor core, it is useful to describe the handling of interrupts caused by various types of Machine Check exceptions in those terms. The processor core includes four types of Machine Check exceptions. They are:

Instruction Synchronous Machine Check exception

An Instruction Synchronous Machine Check exception is caused when timeout or read error is signaled on the instruction read L2 cache, L2C, interface during an instruction fetch operation.

Such an exception is not presented to the interrupt handling mechanism, however, unless and until such time as the execution is attempted of an instruction at an address associated with the instruction fetch for which the Instruction Machine Check exception was asserted. When the exception is presented, the ESR[MCI] bit will be set to indicated the type of exception, regardless of the state of the MSR[ME] bit.

If MSR[ME] is 1 when the Instruction Machine Check exception is presented to the interrupt mechanism, then execution of the instruction associated with the exception will be suppressed, a Machine Check interrupt will occur, and the interrupt processing registers will be updated as described on page 264. If MSR[ME] is 0, however, then the instruction associated with the exception will be processed as though the exception did not exist and a Machine Check interrupt will *not* occur (*ever*, even if and when MSR[ME] is subsequently set to 1), although the ESR will still be updated as described on page 264.

Instruction Asynchronous Machine Check exception

An Instruction Asynchronous Machine Check exception is caused when either:

- an instruction cache parity error is detected
- the read interrupt request is asserted on the instruction read L2C interface.

Data Asynchronous Machine Check exception

A Data Asynchronous Machine Check exception is caused when one of the following occurs:

- a timeout, read error, or read interrupt request is signaled on the data read read L2 cache, L2C, interface interface, during a data read operation
- a timeout, write error, or write interrupt request is signaled on the data write read L2 cache, L2C, interface interface, during a data write operation
- a parity error is detected on an access to the data cache.

TLB Asynchronous Machine Check exception

A TLB Asynchronous Machine Check exception is caused when a parity error is detected on an access to the TLB.

When any Machine Check exception which is handled as an asynchronous interrupt occurs, it is immediately presented to the interrupt handling mechanism. MCSR[MCS] is set, as are other bits of the MCSR as appropriate. A Machine Check interrupt will occur immediately if MSR[ME] is 1, and the interrupt processing registers will be updated as described below. If MSR[ME] is 0, however, then the exception will be “recorded” by the setting of the MCSR[MCS] bit, and deferred until such time as MSR[ME] is subsequently set to 1. Any time the MCSR[MCS] and MSR[ME] are both set to 1, the Machine Check interrupt will be taken. Therefore, MCSR[MCS] must be cleared by software in the Machine Check interrupt handler before executing an **rfmci** to return to processing with MSR[ME] set to 1.

When a Machine Check interrupt occurs, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR1[IVO] || 0b0000.

Machine Check Save/Restore Register 0 (MCSRR0)

For an Instruction Synchronous Machine Check exception, set to the effective address of the instruction presenting the exception. For an Instruction Asynchronous Machine Check, Data Asynchronous Machine Check, or TLB Asynchronous Machine Check exception, set to the effective address of the next instruction to be executed.

Machine Check Save/Restore Register 1 (MCSRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

All MSR bits set to 0.

Exception Syndrome Register (ESR)

MCI	Set to 1 for an Instruction Machine Check exception; otherwise left unchanged.

All other defined ESR bits are set to 0 for an Instruction Machine Check exception; otherwise they are left unchanged.

Programming Note: If an Instruction Synchronous Machine Check exception is associated with an instruction, and execution of that instruction is attempted while MSR[ME] is 0, then no Machine Check interrupt will occur, but ESR[MCI] will still be set to 1 when the instruction actually executes. Once set, ESR[MCI] cannot be cleared except by software, using the **mtspr** instruction. When processing a Machine Check interrupt handler, software should query ESR[MCI] to determine the type of Machine Check exception, and then clear ESR[MCI]. Then, prior to re-enabling Machine Check interrupts by setting MSR[ME] to 1, software should query the status of ESR[MCI] again to determine whether any additional Instruction Machine Check exceptions have occurred while MSR[ME] was disabled.

Machine Check Status Register (MCSR)

The MCSR collects status for the Machine Check exceptions that are handled as asynchronous interrupts. MCSR[MCS] is set by any Instruction Asynchronous Machine Check exception, Data Asynchronous Machine Check exception, or TLB Asynchronous Machine Check exception. Other bits in the MCSR are set to indicate the exact type of Machine Check exception.

MCS	Set to 1.
IB	Set to 1 if Instruction Read PLB Interrupt Request (IRQ) is asserted; otherwise set to 0.
TLBP	Set to 1 if the exception is a TLB parity error; otherwise set to 0.
ICP	Set to 1 if the exception is an instruction cache parity error; otherwise set to 0.
DCSP	Set to 1 if the exception is a data cache parity error that resulted during a DCU Search operation; otherwise set to 0. See *Data Cache Parity Operations* on page 153.
DCFP	Set to 1 if the exception is a data cache parity error that resulted during a DCU Flush operation; otherwise set to 0. See *Data Cache Parity Operations* on page 153.
IMPE	Set to 1 if MCSR[MCS] is set (or would be, if it were not already set) and MSR[ME] = 0; otherwise set to 0. When set, this bit indicates that a Machine Check exception happened while Machine Check interrupts were disabled.

See "Machine Check Interrupts" on page 249 for more information on the handling of Machine Check interrupts within the PPC465.

Programming Note: If an Instruction Synchronous Machine Check exception occurs (i.e. an error occurs on the L2C transfer that is intended to fill a line in the instruction cache, any data associated with the exception will *not* be placed into the instruction cache. On the other hand, if a Data Asynchronous Machine Check exception occurs due to a L2C error during a cacheable read operation, the data associated with the exception will be placed into the data cache, and could subsequently be loaded into a register. Similarly, if a Data Asynchronous Machine Check exception due to a L2C error occurs during a caching inhibited read operation, the data associated with the exception will be read into a register. Data Asynchronous Machine Check exceptions resulting from parity errors may or may not corrupt a GPR value, depending on the setting of the CCR0[PRE] field. See *Data Cache Parity Operations* on page 153.

Since a **dcbz** instruction establishes a real address in the data cache without actually reading the block of data from memory, it is possible for a delayed Data Machine Check exception to occur if and when a line established by a **dcbz** instruction is cast-out of the data cache and written to memory, if the address of the cache line is not valid within the system implementation.

9.5.3 Data Storage Interrupt

A Data Storage interrupt *may* occur when no higher priority exception exists and a Data Storage exception is presented to the interrupt mechanism. The PPC465 includes four types of Data Storage exception. They are:

Read Access Control exception

A Read Access Control exception is caused by one of the following:

- While in user mode (MSR[PR] = 1), a load, **icbi**, **icbt**, **dcbst**, **dcbf**, **dcbt**, or **dcbtst** instruction attempts to access a location in storage that is not enabled for read access in user mode (that is, the TLB entry associated with the memory page being accessed has UR=0).
- While in supervisor mode (MSR[PR] = 0), a load, **icbi**, **icbt**, **dcbst**, **dcbf**, **dcbt**, or **dcbtst** instruction attempts to access a location in storage that is not enabled for read access in supervisor mode (that is, the TLB entry associated with the memory page being accessed has SR=0).

Programming Note: The instruction cache management instructions **icbi** and **icbt** are treated as "loads" from the addressed byte with respect to address translation and protection. These instruction cache management instructions use MSR[DS] rather than MSR[IS] to determine translation for their target effective address. Similarly, they use the read access control field (UR or SR) rather than the execute access control field (UX or SX) of the TLB entry to determine whether a Data Storage exception should occur. Instruction Storage exceptions and Instruction TLB Miss exceptions are associated with the *fetching* of instructions not with the *execution* of instructions. Data Storage exceptions and Data TLB Miss exceptions are associated with the *execution* of instruction cache management instructions, as well as with the execution of load, store, and data cache management instructions.

Write Access Control exception

A Write Access Control exception is caused by one of the following:

- While in user mode (MSR[PR] = 1), a store, **dcbz**, or **dcbi** instruction attempts to access a location in storage that is not enabled for write access in user mode (that is, the TLB entry associated with the memory page being accessed has UW=0).
- While in supervisor mode (MSR[PR] = 0), a store, **dcbz**, or **dcbi** instruction attempts to access a location in storage that is not enabled for write access in supervisor mode (that is, the TLB entry associated with the memory page being accessed has SW=0).

Byte Ordering exception

A Byte Ordering exception will occur when a floating-point unit or auxiliary processor is attached to the PPC465, and a floating-point or auxiliary processor load or store instruction attempts to access a memory page with a byte order which is not supported by the attached processor. Whether or not a given load or store instruction type is supported for a given byte order is dependent on the implementation of the floating-point or auxiliary processor. All integer load and store instructions supported by the PPC465 are supported for both big endian and little endian memory pages.

Cache Locking exception

A Cache Locking exception is caused by one of the following:

- While in user mode (MSR[PR] = 1) with MMUCR[IULXE]=1, execution of an **icbi** instruction is attempted. The exception occurs whether or not the cache line targeted by the **icbi** instruction is actually locked in the instruction cache.
- While in user mode (MSR[PR] = 1) with MMUCR[DULXE]=1, execution of a **dcbf** instruction is attempted. The exception occurs whether or not the cache line targeted by the **dcbf** instruction is actually locked in the data cache.

See *Level 1 Cache* on page 125 and *Memory Management Unit Control Register (MMUCR)* on page 236 for more information on cache locking and Cache Locking exceptions, respectively.

If a **stwcx.** instruction causes a Write Access Control exception, but the processor does not have the reservation from a **lwarx** instruction, then a Data Storage interrupt does not occur and the instruction completes, updating CR[CR0] to indicate the failure of the store due to the lost reservation.

However if CCR1[L2COBE] =1, the Write Access Control exception occurs regardless of the state of the reservation and CR[CR0] is updated to indicate the failure of the store due to the lost reservation.

PPC465 core will also generate a Data Storage Interrupt whenever a lwarx and stwcx. to W=1 or IL1D/IL2D/I=1 pages when CCR1[L2COBE]=1.

If a Data Storage exception occurs on any of the following instructions, then the instruction is treated as a no-op, and a Data Storage interrupt does not occur.

- **lswx** or **stswx** with a length of zero (although the target register of **lswx** will still be undefined, as it is whether or not a Data Storage exception occurs)
- **icbt**
- **dcbt**
- **dcbtst**

For all other instructions, if a Data Storage exception occurs, then execution of the instruction causing the exception is suppressed, a Data Storage interrupt is generated, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR2[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the instruction causing the Data Storage interrupt.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE	Unchanged.

All other MSR bits set to 0.

Data Exception Address Register (DEAR)

If the instruction causing the Data Storage exception does so with respect to the memory page targeted by the initial effective address calculated by the instruction, then the DEAR is set to this calculated effective address. On the other hand, if the Data Storage exception only occurs due to the instruction causing the exception crossing a memory page boundary, in that the exception is with respect to the attributes of the page accessed after crossing the boundary, then the DEAR is set to the address of the first byte within that page.

For example, consider a misaligned load word instruction that targets effective address 0x00000FFF, and that the page containing that address is a 4KB page. The load word will thus cross the page boundary, and access the next page starting at address 0x00001000. If a Read Access Control exception exists within the first page (because the Read Access Control field for that page is 0), the DEAR will be set to 0x00000FFF. On the other hand, if the Read Access Control field of the first page is 1, but the same field is 0 for the next page, then the Read Access Control exception exists only for the second page and the DEAR will be set to 0x00001000. Furthermore, the load word instruction in this latter scenario will have been partially executed (see “Partially Executed Instructions” on page 251).

Exception Syndrome Register (ESR)

FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to

0.

ST	Set to 1 if the instruction causing the interrupt is a store, **dcbz**, or **dcbi** instruction; otherwise set to 0.
$DLK_{0:1}$	Set to 0b10 if an **icbi** instruction caused a Cache Locking exception; set to 0b01 if a **dcbf** instruction caused a Cache Locking exception; otherwise set to 0b00. Note that a Read Access Control exception may occur in combination with a Cache Locking exception, in which case software would need to examine the TLB entry associated with the address reported in the DEAR to determine whether both exceptions had occurred, or just a Cache Locking exception.
AP	Set to 1 if the instruction causing the interrupt is an auxiliary processor load or store; otherwise set to 0.
BO	Set to 1 if the instruction caused a Byte Ordering exception; otherwise set to 0. Note that a Read or Write Access Control exception may occur in combination with a Byte Ordering exception, in which case software would need to examine the TLB entry associated with the address reported in the DEAR to determine whether both exceptions had occurred, or just a Byte Ordering exception.
MCI	Unchanged.

All other defined ESR bits are set to 0.

9.5.4 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists and an Instruction Storage exception is presented to the interrupt mechanism. Note that although an Instruction Storage exception may occur during an attempt to *fetch* an instruction, such an exception is not actually presented to the interrupt mechanism until an attempt is made to *execute* that instruction. The PPC465 includes one type of Instruction Storage exception. That is:

Execute Access Control exception

An Execute Access Control exception is caused by one of the following:

- While in user mode (MSR[PR] = 1), an instruction fetch attempts to access a location in storage that is not enabled for execute access in user mode (that is, the TLB entry associated with the memory page being accessed has UX = 0).
- While in supervisor mode (MSR[PR] = 0), an instruction fetch attempts to access a location in storage that is not enabled for execute access in supervisor mode (that is, the TLB entry associated with the memory page being accessed has SX = 0).

Architecture Note: The PowerPC Book-E architecture defines an additional Instruction Storage exception -- the Byte Ordering exception. This exception is defined to assist implementations that cannot support dynamically switching byte ordering between consecutive instruction fetches and/or cannot support a given byte order at all. The PPC465 however supports instruction fetching from both big endian and little endian memory pages, so this exception cannot occur.

When an Instruction Storage interrupt occurs, the processor suppresses the execution of the instruction causing the Instruction Storage exception, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR3[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the instruction causing the Instruction Storage interrupt.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE	Unchanged.

All other MSR bits set to 0.

Exception Syndrome Register (ESR)

BO	Set to 0.
MCI	Unchanged.

All other defined ESR bits are set to 0.

9.5.5 External Input Interrupt

An External Input interrupt occurs when no higher priority exception exists, an External Input exception is presented to the interrupt mechanism, and MSR[EE] = 1. An External Input exception is caused by the activation of an asynchronous input to the PPC465. Although the only mask for this interrupt type within the core is the MSR[EE] bit, system implementations typically provide an alternative means for independently masking the interrupt requests from the various devices which collectively may activate the core’s External Input interrupt request input.

Note: MSR[EE] also enables the External Input and Fixed-Interval Timer interrupts.

When an External Input interrupt occurs, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR4[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the next instruction to be executed.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE	Unchanged.

All other MSR bits set to 0.

Programming Note: Software is responsible for taking any action(s) that are required by the implementation in order to clear any External Input exception status (such that the External Input interrupt request input signal is deasserted) before reenabling MSR[EE], in order to avoid another, redundant External Input interrupt.

9.5.6 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists and an Alignment exception is presented to the interrupt mechanism. An Alignment exception occurs if execution of any of the following is attempted:

- An integer load or store instruction that references a data storage operand that is not aligned on an operand-sized boundary, when CCR0[FLSTA] is 1. Load and store multiple instructions are considered to reference word operands, and hence word-alignment is required for the target address of these instructions when CCR0[FLSTA] is 1. Load and store string instructions are considered to reference byte operands, and hence they cannot cause an Alignment exception due to CCR0[FLSTA] being 1, regardless of the target address alignment.

- A floating-point load or store instruction that references a data storage operand that is not aligned on a word.
- A **dcbz** instruction that targets a memory page that is either write-through required or caching inhibited.

If a **stwcx.** instruction causes an Alignment exception, and the processor does not have the reservation from a **lwarx** instruction, then an Alignment interrupt still occurs.

Programming Note: The architecture does not support the use of an unaligned effective address by the **lwarx** and **stwcx.** instructions. If an Alignment interrupt occurs due to the attempted execution of one of these instructions, the Alignment interrupt handler must not attempt to emulate the instruction, but instead should treat the instruction as a programming error.

When an Alignment interrupt occurs, the processor suppresses the execution of the instruction causing the Alignment exception, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR5[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the instruction causing the Alignment interrupt.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE	Unchanged.

All other MSR bits set to 0.

Data Exception Address Register (DEAR)

Set to the effective address of the target data operand as calculated by the instruction causing the Alignment exception. Note that for **dcbz**, this effective address is not necessarily the address of the first byte of the targeted cache block, but could be the address of any byte within the block (it will be the address calculated by the **dcbz** instruction).

Exception Syndrome Register (ESR)

FP	Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.
ST	Set to 1 if the instruction causing the interrupt is a store, **dcbz**, or **dcbi** instruction; otherwise set to 0.
AP	Set to 1 if the instruction causing the interrupt is an auxiliary processor load or store; otherwise set to 0.

All other defined ESR bits are set to 0.

9.5.7 Program Interrupt

A Program interrupt occurs when no higher priority exception exists, a Program exception is presented to the interrupt mechanism, and -- for the Floating-Point Enabled form of Program exception only -- MSR[FE0,FE1] is non-zero. The PPC465 includes six types of Program exception. They are:

Illegal Instruction exception

An Illegal Instruction exception occurs when execution is attempted of any of the following kinds of instructions:

- A reserved-illegal instruction
- When MSR[PR] = 1 (user mode), an **mtspr** or **mfspr** that specifies an SPRN value with $SPRN_5$ = 0 (user-mode accessible) that represents an unimplemented Special Purpose Register. For **mtspr**, this includes any SPR number other than the XER, LR, CTR, or USPRG0. For **mfspr**, this includes any SPR number other than the ones listed for **mtspr**, plus SPRG4-7, TBH, and TBL.
- A defined instruction which is not implemented within the PPC465, and which is not a floating-point instruction. This includes all instructions that are defined for 64-bit implementations only, as well as **tlbiva** and **mfapidi** (see the PowerPC Book-E specification)
- A defined floating-point instruction that is not recognized by an attached floating-point unit (or when no such floating-point unit is attached)
- An allocated instruction that is not implemented within the PPC465 and which is not recognized by an attached auxiliary processor (or when no such auxiliary processor is attached)

See *Instruction Classes* on page 53 for more information on the PPC465's support for defined and allocated instructions.

Privileged Instruction exception

A Privileged Instruction exception occurs when MSR[PR] = 1 and execution is attempted of any of the following kinds of instructions:

- a privileged instruction
- an **mtspr** or **mfspr** instruction that specifies an SPRN value with $SPRN_5$ = 1 (a Privileged Instruction exception occurs regardless of whether or not the SPR referenced by the SPRN value is defined)

Trap exception

A Trap exception occurs when any of the conditions specified in a **tw** or **twi** instruction are met. However, if Trap debug events are enabled (DBCR0[TRAP]=1), internal debug mode is enabled (DBCR0[IDM]=1), and Debug interrupts are enabled (MSR[DE]=1), then a Trap exception will cause a Debug interrupt to occur, rather than a Program interrupt.

See *Debug Facilities* on page 315 for more information on Trap debug events.

Unimplemented Operation exception

An Unimplemented Operation exception occurs when execution is attempted of any of the following kinds of instructions:

- a defined floating-point instruction that is recognized but not supported by an attached floating-point unit, when floating-point instruction processing is enabled (MSR[FP]=1).
- an allocated instruction that is not implemented within the PPC465, and is recognized but not supported by an attached auxiliary processor, when auxiliary processor instruction processing is enabled. The enabling of auxiliary processor instruction processing is implementation-dependent.

Floating-Point Enabled exception

A Floating-Point Enabled exception occurs when the execution or attempted execution of a defined floating-point instruction causes FPSCR[FEX] to be set to 1, in an attached floating-point unit. FPSCR[FEX] is the Floating-Point Status and Control Register Floating-Point Enabled Exception Summary bit (see the user's manual for the floating-point unit implementation for more details).

If MSR[FE0,FE1] is non-zero when the Floating-Point Enabled exception is presented to the interrupt mechanism, then a Program interrupt will occur, and the interrupt processing registers will be updated as described below. If MSR[FE0,FE1] are both 0, however, then a Program interrupt will *not* occur and the instruction associated with the exception will execute according to the definition of the floating-point unit (see the user's manual for the floating-point unit implementation). If and when MSR[FE0,FE1] are subsequently set to a non-zero value, and the Floating-Point Enabled exception is still being presented to the interrupt mechanism (that is, FPSCR[FEX] is still set), then a "delayed" Program interrupt will occur, updating the interrupt processing registers as described below.

See "Synchronous, Imprecise Interrupts" on page 248 for more information on this special form of "delayed" Floating-Point Enabled exception.

Auxiliary Processor Enabled exception

An Auxiliary Processor Enabled exception may occur due to the execution or attempted execution of an allocated instruction that is not implemented within the PPC465, but is recognized and supported by an attached auxiliary processor. The cause of such an exception is implementation-dependent. See the user's manual for the auxiliary processor implementation for more details.

When a Program interrupt occurs, the processor suppresses the execution of the instruction causing the Program exception (for all cases except the "delayed" form of Floating-Point Enabled exception described above), the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR6[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the instruction causing the Program interrupt, for all cases except the "delayed" form of Floating-Point Enabled exception described above.

For the special case of the delayed Floating-Point Enabled exception, where the exception was already being presented to the interrupt mechanism at the time MSR[FE0,FE1] was changed from 0 to a non-zero value, SRR0 is set to the address of the instruction that would have executed after the MSR-changing instruction. If the instruction which set MSR[FE0,FE1] was **rfi**, **rfci,** or **rfmci**, then CSRR0 is set to the address to which the **rfi**, **rfci,** or **rfmci** was returning, and not to the address of the instruction which was sequentially after the **rfi**, **rfci,** or **rfmci**.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE	Unchanged.

All other MSR bits set to 0.

Exception Syndrome Register (ESR)

PIL	Set to 1 for an Illegal Instruction exception; otherwise set to 0
PPR	Set to 1 for a Privileged Instruction exception; otherwise set to 0
PTR	Set to 1 for a Trap exception; otherwise set to 0
PUO	Set to 1 for an Unimplemented Operation exception; otherwise set to 0
FP	Set to 1 if the instruction causing the interrupt is a floating-point instruction; otherwise set to 0.
AP	Set to 1 if the instruction causing the interrupt is an auxiliary processor instruction; otherwise set to 0.
PIE	Set to 1 if a "delayed" form of Floating-point Enabled exception type Program interrupt; otherwise set to 0. The setting of ESR[PIE] to 1 indicates to the Program interrupt handler that the interrupt was imprecise due to being caused by the changing of MSR[FE0,FE1] and not directly by the execution of the floating-point instruction which caused the exception by setting

FPSCR[FEX]. Thus the Program interrupt handler can recognize that SRR0 contains the address of the instruction after the MSR-changing instruction, and not the address of the instruction that caused the Floating-Point Enabled exception.

PCRE	Set to 1 if a Floating-Point Enabled exception and the floating-point instruction which caused the exception was a CR-updating instruction. Note that ESR[PCRE] is undefined if ESR[PIE] is 1.
PCMP	Set to 1 if a Floating-Point Enabled exception and the instruction which caused the exception was a floating-point compare instruction. Note that ESR[PCMP] is undefined if ESR[PIE] is 1.
PCRF	Set to the number of the CR field (0 - 7) which was to be updated, if a Floating-Point Enabled exception and the floating-point instruction which caused the exception was a CR-updating instruction. Note that ESR[PCRF] is undefined if ESR[PIE] is 1.

All other defined ESR bits are set to 0.

Programming Note: The ESR[PCRE,PCMP,PCRF] fields are provided to assist the Program interrupt handler with the emulation of part of the function of the various floating-point CR-updating instructions, when any of these instructions cause a precise (“non-delayed”) Floating-Point Enabled exception type Program interrupt. The PowerPC Book-E floating-point architecture defines that when such exceptions occur, the CR is to be updated even though the rest of the instruction execution may be suppressed. The PPC465, however, does not support such CR updates when the instruction which is supposed to cause the update is being suppressed due to the occurrence of a synchronous, precise interrupt. Instead, the PPC465 records in the ESR[PCRE,PCMP,PCRF] fields information about the instruction causing the interrupt, to assist the Program interrupt handler software in performing the appropriate CR update manually.

9.5.8 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction which is recognized by an attached floating-point unit, and MSR[FP]=0.

When a Floating-Point Unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the Floating-Point Unavailable exception, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR7[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the next instruction causing the Floating-Point Unavailable interrupt.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE Unchanged.

All other MSR bits set to 0.

9.5.9 System Call Interrupt

A System Call interrupt occurs when no higher priority exception exists and a system call (**sc**) instruction is executed.

When a System Call interrupt occurs, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR8[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the instruction after the system call instruction.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE	Unchanged.

All other MSR bits set to 0.

9.5.10 Auxiliary Processor Unavailable Interrupt

An Auxiliary Processor Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute an auxiliary processor instruction which is not implemented within the PPC465 but which is recognized by an attached auxiliary processor, and auxiliary processor instruction processing is not enabled. The enabling of auxiliary processor instruction processing is implementation-dependent. See the user's manual for the attached auxiliary processor.

When an Auxiliary Processor Unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the Auxiliary Processor Unavailable exception, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR9[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the next instruction causing the Auxiliary Processor Unavailable interrupt.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE	Unchanged.

All other MSR bits set to 0.

9.5.11 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists, a Decrementer exception exists (TSR[DIS] = 1), and the interrupt is enabled (TCR[DIE] = 1 and MSR[EE] = 1). See *Timer Facilities* on page 307 for more information on Decrementer exceptions.

Note: MSR[EE] also enables the External Input and Fixed-Interval Timer interrupts.

When a Decrementer interrupt occurs, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR10[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the next instruction to be executed.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE Unchanged.

All other MSR bits set to 0.

Programming Note: Software is responsible for clearing the Decrementer exception status by writing to TSR[DIS], prior to reenabling MSR[EE], in order to avoid another, redundant Decrementer interrupt.

9.5.12 Fixed-Interval Timer Interrupt

A Fixed-Interval Timer interrupt occurs when no higher priority exception exists, a Fixed-Interval Timer exception exists (TSR[FIS] = 1), and the interrupt is enabled (TCR[FIE] = 1 and MSR[EE]=1). See *Timer Facilities* on page 307 for more information on Fixed Interval Timer exceptions.

Note: MSR[EE] also enables the External Input and Decrementer interrupts.

When a Fixed interval Timer interrupt occurs, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR11[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the next instruction to be executed.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE Unchanged.

All other MSR bits set to 0.

Programming Note: Software is responsible for clearing the Fixed Interval Timer exception status by writing to TSR[FIS], prior to reenabling MSR[EE], in order to avoid another, redundant Fixed Interval Timer interrupt.

9.5.13 Watchdog Timer Interrupt

A Watchdog Timer interrupt occurs when no higher priority exception exists, a Watchdog Timer exception exists (TSR[WIS] = 1), and the interrupt is enabled (TCR[WIE] = 1 and MSR[CE] = 1). See *Timer Facilities* on page 307 for more information on Watchdog Timer exceptions.

Note: MSR[CE] also enables the Critical Input interrupt.

When a Watchdog Timer interrupt occurs, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR12[IVO] || 0b0000.

Critical Save/Restore Register 0 (CSRR0)

Set to the effective address of the next instruction to be executed.

Critical Save/Restore Register 1 (CSRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

ME	Unchanged.

All other MSR bits set to 0.

Programming Note: Software is responsible for clearing the Watchdog Timer exception status by writing to TSR[WIS], prior to reenabling MSR[CE], in order to avoid another, redundant Watchdog Timer interrupt.

9.5.14 Data TLB Error Interrupt

A Data TLB Error interrupt *may* occur when no higher priority exception exists and a Data TLB Miss exception is presented to the interrupt mechanism. A Data TLB Miss exception occurs when a load, store, **icbi**, **icbt**, **dcbst**, **dcbf**, **dcbz**, **dcbi**, **dcbt**, or **dcbtst** instruction attempts to access a virtual address for which a valid TLB entry does not exist. See *Memory Management* on page 219 for more information on the TLB.

Programming Note: The instruction cache management instructions **icbi** and **icbt** are treated as “loads” from the addressed byte with respect to address translation and protection, and therefore use MSR[DS] rather than MSR[IS] as part of the calculated virtual address when searching the TLB to determine translation for their target storage address. Instruction TLB Miss exceptions are associated with the *fetching* of instructions not with the *execution* of instructions. Data TLB Miss exceptions are associated with the *execution* of instruction cache management instructions, as well as with the execution of load, store, and data cache management instructions.

If a **stwcx.** instruction causes a Data TLB Miss exception, and the processor does not have the reservation from a **lwarx** instruction, then a Data TLB Error interrupt still occurs.

If a Data TLB Miss exception occurs on any of the following instructions, then the instruction is treated as a no-op, and a Data TLB Error interrupt does not occur.

- **lswx** or **stswx** with a length of zero (although the target register of **lswx** will be undefined)
- **icbt**
- **dcbt**
- **dcbtst**

For all other instructions, if a Data TLB Miss exception occurs, then execution of the instruction causing the exception is suppressed, a Data TLB Error interrupt is generated, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR13[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the instruction causing the Data TLB Error interrupt.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE Unchanged.

All other MSR bits set to 0.

Data Exception Address Register (DEAR)

If the instruction causing the Data TLB Miss exception does so with respect to the memory page targeted by the initial effective address calculated by the instruction, then the DEAR is set to this calculated effective address. On the other hand, if the Data TLB Miss exception only occurs due to the instruction causing the exception crossing a memory page boundary, in that the missing TLB entry is for the page accessed after crossing the boundary, then the DEAR is set to the address of the first byte within that page.

As an example, consider a misaligned load word instruction that targets effective address 0x00000FFF, and that the page containing that address is a 4KB page. The load word will thus cross the page boundary, and attempt to access the next page starting at address 0x00001000. If a valid TLB entry does not exist for the first page, then the DEAR will be set to 0x00000FFF. On the other hand, if a valid TLB entry *does* exist for the first page, but not for the second, then the DEAR will be set to 0x00001000. Furthermore, the load word instruction in this latter scenario will have been partially executed (see “Partially Executed Instructions” on page 251).

Exception Syndrome Register (ESR)

FP	Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.
ST	Set to 1 if the instruction causing the interrupt is a store, **dcbz**, or **dcbi** instruction; otherwise set to 0.
AP	Set to 1 if the instruction causing the interrupt is an auxiliary processor load or store; otherwise set to 0.
MCI	Unchanged.

All other defined ESR bits are set to 0.

9.5.15 Instruction TLB Error Interrupt

An Instruction TLB Error interrupt occurs when no higher priority exception exists and an Instruction TLB Miss exception is presented to the interrupt mechanism. Note that although an Instruction TLB Miss exception may occur during an attempt to *fetch* an instruction, such an exception is not actually presented to the interrupt mechanism until an attempt is made to *execute* that instruction. An Instruction TLB Miss exception occurs when an instruction fetch attempts to access a virtual address for which a valid TLB entry does not exist. See *Memory Management* on page 219 for more information on the TLB.

When an Instruction TLB Error interrupt occurs, the processor suppresses the execution of the instruction causing the Instruction TLB Miss exception, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || IVOR14[IVO] || 0b0000.

Save/Restore Register 0 (SRR0)

Set to the effective address of the instruction causing the Instruction TLB Error interrupt.

Save/Restore Register 1 (SRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CE, ME, DE	Unchanged.

All other MSR bits set to 0.

9.5.16 Debug Interrupt

A Debug interrupt occurs when no higher priority exception exists, a Debug exception exists in the Debug Status Register (DBSR), the processor is in internal debug mode (DBCR0[IDM]=1), and Debug interrupts are enabled (MSR[DE] = 1). A Debug exception occurs when a debug event causes a corresponding bit in the DBSR to be set.

There are several types of Debug exception, as follows:

Instruction Address Compare (IAC) exception

An IAC Debug exception occurs when execution is attempted of an instruction whose address matches the IAC conditions specified by the various debug facility registers. This exception can occur regardless of debug mode, and regardless of the value of MSR[DE].

Data Address Compare (DAC) exception

A DAC Debug exception occurs when the DVC mechanism is not enabled, and execution is attempted of a load, store, **icbi**, **icbt**, **dcbst**, **dcbf**, **dcbz**, **dcbi**, **dcbt**, or **dcbtst** instruction whose target storage operand address matches the DAC conditions specified by the various debug facility registers. This exception can occur regardless of debug mode, and regardless of the value of MSR[DE].

Programming Note: The instruction cache management instructions **icbi** and **icbt** are treated as “loads” from the addressed byte with respect to Debug exceptions. IAC Debug exceptions are associated with the *fetching* of instructions not with the *execution* of instructions. DAC Debug exceptions are associated with the *execution* of instruction cache management instructions, as well as with the execution of load, store, and data cache management instructions.

Data Value Compare (DVC) exception

A DVC Debug exception occurs when execution is attempted of a load, store, or **dcbz** instruction whose target storage operand address matches the DAC and DVC conditions specified by the various debug facility registers. This exception can occur regardless of debug mode, and regardless of the value of MSR[DE].

Branch Taken (BRT) exception

A BRT Debug exception occurs when BRT debug events are enabled (DBCR0[BRT]=1) and execution is attempted of a branch instruction for which the branch conditions are met. This exception cannot occur in internal debug mode when MSR[DE]=0, unless external debug mode or debug wait mode is also enabled.

Trap (TRAP) exception

A TRAP Debug exception occurs when TRAP debug events are enabled (DBCR0[TRAP]=1) and execution is attempted of a **tw** or **twi** instruction that matches any of the specified trap conditions. This exception can occur regardless of debug mode, and regardless of the value of MSR[DE].

Return (RET) exception

A RET Debug exception occurs when RET debug events are enabled (DBCR0[RET]=1) and execution is attempted of an **rfi**, **rfci,** or **rfmci** instruction. For **rfi**, the RET Debug exception can occur regardless of debug mode and regardless of the value of MSR[DE]. For **rfci** or **rfmci**, the RET Debug exception cannot occur in internal debug mode when MSR[DE]=0, unless external debug mode or debug wait mode is also enabled.

Instruction Complete (ICMP) exception

An ICMP Debug exception occurs when ICMP debug events are enabled (DBCR0[ICMP]=1) and execution of any instruction is completed. This exception cannot occur in internal debug mode when MSR[DE]=0, unless external debug mode or debug wait mode is also enabled.

Interrupt (IRPT) exception

An IRPT Debug exception occurs when IRPT debug events are enabled (DBCR0[IRPT]=1) and an interrupt occurs. For non-critical class interrupt types, the IRPT Debug exception can occur regardless of debug mode and regardless of the value of MSR[DE]. For critical class interrupt types, the IRPT Debug exception cannot occur in internal debug mode (regardless of the value of MSR[DE]), unless external debug mode or debug wait mode is also enabled.

Unconditional Debug Event (UDE) exception

A UDE Debug exception occurs when an Unconditional Debug Event is signaled over the JTAG interface to the PPC465. This exception can occur regardless of debug mode, and regardless of the value of MSR[DE].

There are four debug modes supported by the PPC465. They are: internal debug mode, external debug mode, debug wait mode, and trace mode. Debug exceptions and interrupts are affected by the debug mode(s) which are enabled at the time of the Debug exception. Debug interrupts occur only when internal debug mode is enabled, although it is possible for external debug mode and/or debug wait mode to be enabled as well. The remainder of this section assumes that internal debug mode is enabled and that external debug mode and debug wait mode are not enabled, at the time of a Debug exception.

See *Debug Facilities* on page 315 for more information on the different debug modes and the behavior of each of the Debug exception types when operating in each of the modes.

Programming Note: It is a programming error for software to enable internal debug mode (by setting DBCR0[IDM] to 1) while Debug exceptions are already present in the DBSR. Software must first clear all DBSR Debug exception status (that is, all fields except IDE, MRR, IAC12ATS, and IAC34ATS) before setting DBCR0[IDM] to 1.

If a **stwcx.** instruction causes a DAC or DVC Debug exception, but the processor does not have the reservation from a **lwarx** instruction, then the Debug exception is not recorded in the DBSR and a Debug interrupt does not occur. Instead, the instruction completes and updates CR[CR0] to indicate the failure of the store due to the lost reservation.

If a DAC exception occurs on an **lswx** or **stswx** with a length of zero, then the instruction is treated as a no-op, the Debug exception is not recorded in the DBSR, and a Debug interrupt does not occur.

If a DAC exception occurs on an **icbt**, **dcbt**, or **dcbtst** instruction which is being no-op'ed due to some other reason (either the referenced cache block is in a caching inhibited memory page, or a Data Storage or Data TLB Miss exception occurs), then the Debug exception is not recorded in the DBSR and a Debug interrupt does not occur. On the other hand, if the **icbt**, **dcbt**, or **dcbtst** instruction is not being no-op'ed for one of these other reasons, the DAC Debug exception does occur and is handled in the same fashion as other DAC Debug exceptions (see below).

For all other cases, when a Debug exception occurs, it is immediately presented to the interrupt handling mechanism. A Debug interrupt will occur immediately if MSR[DE] is 1, and the interrupt processing registers will be updated as described below. If MSR[DE] is 0, however, then the exception condition remains set in the DBSR. If and when MSR[DE] is subsequently set to 1, and the exception condition is still present in the DBSR, a “delayed” Debug interrupt will then occur either as a synchronous, imprecise interrupt, or as an asynchronous interrupt, depending on the type of Debug exception.

When a Debug interrupt occurs, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged) and instruction execution resumes at address IVPR[IVP] || IVOR15[IVO] || 0b0000.

Critical Save/Restore Register 0 (CSRR0)

For Debug exceptions that occur while Debug interrupts are enabled (MSR[DE] = 1), CSRR0 is set as follows:

- For IAC, BRT, TRAP, and RET Debug exceptions, set to the address of the instruction causing the Debug interrupt. Execution of the instruction causing the Debug exception is suppressed, and the interrupt is synchronous and precise.
- For DAC and DVC Debug exceptions, if DBCR2[DAC12A] is 0, set to the address of the instruction causing the Debug interrupt. Execution of the instruction causing the Debug exception is suppressed, and the interrupt is synchronous and precise.

  If DBCR2[DAC12A] is 1, however, then DAC and DVC Debug exceptions are handled asynchronously, and CSRR0 is set to the address of the instruction that would have executed next had the Debug interrupt not occurred. This could either be the address of the instruction causing the DAC or DVC Debug exception, or the address of a subsequent instruction.
- For ICMP Debug exceptions, set to the address of the next instruction to be executed (the instruction after the one whose completion caused the ICMP Debug exception). The interrupt is synchronous and precise.

  Since the ICMP Debug exception does not suppress the execution of the instruction causing the exception, but rather allows it to complete before causing the interrupt, the behavior of the interrupt is different in the special case where the instruction causing the ICMP Debug exception is itself setting MSR[DE] to 0. In this case, the interrupt will be delayed and will occur if and when MSR[DE] is again set to 1, assuming DBSR[ICMP] is still set. If the Debug interrupt occurs in this fashion, it will be synchronous and imprecise, and CSRR0 will be set to the address of the instruction after the one which set MSR[DE] to 1 (not the one which originally caused the ICMP Debug exception and in so doing set MSR[DE] to 0). If the instruction which set MSR[DE] to 1 was **rfi**, **rfci,** or **rfmci**, then CSRR0 is set to the address to which the **rfi**, **rfci,** or **rfmci** was returning, and not to the address of the instruction which was sequentially after the **rfi**, **rfci,** or **rfmci**.
- For IRPT Debug exceptions, set to the address of the first instruction in the interrupt handler associated with the interrupt type that caused the IRPT Debug exception. The interrupt is asynchronous.
- For UDE Debug exceptions, set to the address of the instruction that would have executed next if the Debug interrupt had not occurred. The interrupt is asynchronous.

For all Debug exceptions that occur while Debug interrupts are disabled (MSR[DE] = 0), the Debug interrupt will be delayed and will occur if and when MSR[DE] is again set to 1, assuming the Debug exception status is still set in the DBSR. If the Debug interrupt occurs in this fashion, CSRR0 is set to the address of the instruction after the one which set MSR[DE]. If the instruction which set MSR[DE] was **rfi**, **rfci,** or **rfmci**, then CSRR0 is set to the address to which the **rfi**, **rfci,** or **rfmci** was returning, and not to the address of the instruction which was sequentially after the **rfi**, **rfci,** or **rfmci**. The interrupt is either synchronous and imprecise, or asynchronous, depending on the type of Debug exception, as follows:

- For IAC and RET Debug exceptions, the interrupt is synchronous and imprecise.

- For BRT Debug exceptions, this scenario cannot occur. BRT Debug exceptions are not recognized when MSR[DE]=0 if operating in internal debug mode.
- For TRAP Debug exceptions, the Debug interrupt is synchronous and imprecise. However, under these conditions (TRAP Debug exception occurring while MSR[DE] is 0), the attempted execution of the trap instruction for which one or more of the trap conditions is met will itself lead to a Trap exception type Program interrupt. The corresponding Debug interrupt which will occur later if and when Debug interrupts are enabled will be *in addition* to the Program interrupt.
- For DAC and DVC Debug exceptions, if DBCR2[DAC12A] is 0, then the interrupt is synchronous and imprecise. If DBCR2[DAC12A] is 1, then the interrupt is asynchronous.
- For ICMP Debug exceptions, this scenario cannot occur in this fashion. ICMP Debug exceptions are not recognized when MSR[DE]=0 if operating in internal debug mode. However, a similar scenario can occur when MSR[DE] is 1 at the time of the ICMP Debug exception, but the instruction whose completion is causing the exception is itself setting MSR[DE] to 0. This scenario is described above in the subsection on the ICMP Debug exception for which MSR[DE] is 1 at the time of the exception. In that scenario, the interrupt is synchronous and imprecise.
- For IRPT and UDE Debug exceptions, the interrupt is asynchronous.

Critical Save/Restore Register 1 (CSRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

ME	Unchanged.

All other MSR bits set to 0.

9.6 Interrupt Ordering and Masking

It is possible for multiple exceptions to exist simultaneously, each of which could cause the generation of an interrupt. Furthermore, the PowerPC Book-E architecture does not provide for the generation of more than one interrupt of the same class (critical or non-critical) at a time. Therefore, the architecture defines that interrupts are ordered with respect to each other, and provides a masking mechanism for certain persistent interrupt types.

When an interrupt type is masked (disabled), and an event causes an exception that would normally generate an interrupt of that type, the exception *persists* as a *status* bit in a register (which register depends upon the exception type). 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 will then finally be generated.

All asynchronous interrupt types can be masked. Machine Check interrupts can be masked, as well. In addition, certain synchronous interrupt types can be masked. The two synchronous interrupt types which can be masked are the Floating-Point Enabled exception type Program interrupt (masked by MSR[FE0,FE1]), and the IAC, DAC, DVC, RET, and ICMP exception type Debug interrupts (masked by MSR[DE]).

Architecture Note: When an otherwise synchronous, *precise* interrupt type is “delayed” in this fashion via masking, and the interrupt type is later enabled, the interrupt that is then generated due to the exception event that occurred while the interrupt type was disabled is then considered a synchronous, *imprecise* class of interrupt.

In order to prevent a subsequent interrupt from causing the state information (saved in SRR0/SRR1, CSRR0/CSRR1, or MCSRR0/MCSRR1) from a previous interrupt to be overwritten and lost, the PPC465 performs certain functions. As a first step, upon any non-critical class interrupt, the processor automatically disables any further asynchronous, non-critical class interrupts (External Input, Decrementer, and Fixed Interval Timer) by clearing MSR[EE]. Likewise, upon any critical class interrupt, hardware automatically disables any further asynchronous interrupts of either class (critical and non-critical) by clearing MSR[CE] and MSR[DE], in addition to MSR[EE]. The additional interrupt types that are disabled by the clearing of MSR[CE,DE] are the Critical Input, Watchdog Timer, and Debug interrupts. For machine check interrupts, the processor automatically disables all maskable interrupts by clearing MSR[ME] as well as MSR[EE,CE,DE].

This first step of clearing MSR[EE] (and MSR[CE,DE] for critical class interrupts, and MSR[ME] for machine checks) prevents any subsequent asynchronous interrupts from overwriting the relevant save/restore registers (SRR0/SRR1, CSRR0/CSRR1, or MCSRR0/MCSRR1), prior to software being able to save their contents. The processor also automatically clears, on any interrupt, MSR[WE,PR,FP,FE0,FE1,IS,DS]. The clearing of these bits assists in the avoidance of subsequent interrupts of certain other types. However, *guaranteeing* that these interrupt types do not occur and thus do not overwrite the save/restore registers also requires the cooperation of system software. Specifically, system software must avoid the execution of instructions that could cause (or enable) a subsequent interrupt, if the contents of the save/restore registers have not yet been saved.

9.6.1 Interrupt Ordering Software Requirements

The following list identifies the actions that system software must *avoid*, prior to having saved the save/restore registers' contents:

- Reenabling of MSR[EE] (or MSR[CE,DE] in critical class interrupt handlers)

  This prevents any asynchronous interrupts, as well as (in the case of MSR[DE]) any Debug interrupts (which include both synchronous and asynchronous types).

- Branching (or sequential execution) to addresses not mapped by the TLB, or mapped without execute access permission

  This prevents Instruction Storage and Instruction TLB Error interrupts.

- Load, store, or cache management instructions to addresses not mapped by the TLB or not having the necessary access permission (read or write)

  This prevents Data Storage and Data TLB Error interrupts.

- Execution of system call (**sc**) or trap (**tw**, **twi**) instructions

  This prevents System Call and Trap exception type Program interrupts.

- Execution of any floating-point instructions

  This prevents Floating-Point Unavailable interrupts. Note that this interrupt would occur upon the execution of any floating-point instruction, due to the automatic clearing of MSR[FP]. However, even if software were to re-enable MSR[FP], floating-point instructions must still be avoided in order to prevent Program interrupts due to the possibility of Floating-Point Enabled and/or Unimplemented Operation exceptions.

- Reenabling of MSR[PR]

  This prevents Privileged Instruction exception type Program interrupts. Alternatively, software could re-enable MSR[PR], but avoid the execution of any privileged instructions.

- Execution of any Auxiliary Processor instructions that are not implemented in the PPC465

  This prevents Auxiliary Processor Unavailable interrupts, as well as Auxiliary Processor Enabled and Unimplemented Operation exception type Program interrupts. Note that the auxiliary processor instructions that are implemented within the PPC465 do not cause any of these types of exceptions, and can therefore be executed prior to software having saved the save/restore registers' contents.

- Execution of any illegal instructions, or any defined instructions not implemented within the PPC465 (64-bit instructions, **tlbiva**, **mfapidi**)

  This prevents Illegal Instruction exception type Program interrupts.
- Execution of any instruction that could cause an Alignment interrupt

  This prevents Alignment interrupts. See “Alignment Interrupt” on page 269 for a complete list of instructions that may cause Alignment interrupts.
- In the Machine Check handler, use of the caches and TLBs until any detected parity errors have been corrected.

  This will avoid additional parity errors.

It is not necessary for hardware or software to avoid critical class interrupts from within non-critical class interrupt handlers (and hence the processor does not automatically clear MSR[CE,ME,DE] upon a non-critical interrupt), since the two classes of interrupts use different pairs of save/restore registers to save the instruction address and MSR. The converse, however, is not true. That is, hardware and software must cooperate in the avoidance of both critical *and* non-critical class interrupts from within critical class interrupt handlers, even though the two classes of interrupts use different save/restore register pairs. This is because the critical class interrupt may have occurred from within a non-critical class interrupt handler, prior to the non-critical class interrupt handler having saved SRR0 and SRR1. Therefore, within the critical class interrupt handler, both pairs of save/restore registers may contain data that is necessary to the system software.

Similarly, the Machine Check handler must avoid further machine checks, as well as both critical and non-critical interrupts, since the machine check handler may have been called from within a critical or non-critical interrupt handler.

9.6.2 Interrupt Order

The following is a prioritized listing of the various enabled interrupt types for which exceptions might exist simultaneously:

1. Synchronous (non-debug) interrupts:
   1. Data Storage
   2. Instruction Storage
   3. Alignment
   4. Program
   5. Floating-Point Unavailable
   6. System Call
   7. Auxiliary Processor Unavailable
   8. Data TLB Error
   9. Instruction TLB Error

   Only one of the above types of synchronous interrupts may have an existing exception generating it at any given time. This is guaranteed by the exception priority mechanism (see “Exception Priorities” on page 284) and the requirements of the sequential execution model defined by the PowerPC Book-E architecture.
2. Machine Check
3. Debug
4. Critical Input

5. Watchdog Timer
6. External Input
7. Fixed-Interval Timer
8. Decrementer

Even though, as indicated above, the non-critical, synchronous exception types listed under item 1 are generated with higher priority than the critical interrupt types listed in items 2-5, the fact is that these non-critical interrupts will immediately be followed by the highest priority existing critical interrupt type, without executing any instructions at the non-critical interrupt handler. This is because the non-critical interrupt types do not automatically clear MSR[ME,DE,CE] and hence do not automatically disable the critical interrupt types. In all other cases, a particular interrupt type from the above list will automatically disable any subsequent interrupts of the same type, as well as all other interrupt types that are listed below it in the priority order.

9.7 Exception Priorities

PowerPC Book-E requires all synchronous (precise and imprecise) interrupts to be reported in program order, as implied by the sequential execution model. The one exception to this rule is the case of multiple synchronous imprecise interrupts. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions, and the interrupt(s) will then be generated according to the general interrupt ordering rules outlined in “Interrupt Order” on page 283. For example, if a **mtmsr** instruction causes MSR[FE0,FE1,DE] to all be set, it is possible that a previous Floating-Point Enabled exception and a previous Debug exception both are still being presented (in the FPSCR and DBSR, respectively). In such a scenario, a Floating-Point Enabled exception type Program interrupt will occur first, followed immediately by a Debug interrupt.

For any single instruction attempting to cause multiple exceptions for which the corresponding synchronous interrupt types are enabled, this section defines the priority order by which the instruction will be permitted to cause a *single* enabled exception, thus generating a particular synchronous interrupt. Note that it is this exception priority mechanism, along with the requirement that synchronous interrupts be generated in program order, that guarantees that at any given time there exists for consideration only one of the synchronous interrupt types listed in item 1 of “Interrupt Order” on page 283. The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

This section does not define the permitted setting of multiple exceptions for which the corresponding interrupt types are disabled. The generation of exceptions for which the corresponding interrupt types are disabled will have no effect on the generation of other exceptions for which the corresponding interrupt types are enabled. Conversely, if a particular exception for which the corresponding interrupt type is enabled is shown in the following sections to be of a higher priority than another exception, the occurrence of that enabled higher priority exception will prevent the setting of the other exception, independent of whether that other exception’s corresponding interrupt type is enabled or disabled.

Except as specifically noted below, only one of the exception types listed for a given instruction type will be permitted to be generated at any given time, assuming the corresponding interrupt type is enabled. The priority of the exception types are listed in the following sections ranging from highest to lowest, within each instruction type.

Finally, note that Machine Check exceptions are defined by the PowerPC architecture to be neither synchronous nor asynchronous. As such, Machine Check exceptions are not considered in the remainder of this section, which is specifically addressing the priority of synchronous interrupts.

9.7.1 Exception Priorities for Integer Load, Store, and Cache Management Instructions

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of any integer load, store, or cache management instruction. Included in this category is the former opcode for the **icbt** instruction, which is an allocated opcode still supported by the PPC465.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Program (Illegal Instruction exception)

   Only applies to the defined 64-bit load, store, and cache management instructions, which are not recognized by the PPC465.
5. Program (Privileged Instruction)

   Only applies to the **dcbi** instruction, and only occurs if MSR[PR]=1.
6. Data TLB Error (Data TLB Miss exception)
7. Data Storage (all exception types except Byte Ordering exception)
8. Alignment (Alignment exception)
9. Debug (DAC or DVC exception)
10. Debug (ICMP exception)

9.7.2 Exception Priorities for Floating-Point Load and Store Instructions

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of any floating-point load or store instruction.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Program (Illegal Instruction exception)

   This exception will occur if no floating-point unit is attached to the PPC465, or if the particular floating-point load or store instruction is not recognized by the attached floating-point unit.
5. Floating-Point Unavailable (Floating-Point Unavailable exception)

   This exception will occur if an attached floating-point unit recognizes the instruction, but floating-point instruction processing is disabled (MSR[FP]=0).
6. Program (Unimplemented Operation exception)

   This exception will occur if an attached floating-point unit recognizes but does not support the instruction, and floating-point instruction processing is enabled (MSR[FP]=1).
7. Data TLB Error (Data TLB Miss exception)
8. Data Storage (all exception types except Cache Locking exception)
9. Alignment (Alignment exception)
10. Debug (DAC or DVC exception)
11. Debug (ICMP exception)

9.7.3 Exception Priorities for Allocated Load and Store Instructions

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of any allocated load or store instruction.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Program (Illegal Instruction exception)

   This exception will occur if no auxiliary processor unit is attached to the PPC465, or if the particular allocated load or store instruction is not recognized by the attached auxiliary processor.
5. Program (Privileged Instruction exception)

   This exception will occur if an attached auxiliary processor unit recognizes the instruction and indicates that the instruction is privileged, but MSR[PR]=1.
6. Auxiliary Processor Unavailable (Auxiliary Processor Unavailable exception)

   This exception will occur if an attached auxiliary processor recognizes the instruction, but indicates that auxiliary processor instruction processing is disabled (whether or not auxiliary processor instruction processing is enabled is implementation-dependent).
7. Program (Unimplemented Operation exception)

   This exception will occur if an attached auxiliary processor recognizes but does not support the instruction, and also indicates that auxiliary processor instruction processing is enabled (whether or not auxiliary processor instruction processing is enabled is implementation-dependent).
8. Data TLB Error (Data TLB Miss exception)
9. Data Storage (all exception types except Cache Locking exception)
10. Alignment (Alignment exception)
11. Debug (DAC or DVC exception)
12. Debug (ICMP exception)

9.7.4 Exception Priorities for Floating-Point Instructions (Other)

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of any floating-point instruction other than a load or store.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Program (Illegal Instruction exception)

   This exception will occur if no floating-point unit is attached to the PPC465, or if the particular floating-point instruction is not recognized by the attached floating-point unit.
5. Floating-Point Unavailable (Floating-Point Unavailable exception)

   This exception will occur if an attached floating-point unit recognizes the instruction, but floating-point instruction processing is disabled (MSR[FP]=0).
6. Program (Unimplemented Operation exception)

This exception will occur if an attached floating-point unit recognizes but does not support the instruction, and floating-point instruction processing is enabled (MSR[FP]=1).

7. Program (Floating-Point Enabled exception)

   This exception will occur if an attached floating-point unit recognizes and supports the instruction, floating-point instruction processing is enabled (MSR[FP]=1), and the instruction sets FPSCR[FEX] to 1.

8. Debug (ICMP exception)

9.7.5 Exception Priorities for Allocated Instructions (Other)

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of any allocated instruction other than a load or store, and which is not one of the allocated instructions implemented within the PPC465.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Program (Illegal Instruction exception)

   This exception will occur if no auxiliary processor unit is attached to the PPC465, or if the particular allocated instruction is not recognized by the attached auxiliary processor and is not one of the allocated instructions implemented within the PPC465.

5. Program (Privileged Instruction exception)

   This exception will occur if an attached auxiliary processor unit recognizes the instruction and indicates that the instruction is privileged, but MSR[PR]=1.

6. Auxiliary Processor Unavailable (Auxiliary Processor Unavailable exception)

   This exception will occur if an attached auxiliary processor recognizes the instruction, but indicates that auxiliary processor instruction processing is disabled (whether or not auxiliary processor instruction processing is enabled is implementation-dependent).

7. Program (Unimplemented Operation exception)

   This exception will occur if an attached auxiliary processor recognizes but does not support the instruction, and also indicates that auxiliary processor instruction processing is enabled (whether or not auxiliary processor instruction processing is enabled is implementation-dependent).

8. Program (Auxiliary Processor Enabled exception)

   This exception will occur if an attached auxiliary processor recognizes and supports the instruction, indicates that auxiliary processor instruction processing is enabled, and the instruction execution results in an Auxiliary Processor Enabled exception. Whether or not auxiliary processor instruction processing is enabled is implementation-dependent, as is whether or not a given auxiliary processor instruction results in an Auxiliary Processor Enabled exception.

9. Debug (ICMP exception)

9.7.6 Exception Priorities for Privileged Instructions

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of any privileged instruction other than **dcbi**, **rfi**, **rfci**, **rfmci**, or any allocated instruction not implemented within the PPC465 (all of which are covered elsewhere). This list *does* cover, however, the **dccci**,

dcread, **iccci**, and **icread** instructions, which are privileged, allocated instructions that *are* implemented within the PPC465. This list also covers the defined 64-bit privileged instructions, the **tlbiva** instruction, and the **mfapidi** instruction, all of which are not implemented by the PPC465.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Program (Illegal Instruction exception)

   Only applies to the defined 64-bit privileged instructions, the **tlbiva** instruction, and the **mfapidi** instruction.
5. Program (Privileged Instruction exception)

   Does not apply to the defined 64-bit privileged instructions, the **tlbiva** instruction, nor the **mfapidi** instruction.
6. Debug (ICMP exception)

   Does not apply to the defined 64-bit privileged instructions, the **tlbiva** instruction, nor the **mfapidi** instruction.

9.7.7 Exception Priorities for Trap Instructions

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of a trap (**tw**, **twi**) instruction.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Debug (TRAP exception)
5. Program (TRAP exception)
6. Debug (ICMP exception)

9.7.8 Exception Priorities for System Call Instruction

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of a system call (**sc**) instruction.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. System Call (System Call exception)
5. Debug (ICMP exception)

Since the System Call exception does not suppress the execution of the **sc** instruction, but rather the exception occurs once the instruction has completed, it is possible for an **sc** instruction to cause both a System Call exception and an ICMP Debug exception at the same time. In such a case, the associated interrupts will occur in the order indicated in “Interrupt Order” on page 283.

9.7.9 Exception Priorities for Branch Instructions

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of a branch instruction.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Debug (BRT exception)
5. Debug (ICMP exception)

9.7.10 Exception Priorities for Return From Interrupt Instructions

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of an **rfi**, **rfci,** or **rfmci** instruction.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Debug (RET exception)
5. Debug (ICMP exception)

9.7.11 Exception Priorities for Preserved Instructions

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of a preserved instruction.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Program (Illegal Instruction exception)

   Applies to all preserved instructions except the **mftb** instruction, which is the only preserved class instruction implemented within the PPC465.
5. Debug (ICMP exception)

   Only applies to the **mftb** instruction, which is the only preserved class instruction implemented within the PPC465.

9.7.12 Exception Priorities for Reserved Instructions

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of a reserved instruction.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Program (Illegal Instruction exception)

   Applies to all reserved instruction opcodes except the reserved-nop instruction opcodes.
5. Debug (ICMP exception)

   Only applies to the reserved-nop instruction opcodes.

9.7.13 Exception Priorities for All Other Instructions

The following list identifies the priority order of the exception types that may occur within the PPC465 as the result of the attempted execution of all other instructions (that is, those not covered by one of the sections 9.7.1 through 9.7.12). This includes both defined instructions and allocated instructions implemented within the PPC465.

1. Debug (IAC exception)
2. Instruction TLB Error (Instruction TLB Miss exception)
3. Instruction Storage (Execute Access Control exception)
4. Program (Illegal Instruction exception)

   Applies only to the defined 64-bit instructions, as these are not implemented within the PPC465.
5. Debug (ICMP exception)

   Does not apply to the defined 64-bit instructions, as these are not implemented by the PPC465.

9.8 L2 Cache Machine Checks

The L2C generates an asynchronous machine check exception when it encounters a hardware error either internally or on the PLB. The different types of machine checks are:

- Instruction side access tag array single bit ECC error
- Instruction side access tag array double bit ECC error
- Instruction side access data array single bit ECC error
- Instruction side access data array double bit ECC error
- Data side access tag array single bit ECC error
- Data side access tag array double bit ECC error
- Data side access data array single bit ECC error
- Data side access data array double bit ECC error
- Snoop access tag array single bit ECC error
- Snoop access tag array double bit ECC error
- Snoop access data array single bit ECC error
- Snoop access data array double bit ECC error
- Slave access data array single bit ECC error
- Slave access data array double bit ECC error
- L2 Performance monitor exception
- PLB request error
- PLB read error
- PLB Snooper port address bus parity error
- PLB Master port read data bus parity error
- PLB Slave Port request that starts inside but ends outside of the FAR
- PLB Slave Port request address parity error
- PLB Slave Port byte enable parity error
- PLB Slave Port write data parity error
- PLB Slave Port write data error

Upon detecting the machine check the L2C will update the L2MCSR (see *L2 Machine Check Status Register (L2MCSR)* on page 197). The L2MCRER provides a capability to control/enable corresponding errors reporting, and their assignments are defined in *L2 Machine Check Reporting Enable Register (L2MCRER)* on page 196. If L2MCRER are enabled, the L2C reports the machine check to the L1cache, refer to See “Interrupts and Exceptions” on page 247. The error stays asserted until the L2MCSR is cleared by software.

10. Floating Point Unit Interrupts and Exceptions

An *interrupt* is the action in which the processor saves its old context (Machine State Register (MSR) and next instruction address NIA)) and begins execution at a pre-determined interrupt-handler address, with a modified MSR. *Exceptions* are the events that may cause the processor to take an interrupt, if the corresponding interrupt type is enabled.

Exceptions may be generated by the execution of instructions, or by signals from devices external to the PPC465-S processor, the internal timer facilities, debug events, or error conditions.

10.1 Floating-Point Exceptions

Book-E requires all synchronous (precise and imprecise) interrupts to be reported in program order, as required by the sequential execution model. The only exception to this rule is the case of multiple synchronous imprecise interrupts. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions, and the interrupt is then generated with all of those exception types reported cumulatively, in both the Exception Syndrome Register (ESR), and any status registers associated with the particular exception type, such as the Floating-Point Status and Control Register (FPSCR).

For any single instruction attempting to cause multiple exceptions for which the corresponding synchronous interrupt types are enabled, this section defines the priority order by which the instruction will be permitted to cause a *single* enabled exception, thus generating a particular synchronous interrupt. This exception priority mechanism, along with the requirement that synchronous interrupts must be generated in program order, guarantees that only one of the synchronous interrupt types is considered at any given time. The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

This section does not define the permitted setting of multiple exceptions for which the corresponding interrupt types are disabled. The generation of exceptions for which the corresponding interrupt types are disabled has no effect on the generation of other exceptions for which the corresponding interrupt types are enabled. Conversely, if a particular exception for which the corresponding interrupt type is enabled is shown in the following sections to be of a higher priority than another exception, it will prevent the setting of that other exception, regardless of whether the corresponding interrupt type of the other exception is enabled or disabled.

Except as noted, only one of the exception types listed for a given instruction type can be generated at any given time. The priority of the exception types are listed in subsequent sections ranging from highest to lowest, within each instruction type.

Note: Some exception types may be mutually exclusive of each other and could otherwise be considered the same priority. In such cases, the exceptions are listed in the order suggested by the sequential execution model.

Computational instructions may cause exceptions. Aside from instructions that write the FPSCR, none of the noncomputational instructions can cause a floating-point exception.

All exceptions are handled precisely. Because this can affect performance adversely, it is strongly recommended that exceptions should be disabled when possible. This prevents the PPC465-S FPU instruction stream from waiting for the execution of long latency instructions, such as **fdiv**[**s**].

10.2 Exceptions List

Book-E defines the following floating-point exceptions:

Table 10-1. Invalid Operation Exception Categories

Category	FPSCR Field
SNaN	VXSNAN
Infinity – Infinity	VXISI
Infinity ÷ Infinity	VXIDI
Zero ÷ Zero	VXZDZ
Infinity × Zero	VXIMZ
Invalid Compare	VXVC
Software Request	VXSOFT
Invalid Square Root	VXSQRT
Invalid Integer Convert	VXCVI

- Invalid Operation exception (VX)
- Zero Divide exception (ZX)
- Overflow exception (OX)
- Underflow exception (UX)
- Inexact exception (XI)

These exceptions can occur during execution of computational instructions. In addition, an Invalid Operation exception occurs when a **mtsfs** or **mtsfsi** instruction sets FPSCR[VXSOFT] = 1.

Each floating-point exception, and each category of Invalid Operation exception, has an exception bit in the FPSCR. Each floating-point exception also has a corresponding enable bit in the FPSCR. The exception bit indicates the occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit controls the result produced by the instruction and, with MSR[FE0, FE1] whether and how the Enabled exception type Program interrupt is taken. (See *Floating-Point Exceptions* on page 293 for more information.) In general, the enabling specified by an enable bit is to enable the invoking the interrupt, not to enable the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any enable bits. The only exceptions to this general rule are the occurrence of an Underflow exception, which may depend on the setting of the enable bit, and the occurrence of a Inexact exception, which may depend on the Overflow exception bit not being set.

A single instruction, other than **mtfsf** or **mtfsfi**, can set more than one exception bit only in the following cases:

- An Inexact exception may be set with an Overflow exception.
- An Inexact Exception may be set with an Underflow exception.
- An Invalid Operation exception (SNaN) is set with Invalid Operation exception ($\infty \times 0$) for *Multiply-Add* instructions for which the values being multiplied are infinity and 0, and the value being added is an SNaN.
- An Invalid Operation exception (SNaN) can be set with Invalid Operation exception (Invalid Compare) for *Compare Ordered* instructions.

- Invalid Operation exception (SNaN) can be set with Invalid Operation exception (Invalid Integer Convert) for *Convert To Integer* instructions.

When an exception occurs, instruction execution may be suppressed or a result may be delivered, depending on the exception.

Instruction execution is suppressed for the following kinds of exception, so that there is no possibility that one of the operands is lost:

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining exceptions, a result is generated and written to the target specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The exceptions that deliver a result are:

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

IEEE 754 specifies the handling of exceptional conditions in terms of “traps” and “trap handlers.” In Book-E, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the “trap enabled” case. The exception is expected to be detected by software, which revises the result. An FPSCR exception enable bit of 0 causes generation of the “default result” value specified for the “trap disabled” (or “no trap occurs” or “trap is not implemented”) case. Software is not expected to detect the exception, and simply uses the default result. The result to be delivered in each case for each exception is described in subsequent sections.

The IEEE 754 default behavior when an exception occurs is to generate a default value and to not notify software. In Book-E, if the IEEE 754 default behavior is desired for all exceptions, all FPSCR exception enable bits should be set to 0 and Ignore Exceptions Mode should be used (see *Table 10-2* on page 296). In this case, an Enabled exception type Program interrupt is not taken, even if floating-point exceptions occur. Software can inspect the FPSCR exception bits, if necessary, to determine whether exceptions have occurred.

If software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1 and a mode other than Ignore Exceptions Mode must be used. In this case, the Enabled exception type Program interrupt is taken if an enabled floating-point exception occurs. An Enabled exception type Program interrupt is also taken if an **mtsfs** or **mtsfsi** instruction sets an exception bit and its corresponding enable bit both to 1; the **mtsfs** or **mtsfsi** instruction is considered to cause the enabled exception.

MSR[FE0, FE1] control whether and how Enabled exception type Program interrupt are taken when an enabled floating-point exception occurs. An Enabled exception type Program interrupt is never taken because of a disabled floating-point exception.

Table 10-2. MSR[FE0, FE1] Modes

MSR[FE0]	MSR[FE1]	Mode
0	0	**Ignore Exceptions Mode** Floating-point exceptions do not cause an Enabled exception type Program interrupt to be taken.
1	1	**Precise Mode** An Enabled exception type Program interrupt is taken precisely at the instruction that caused the enabled exception.

If either MSR[FE0] or MSR[FE1] is 1, Enabled exception type Program interrupts are treated as in Precise Mode.

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of MSR[FE0, FE1].

In all cases in which an Enabled exception type Program interrupt is taken, all instructions before the instruction at which the Enabled exception type Program interrupt is taken have completed, and no instruction after the instruction at which the Enabled exception type Program interrupt is taken has begun execution. The instruction at which the Enabled exception type Program interrupt is taken has not been executed unless it is the excepting instruction, in which case it has been executed if the exception is not an Enabled Invalid Operation exception or Enabled Zero Divide exception.

A **sync** instruction, or any other execution-synchronizing instruction or event, such as **isync**, also has the effects described above.

In order to obtain the best performance across the widest range of implementations, the programmer should follow these guidelines.

- If the IEEE 754 default results are acceptable to the application, Ignore Exceptions Mode should be used, with all FPSCR exception enable bits set to 0.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

10.3 Floating-Point Interrupts

The following interrupts are taken under the control of the PPC465-S processor, and are not enabled by or reported in FPSCR bits:

- Floating-Point Unavailable
- Floating-Point Assist

10.3.1 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point loads, stores, and moves), and MSR[FP] = 0.

When a Floating-Point Unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the Floating-Point Unavailable interrupt.

10.4 Floating-Point Exception Behavior

The following sections describe the behavior that results from the floating-point exceptions. For each exception, the definition of the exception is given, followed by a description of the action caused by the exception.

In general, each exception can result in either of two types of action, depending on whether the exception is enabled by its associated exception enable bit in the FPSCR.

10.4.1 Invalid Operation Exception

An Invalid Operation exception occurs when an operand is invalid for the specified operation. The invalid operations are:

- Any floating-point operation on a signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities ($\infty - \infty$)
- Division of infinity by infinity ($\infty \div \infty$)
- Division of zero by zero ($0 \div 0$)
- Multiplication of infinity by zero ($\infty \times 0$)
- Ordered comparison involving a NaN (Invalid Compare)
- Square root or reciprocal square root of a negative and nonzero number (Invalid Square Root)
- Integer conversion involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN (Invalid Integer Convert)

In addition, an Invalid Operation exception occurs if software explicitly requests this by executing an **mtfsf**, **mtfsfi**, or **mtfsb1** instruction that sets FPSCR[VXSOFT] = 1.

Programming Note: The purpose of FPSCR[VXSOFT] is to enable software to cause an Invalid Operation exception for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it may be set by a program that computes a square root, if the source operand is negative.

10.4.1.1 Action

The action taken depends on the setting of FPSCR[VE].

When Invalid Operation exception is enabled (FPSCR[VE] = 1) and an Invalid Operation exception occurs or software explicitly requests the exception, the following actions are taken:

- One or two FPSCR Invalid Operation exception bits, listed in *Table 10-3*, are set.

Table 10-3. Invalid Operation Exceptions

FPSCR Bit	Category
VXSNAN	SNaN
VXISI	Infinity – Infinity
VXIDI	Infinity ÷ Infinity
VXZDZ	Zero ÷ Zero
VXIMZ	Infinity × Zero
VXVC	Invalid Compare
VXSOFT	Software Request
VXSQRT	Invalid Square Root
VXCVI	Invalid Integer Convert

- If the operation is an arithmetic, **frsp**, or convert to integer operation, the target FPR is unchanged.
  - FPSCR[FR, FI] ← 0
  - FPSCR[FPRF] ← unchanged
- If the operation is a compare:
  - FPSCR[FR, FI, C] ← unchanged
  - FPSCR[FPCC] ← unordered
- If software explicitly requests the exception:

  FPSCR[FR, FI, FPRF] are as set by the **mtfsf**, **mtfsfi**, or **mtfsb1** instruction.

When Invalid Operation exception is disabled (FPSCR[VE] = 0) and an Invalid Operation exception occurs, or software explicitly requests the exception, the following actions are taken:

- One or two FPSCR Invalid Operation exception bits, listed in *Table 10-3*, are set.
- If the operation is an arithmetic or *Floating Round to Single-Precision* operation, the target FPR is set to a Quiet NaN
  - FPSCR[FR, FI] ← 0
  - FPSCR[FPRF] ← the class of the result (Quiet NaN)
- If the operation is a convert to 32-bit integer operation, the target FPR is set as follows:
  - FPR(FRT)$_{0:31}$ ← undefined
  - FPR(FRT)$_{32:63}$ are set to the most positive 32-bit integer if the operand in FPR(FRB) is a positive number or +∞, and to the most negative 32-bit integer if the operand in FPR(FRB) is a negative number, -∞, or NaN.
  - FPSCR[FR, FI] ← 0
  - FPSCR[FPRF] ← undefined
- If the operation is a compare:
  - FPSCR[FR, FI, C] ← unchanged
  - FPSCR[FPCC] ← unordered

- If software explicitly requests the exception:

  FPSCR[FR, FI, FPRF] are as set by the **mtfsf**, **mtfsfi**, or **mtfsb1** instruction.

10.4.2 Zero Divide Exception

A Zero Divide exception occurs when an **fdiv[s]** instruction is executed with a zero divisor value and a finite nonzero dividend value. This exception also occurs when a *Reciprocal Estimate* instruction (**fres** or **frsqrte**) is executed with an operand value of zero.

10.4.2.1 Action

The action to be taken depends on the setting of FPSCR[ZE].

When Zero Divide exception is enabled (FPSCR[ZE] = 1) and Zero Divide occurs, the following actions are taken:

- The Zero Divide exception bit is set.

  $FPSCR_{ZX} \leftarrow 1$
- $FPR(FRT)_{0:31} \leftarrow$ unchanged
- FPSCR[FR, FI] ← 0
- FPSCR[FPRF] ← unchanged

When Zero Divide exception is disabled (FPSCR[ZE] = 0) and zero divide occurs, the following actions are taken:

- The Zero Divide exception bit is set.

  $FPSCR_{ZX} \leftarrow 1$
- FPR(FRT) ← ±Infinity (the sign is determined by the XOR of the signs of the operands)
- FPSCR[FR, FI] ← 0
- FPSCR[FPRF] ← class and sign of the result (± Infinity)

10.4.3 Overflow Exception

Overflow occurs when the magnitude of what would have been the rounded result, if the exponent range were unbounded, exceeds that of the largest finite number of the specified result precision.

10.4.3.1 Action

The action to be taken depends on the setting of FPSCR[OE].

When Overflow exceptions re enabled (FPSCR[OE] = 1) and exponent overflow occurs, the following actions are taken:

- Overflow Exception is set

  FPSCR[OX] ← 1
- For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536.
- For single-precision arithmetic instructions and the **frsp** instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192.
- FPR(FRT) ← adjusted rounded result

- FPSCR[FPRF] ← class and sign of the result (± Normal Number)

When Overflow Exception is disabled (FPSCR[OE] = 0) and overflow occurs, the following actions are taken:

- Overflow Exception is set

  FPSCR[OX] ← 1
- Inexact Exception is set

  FPSCR[XX] ← 1
- The result is determined by the rounding mode (FPSCR[RN]) and the sign of the intermediate result as follows:
  - Round to Nearest
    Store ± Infinity, where the sign is the sign of the intermediate result
  - Round toward Zero
    Store the format's largest finite number with the sign of the intermediate result
  - Round toward +Infinity
    For negative overflow, store the format's most negative finite number; for positive overflow, store +Infinity
  - Round toward –Infinity
    For negative overflow, store – Infinity; for positive overflow, store the largest finite number of the format
- FPR(FRT) ← result
- FPSCR[FR] ← undefined
- FPSCR[FI] ← 1
- FPSCR[FPRF] ← class and sign of the result (± Infinity or ± Normal Number)

10.4.4 Underflow Exception

Underflow Exception is defined separately for the enabled and disabled states:

- Enabled:
  Underflow occurs when the intermediate result is “Tiny.”
- Disabled:
  Underflow occurs when the intermediate result is “Tiny” and there is “Loss of Accuracy.”

A “Tiny” result is detected before rounding, when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is “Tiny” and Underflow Exception is disabled (FPSCR[UE] = 0), the intermediate result is denormalized (See *Normalization and Denormalization* on page 94) and rounded (See *Rounding* on page 95) before being placed into the target FPR.

“Loss of Accuracy” is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

10.4.4.1 Action

The action to be taken depends on the setting of FPSCR[UE].

When Underflow exception is enabled (FPSCR[UE] = 1) and exponent underflow occurs, the following actions are taken:

- Underflow Exception is set
  FPSCR[UX] ← 1
- For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536
- For single-precision arithmetic instructions and the **frsp** instruction, the exponent of the normalized intermediate result is adjusted by adding 192
- The adjusted rounded result is placed into the target FPR

  FPSCR[FPRF] ← class and sign of the result (± Normalized Number)

  Programming Note: The FR and FI bits are provided to allow the Enabled exception type Program interrupt, when taken because of an Underflow Exception, to simulate a "trap disabled" environment. That is, the FR and FI bits allow the Enabled exception type Program interrupt to unround the result, thus allowing the result to be denormalized.

When Underflow Exception is disabled (FPSCR[UE] = 0) and underflow occurs, the following actions are taken:

- Underflow Exception is set

  FPSCR[UX] ← 1
- FPR(FRT) ← rounded result
- FPSCR[FPRF] ← class and sign of the result (± Normalized Number, ± Denormalized Number, or ± Zero)

10.4.5 Inexact Exception

An Inexact Exception occurs when either of the following conditions occur during rounding:

- The rounded result differs from the intermediate result, assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case, the result is said to be inexact. If the rounding causes an enabled Overflow Exception or an enabled Underflow Exception, an Inexact Exception also occurs only if the significands of the rounded result and the intermediate result differ.)
- The rounded result overflows and Overflow Exception is disabled.

10.4.5.1 Action

The action to be taken does not depend on the setting of FPSCR[XX].

When Inexact Exception occurs, the following actions are taken:

- Inexact Exception is set

  FPSCR[XX] ← 1
- FPR(FRT) ← rounded or overflowed result
- FPSCR[FPRF] ← class and sign of the result

  Programming Note: In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.

10.5 Exception Priorities for Floating-Point Load and Store Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any *Floating-Point Load* and *Store* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error (all types)
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Floating-Point Unavailable
6. Program (Unimplemented Operation)
7. Data TLB Error (all types)
8. Data Storage (all types)
9. Alignment
10. Debug (Data Address Compare, Data Value Compare)
11. Debug (Instruction Complete)

If an instruction causes both a Debug (Instruction Address Compare) exception, and a Debug (Data Address Compare) or Debug (Data Value Compare) exception, and does not cause any exception listed in items 2–9, both exceptions can be generated and recorded in the Debug Status Register (DBSR). A single Debug interrupt results.

10.6 Exception Priorities for other Floating-Point Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any floating-point instruction other than a load or store.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error (all types)
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Floating-Point Unavailable
6. Program (Unimplemented Operation)
7. Program (Enabled)
8. Debug (Instruction Complete)

10.7 QNaN

If any of the source operands is a NaN, either signaling (SNaN) or quiet (QNaN), the result will be that NaN with the high-order fraction bit forced to 1 (that is, forced to a QNaN). The precedence, in decreasing order, is FRA, FRB, FRC. The resultant QNaN is only truncated on an **frsp[.]** instruction, in which case the most significant 35 bits are copied to the target, with the least significant 29 forced to zero.

Table 10-4. QNaN Result

R_a	R_b	R_c	Resultant QNaN[a]
NaN	X	X	R_a
—	—	X	R_b[b]
—	—	NaN	R_c

a. High-order fraction bit is forced to a 1
b. **frsp**: Result is $(FRB)_{0:11}$ || 1 || $(FRB)_{13:34}$|| $^{29}0$?

10.8 Updating FPRs on Exceptions

The target FPR is never updated on enabled invalid exceptions and enabled divide by zero exceptions. This requirement exists because an instruction may potentially use one of the source registers as a target register, yet it is necessary that the trap handler be able to examine and act upon the source operands.

In all other cases, a floating-point exception does not block the writing of the target FPR.

10.9 Floating-Point Status and Control Register

The computational instructions modify the FPSCR. With the exception of instructions which write directly to the FPSCR, none of the noncomputational instructions modify the FPSCR.

The FPSCR controls the handling of floating-point exceptions and records status resulting from the floating-point operations. $FPSCR_{0:23}$ are status bits. $FPSCR_{24:31}$ are control bits.

The exception bits in the FPSCR (bits 3:12, 21:23) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an **mcrfs**, **mtfsfi**, **mtfsf**, or **mtfsb0** instruction. The exception summary bits FPSCR[FX, FEX, VX] are not considered as “exception bits,” and only FPSCR[FX] is sticky.

FPSCR[FEX, VX] are simply ORs of other FPSCR bits. Therefore, these bits are not listed among the FPSCR bits affected by the various instructions.

FPSCR[FPRF], which contains five result flag bits, is set for arithmetic, rounding, and conversion instructions based on the class of the result value placed into the target FPR. If any portion of a result is undefined, the value placed into FPSCR[FPRF] is undefined. *Table 10-5* describes how the values of the result flags in FPSCR[FPRF] correspond to the result value classes.

Table 10-5. FPSCR[FPRF] Result Flags

Result Flags					Result Value Class
C	<	>	=	?	
1	0	0	0	1	Quiet NaN
0	1	0	0	0	–Infinity
0	1	0	0	0	–Normalized Number
1	1	0	1	0	–Denormalized Number
1	0	0	1	0	–Zero
0	0	0	0	0	+Zero
1	0	1	0	0	+Denormalized Number
0	0	1	0	0	+Normalized Number
0	0	1	0	1	+Infinity

Figure 4-2 on page 87 illustrates the FPSCR.

10.10 Updating the CR

Architecturally, excepting floating-point instructions do not block the updating of the CR in the PPC465-S processor. However, the PPC465-S FPU blocks CR updates and requires software assistance to make them.

10.10.1 CR Fields

The CR fields are modified by various floating-point instructions.

Figure 10-1. 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	

10.10.2 Updating CR Fields

The floating-point compare instructions **fcmpo** and **fcmpu** specify a CR field that is updated with the compare results. update a field (specified by the instruction) of the CR.

Table 10-6 illustrates the bit encodings for a CR field containing the results of an **fcmpo** and **fcmpu** instruction.

Table 10-6. Bit Encodings for a CR Field

CR Field (Bit)	Description
0	*Floating-Point Less Than* (FL) Floating-point compare: (FRA) < (FRB)
1	*Floating-Point Greater Than* (FG) Floating-point compare: (FRA) > (FRB)
2	*Floating-Point Equal* (FE) Floating-point compare: (FRA) = (FRB)
3	*Floating-Point Unordered* (FU) Floating-point compare: One or both of (FRA) or (FRB) is a NaN.

The **mcrfs** instruction moves a specified FPSCR field into a CR field.

10.10.3 Generation of QNaN Results

If a disabled Invalid Operation exception is caused by operating on a NaN, the value returned follows the rules indicated in *Table 10-4* on page 303.

If the exception was not caused by operating on a NaN, a QNaN must be generated. The generated QNaN has a sign bit of 0, an exponent of all 1s, a high-order fraction bit of 1 with all other fraction bits of 0: 0x7FF8000000000000.

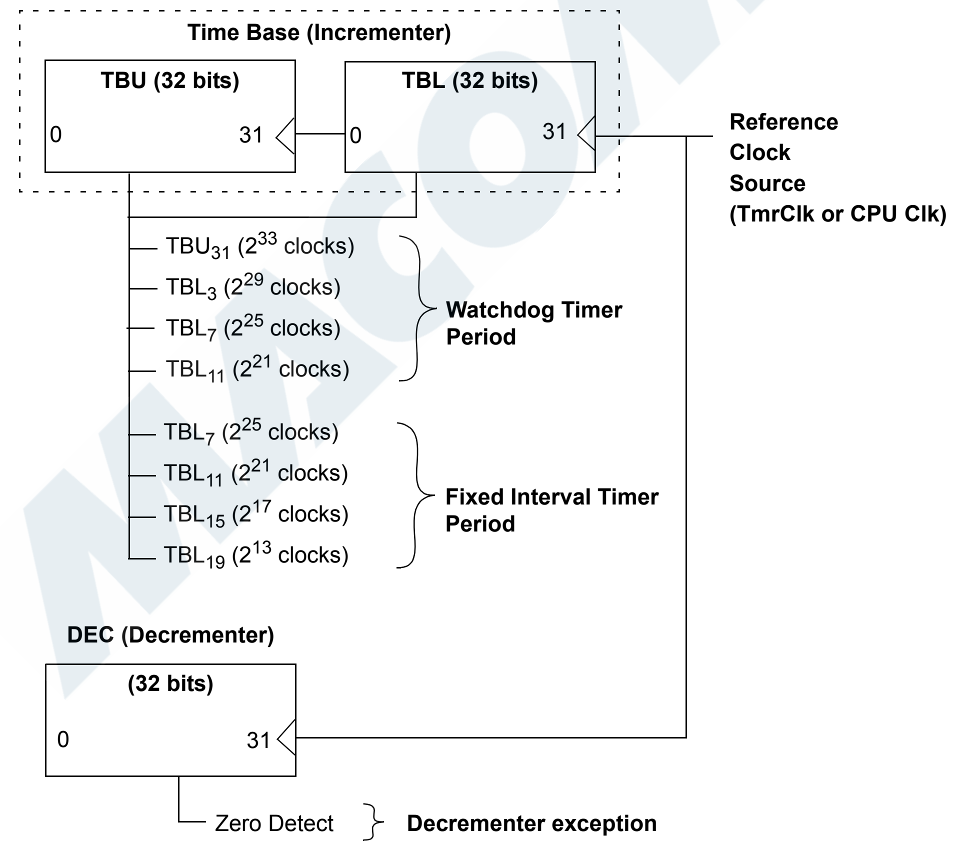
11. Timer Facilities

The PPC465 provides four timer facilities: a time base, a Decrementer (DEC), a Fixed Interval Timer (FIT), and a Watchdog Timer. These facilities, which share the same source clock frequency, can support:

- Time-of-day functions
- General software timing functions
- Peripherals requiring periodic service
- General system maintenance
- System error recovery

Figure 11-1 shows the relationship between these facilities and the clock source.

Figure 11-1. Relationship of Timer Facilities to the Time Base


Time Base (Incrementer)
TBU (32 bits)
0
31
TBL (32 bits)
0
31
Reference Clock Source (TmrClk or CPU Clk)
TBU_{31} (2^{33} clocks)
TBL_3 (2^{29} clocks)
TBL_7 (2^{25} clocks)
TBL_{11} (2^{21} clocks)
Watchdog Timer Period
TBL_7 (2^{25} clocks)
TBL_{11} (2^{21} clocks)
TBL_{15} (2^{17} clocks)
TBL_{19} (2^{13} clocks)
Fixed Interval Timer Period
DEC (Decrementer)
(32 bits)
0
31
Zero Detect
Decrementer exception

Note: CCR1[TCS] selects the Reference Clock Source. When TCS = 0, the internal CPU Clock is the clock source. When TSC = 1, the external timer clock (TmrClk) is the clock source.

11.1 Time Base

The time base is a 64-bit register which increments once during each period of the source clock, and provides a time reference. Access to the time base is via two Special Purpose Registers (SPRs). The Time Base Upper (TBU) SPR contains the high-order 32 bits of the time base, while the Time Base Lower (TBL) SPR contains the low-order 32 bits.

Software access to TBU and TBL is non-privileged for read but privileged for write, and hence different SPR numbers are used for reading than for writing. TBU and TBL are written using **mtspr** and read using **mfspr**.

The period of the 64-bit time base is approximately 5849 years for a 100 MHz clock source. The time base value itself does not generate any exceptions, even when it wraps. For most applications, the time base is set once at system reset and only read thereafter. Note that Fixed Interval Timer and Watchdog Timer exceptions (discussed below) are caused by 0→1 transitions of selected bits from the time base. Transitions of these bits caused by software alteration of the time base have the same effect as transitions caused by normal incrementing of the time base.

Figure 11-2. Time Base Lower (TBL)

0:31		Time Base Lower	Low-order 32 bits of time base.

Figure 11-3. Time Base Upper (TBU)

0:31		Time Base Upper	High-order 32 bits of time base.

11.1.1 Reading the Time Base

The following code provides an example of reading the time base.

```
loop:
mfsprRx,TBU# read TBU into GPR Rx
mfsprRy,TBL# read TBL into GPR Ry
mfsprRz,TBU# read TBU again, this time into GPR Rz
cmpwRz, Rx# see if old = new
bneloop# loop/reread if rollover occurred
```

The comparison and loop ensure that a consistent pair of values is obtained.

11.1.2 Writing the Time Base

The following code provides an example of writing the time base.

```
lwz     Rx, upper                 # load 64-bit time base value into GPRs Rx and Ry
lwz     Ry, lower
li      Rz, 0                     # set GPR Rz to 0
mtspr   TBL,Rz                    # force TBL to 0 (thereby preventing wrap into TBU)
mtspr   TBU,Rx                    # set TBU to initial value
mtspr   TBL,Ry                    # set TBL to initial value
```

11.2 Decrementer (DEC)

The DEC is a 32-bit privileged SPR that decrements at the same rate that the time base increments. The DEC is read and written using **mfspr** and **mtspr**, respectively. When a non-zero value is written to the DEC, it begins to decrement with the next time base clock. A Decrementer exception is signalled when a decrement occurs on a DEC count of 1, and the Decrementer Interrupt Status field of the Timer Status Register (TSR[DIS]; see page 313) is set. A Decrementer interrupt will occur if it is enabled by both the Decrementer Interrupt Enable field of the Timer Control Register (TCR[DIE]; see page 312) and by the External Interrupt Enable field of the Machine State Register (MSR[EE]; see “Machine State Register (MSR)” on page 253). “Interrupts and Exceptions” on page 247 provides more information on the handling of Decrementer interrupts.

The Decrementer interrupt handler software should clear TSR[DIS] before re-enabling MSR[EE], in order to avoid another Decrementer interrupt due to the same exception (unless TCR[DIE] is cleared instead).

The behavior of the DEC itself upon a decrement from a DEC value of 1 depends on which of two modes it is operating in -- normal, or auto-reload. The mode is controlled by the Auto-Reload Enable (ARE) field of the TCR. When operating in normal mode (TCR[ARE]=0), the DEC simply decrements to the value 0 and then stops decrementing until it is re-initialized by software.

When operating in auto-reload mode (TCR[ARE]=1), however, instead of decrementing to the value 0, the DEC is reloaded with the value in the Decrementer Auto-Reload (DECAR) register (see *Figure 11-5*), and continues to decrement with the next time base clock (assuming the DECAR value was non-zero). The DECAR register is a 32-bit privileged, write-only SPR, and is written using **mtspr**.

The auto-reload feature of the DEC is disabled upon reset, and must be enabled by software.

Figure 11-4. Decrementer (DEC)

0:31		Decrement value	

Figure 11-5. Decrementer Auto-Reload (DECAR)

0:31		Decrementer auto-reload value	Copied to DEC at next time base clock when DEC = 1 and auto-reload is enabled (TCR[ARE] = 1).

Using **mtspr** to force the DEC to 0 does *not* cause a Decrementer exception and thus does not cause TSR[DIS] to be set. However, if a time base clock causes a decrement from a DEC value of 1 to occur simultaneously with the writing of the DEC by a **mtspr** instruction, then the Decrementer exception *will* occur, TSR[DIS] will be set, and the DEC will be written with the value from the **mtspr**.

In order for software to quiesce the activity of the DEC and eliminate all DEC exceptions, the following procedure should be followed:

1. Write 0 to TCR[DIE]. This prevents a Decrementer exception from causing a Decrementer interrupt.
2. Write 0 to TCR[ARE]. This disables the DEC auto-reload feature.
3. Write 0 to the DEC to halt decrementing. Although this action does not itself cause a Decrementer exception, it is possible that a decrement from a DEC value of 1 has occurred since the last time that TSR[DIS] was cleared.
4. Write 1 to TSR[DIS] (DEC Interrupt Status bit). This clears the Decrementer exception by setting TSR[DIS] to 0. Because the DEC is no longer decrementing (due to having been written with 0 in step 3), no further Decrementer exceptions are possible.

11.3 Fixed Interval Timer (FIT)

The FIT provides a mechanism for causing periodic exceptions with a regular period. The FIT would typically be used by system software to invoke a periodic system maintenance function, executed by the Fixed Interval Timer interrupt handler.

A Fixed Interval Timer exception occurs on a $0 \rightarrow 1$ transition of a selected bit from the time base. Note that a Fixed Interval Timer exception will also occur if the selected time base bit transitions from $0 \rightarrow 1$ due to a **mtspr** instruction that writes 1 to that time base bit when its previous value was 0.

The Fixed Interval Timer Period (FP) field of the TCR selects one of four bits from the time base, as shown in *Table 11-1*.

Table 11-1. Fixed Interval Timer Period Selection

TCR[FP]	Time Base Bit	Period (Time Base Clocks)	Period (400 Mhz Clock)
0b00	TBL_{19}	2^{13} clocks	20.48 μs
0b01	TBL_{15}	2^{17} clocks	327.68 μs
0b10	TBL_{11}	2^{21} clocks	5.2 ms
0b11	TBL_{7}	2^{25} clocks	83.9 ms

When a Fixed Interval Timer exception occurs, the exception status is recorded by setting the Fixed interval Timer Interrupt Status (FIS) field of the TSR to 1. A Fixed Interval Timer interrupt will occur if it is enabled by both the Fixed Interval Timer Interrupt Enable (FIE) field of the TCR and by MSR[EE]. “Fixed-Interval Timer Interrupt” on page 275 provides more information on the handling of Fixed Interval Timer interrupts.

The Fixed Interval Timer interrupt handler software should clear TSR[FIS] before re-enabling MSR[EE], in order to avoid another Fixed Interval Timer interrupt due to the same exception (unless TCR[FIE] is cleared instead).

11.4 Watchdog Timer

The Watchdog Timer provides a mechanism for system error recovery in the event that the program running on the PPC465 has stalled and cannot be interrupted by the normal interrupt mechanism. The Watchdog Timer can be configured to cause a critical-class Watchdog Timer interrupt upon the expiration of a single period of the Watchdog Timer. It can also be configured to invoke a processor-initiated reset upon the expiration of a second period of the Watchdog Timer.

A Watchdog Timer exception occurs on a $0 \rightarrow 1$ transition of a selected bit from the time base. Note that a Watchdog Timer exception will also occur if the selected time base bit transitions from $0 \rightarrow 1$ due to a **mtspr** instruction that writes 1 to that time base bit when its previous value was 0.

The Watchdog Timer Period (WP) field of the TCR selects one of four bits from the time base, as shown in *Table 11-2*.

Table 11-2. Watchdog Timer Period Selection

TCR[WP]	Time Base Bit	Period (Time Base Clocks)	Period (400 MHz Clock)
0b00	TBL_{11}	2^{21} clocks	5.2 ms
0b01	TBL_7	2^{25} clocks	83.9 ms
0b10	TBL_3	2^{29} clocks	1.34 s
0b11	TBU_{31}	2^{33} clocks	21.47 s

The action taken upon a Watchdog Timer exception depends upon the status of the Enable Next Watchdog (ENW) and Watchdog Timer Interrupt Status (WIS) fields of the TSR at the time of the exception. When TSR[ENW] = 0, the next Watchdog Timer exception is “disabled”, and the only action to be taken upon the exception is to set TSR[ENW] to 1. By clearing TSR[ENW], software can guarantee that the time until the next *enabled* Watchdog Timer exception will be *at least* one full Watchdog Timer period (and a maximum of *two* full Watchdog Timer periods).

When TSR[ENW] = 1, the next Watchdog Timer exception is enabled, and the action to be taken upon the exception depends on the value of TSR[WIS] at the time of the exception. If TSR[WIS] = 0, then the action is to set TSR[WIS] to 1, at which time a Watchdog Timer interrupt will occur if enabled by both the Watchdog Timer Interrupt Enable (WIE) field of the TCR and by the Critical Interrupt Enable (CE) field of the MSR. The Watchdog Timer interrupt handler software should clear TSR[WIS] before re-enabling MSR[CE], in order to avoid another Watchdog Timer interrupt due to the same exception (unless TCR[WIE] is cleared instead). “Watchdog Timer Interrupt” on page 275 provides more information on the handling of Watchdog Timer interrupts.

If TSR[WIS] is already 1 at the time of the next Watchdog Timer exception, then the action to take depends on the value of the Watchdog Reset Control (WRC) field of the TCR. If TCR[WRC] is non-zero, then the value of the TCR[WRC] field will be copied into TSR[WRS], TCR[WRC] will be cleared, and a core reset will occur (see *Processor Core State After Reset* in the chip user’s manual for more information on core behavior when reset).

Note that once software has set TCR[WRC] to a non-zero value, it cannot be reset by software; this feature prevents errant software from disabling the Watchdog Timer reset capability.

Table 11-3 summarizes the action to be taken upon a Watchdog Timer exception according to the values of TSR[ENW] and TSR[WIS].

Table 11-3. Watchdog Timer Exception Behavior

TSR[ENW]	TSR[WIS]	Action upon Watchdog Timer exception
0	0	Set TSR[ENW] to 1
0	1	Set TSR[ENW] to 1
1	0	Set TSR[WIS] to 1. If Watchdog Timer interrupts are enabled (TCR[WIE]=1 and MSR[CE]=1), then interrupt.
1	1	Cause Watchdog Timer reset action specified by TCR[WRC]. Reset will copy pre-reset TCR[WRC] into TSR[WRS], then clear TCR[WRC].

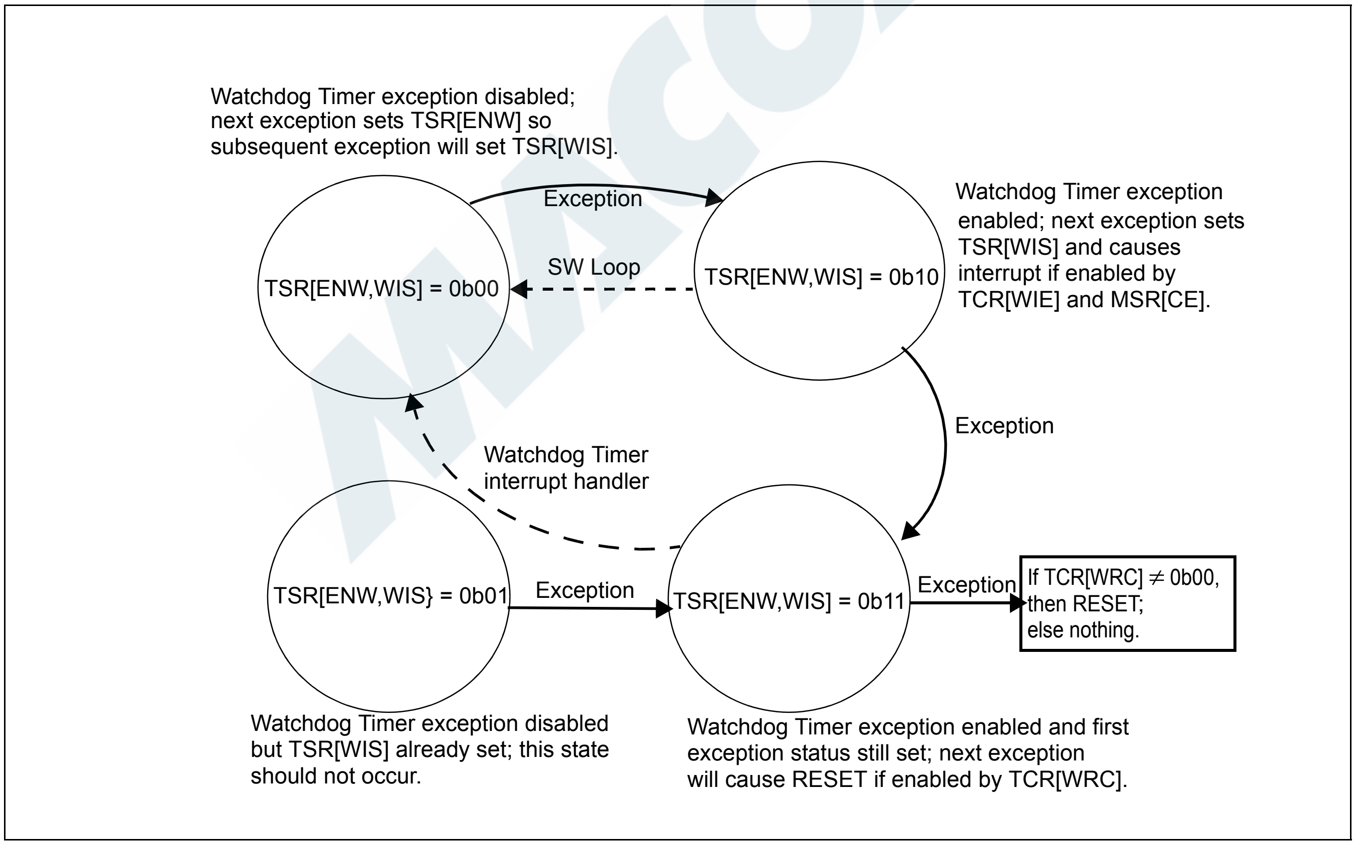
A typical system usage of the Watchdog Timer function is to enable the Watchdog Timer interrupt and the Watchdog Timer reset function in the TCR (and MSR), and to start out with both TSR[ENW] and TSR[WIS] clear. Then, a recurring software loop of reliable duration (or perhaps the interrupt handler for a periodic interrupt such as the Fixed Interval Timer interrupt) performs a periodic check of system integrity. Upon successful completion of the system check, software clears TSR[ENW], thereby ensuring that a minimum of one full Watchdog Timer period and a maximum of two full Watchdog Timer periods must expire before an enabled Watchdog Timer exception will occur.

If for some reason the recurring software loop is not successfully completed (and TSR[ENW] thus not cleared) during this period of time, then an enabled Watchdog Timer exception will occur. This will set TSR[WIS] and a Watchdog Timer interrupt will occur (if enabled by both TCR[WIE] and MSR[CE]). The occurrence of a Watchdog Timer interrupt in this kind of system is interpreted as a “system error”, insofar as the system was for some reason unable to complete the periodic system integrity check in time to avoid the enabled Watchdog Timer exception. The action taken by the Watchdog Timer interrupt handler is of course system-dependent, but typically the software will attempt to determine the nature of the problem and correct it if possible. If and when the system attempts to resume operation, the software would typically clear both TSR[WIS] and TSR[ENW], thus providing a minimum of another full Watchdog Timer period for a new system integrity check to occur.

Finally, if for some reason the Watchdog Timer interrupt is disabled, and/or the Watchdog Timer interrupt handler is unsuccessful in clearing TSR[WIS] and TSR[ENW] prior to another Watchdog Timer exception, then the next exception will cause a processor reset operation to occur, according to the value of TCR[WRC].

Figure 11-6 illustrates the sequence of Watchdog Timer events which occurs according to this typical system usage.

Figure 11-6. Watchdog State Machine


Watchdog Timer exception disabled; next exception sets TSR[ENW] so subsequent exception will set TSR[WIS].
Exception
SW Loop
TSR[ENW,WIS] = 0b00
TSR[ENW,WIS] = 0b10
Watchdog Timer exception enabled; next exception sets TSR[WIS] and causes interrupt if enabled by TCR[WIE] and MSR[CE].
Exception
Watchdog Timer interrupt handler
TSR[ENW,WIS} = 0b01
Exception
TSR[ENW,WIS] = 0b11
Exception
If TCR[WRC] ≠ 0b00, then RESET; else nothing.
Watchdog Timer exception disabled but TSR[WIS] already set; this state should not occur.
Watchdog Timer exception enabled and first exception status still set; next exception will cause RESET if enabled by TCR[WRC].

11.5 Timer Control Register (TCR)

The TCR is a privileged SPR that controls DEC, FIT, and Watchdog Timer operation. The TCR is read into a GPR using **mfspr**, and is written from a GPR using **mtspr**.

The Watchdog Timer Reset Control (WRC) field of the TCR is cleared to 0 by processor reset (see *Reset and Initialization* in the chip user's manual). Each bit of this 2-bit field is set only by software and is cleared only by hardware. For each bit of the field, once software has written it to 1, that bit remains 1 until processor reset occurs. This is to prevent errant code from disabling the Watchdog Timer reset function.

The Auto-Reload Enable (ARE) field of the TCR is also cleared to zero by processor reset. This disables the auto-reload feature of the DEC.

Figure 11-7. Timer Control Register (TCR)

0:1	WP	Watchdog Timer Period 00 2^{21} time base clocks 01 2^{25} time base clocks 10 2^{29} time base clocks 11 2^{33} time base clocks	
2:3	WRC	Watchdog Timer Reset Control 00 No Watchdog Timer reset will occur. 01 Core reset 10 Chip reset 11 System reset	TCR[WRC] resets to 0b00. Type of reset to cause upon Watchdog Timer exception with TSR[ENW,WIS]=0b11. This field can be set by software, but cannot be cleared by software, except by a software-induced reset.
4	WIE	Watchdog Timer Interrupt Enable 0 Disable Watchdog Timer interrupt. 1 Enable Watchdog Timer interrupt.	
5	DIE	Decrementer Interrupt Enable 0 Disable Decrementer interrupt. 1 Enable Decrementer interrupt.	
6:7	FP	Fixed Interval Timer (FIT) Period 00 2^{13} time base clocks 01 2^{17} time base clocks 10 2^{21} time base clocks 11 2^{25} time base clocks	
8	FIE	FIT Interrupt Enable 0 Disable Fixed Interval Timer interrupt. 1 Enable Fixed Interval Timer interrupt.	
9	ARE	Auto-Reload Enable 0 Disable auto reload. 1 Enable auto reload.	TCR[ARE] resets to 0b0.
10:31		Reserved	

11.6 Timer Status Register (TSR)

The TSR is a privileged SPR that records the status of DEC, FIT, and Watchdog Timer events. The fields of the TSR are generally set to 1 only by hardware and cleared to 0 only by software. Hardware cannot clear any fields in the TSR, nor can software set any fields. Software can read the TSR into a GPR using **mfspr**. Clearing the TSR is performed using **mtspr** by placing a 1 in the GPR source register in all bit positions which are to be cleared in the TSR, and a 0 in all other bit positions. The data written from the GPR to the TSR is not direct data, but a mask. A 1 clears the bit and a 0 leaves the corresponding TSR bit unchanged.

Figure 11-8. Timer Status Register (TSR)

0	ENW	Enable Next Watchdog Timer Exception 0 Action on next Watchdog Timer exception is to set TSR[ENW] = 1. 1 Action on next Watchdog Timer exception is governed by TSR[WIS].	
1	WIS	Watchdog Timer Interrupt Status 0 Watchdog Timer exception has not occurred. 1 Watchdog Timer exception has occurred.	
2:3	WRS	Watchdog Timer Reset Status 00 No Watchdog Timer reset has occurred. 01 Core reset was forced by Watchdog Timer. 10 Chip reset was forced by Watchdog Timer. 11 System reset was forced by Watchdog Timer.	
4	DIS	Decrementer Interrupt Status 0 Decrementer exception has not occurred. 1 Decrementer exception has occurred.	
5	FIS	Fixed Interval Timer (FIT) Interrupt Status 0 Fixed Interval Timer exception has not occurred. 1 Fixed Interval Timer exception has occurred.	
6:31		Reserved	

11.7 Freezing the Timer Facilities

The debug mechanism provides a means for temporarily “freezing” the timers upon a debug exception. Specifically, the time base and Decrementer can be prevented from incrementing and decrementing, respectively, whenever a debug exception is recorded in the Debug Status Register (DBSR). This allows a debugger to simulate the appearance of “real time”, even though the application has been temporarily “halted” to service the debug event.

See *Debug Facilities* on page 315 for more information on freezing the timers.

11.8 Selection of the Timer Clock Source

The source clock of the timers is selected by the Timer Clock Select (TCS) field of the Core Configuration Register 1 (CCR1). When set to zero, CCR1[TCS] selects the CPU clock. This is the highest frequency timer clock source.

When set to one, CCR1[TCS] selects an input to the CPU core as the timer clock (TmrClk).

12. Debug Facilities

The debug facilities of the PPC465 include support for several debug modes for debugging during hardware and software development, as well as debug events that allow developers to control the debug process. Debug registers control these debug modes and debug events. The debug registers may be accessed either through software running on the processor or through the JTAG debug port of the PPC465. Access to the debug facilities through the JTAG debug port is typically provided by a debug tool such as the RISCWatch™ development tool. A trace port, which enables the tracing of code running in real time, is also provided.

12.1 Support for Development Tools

The RISCWatch product is an example of a development tool that uses 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 PPC465.

12.2 Debug Interfaces

The PPC465 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 PPC465; the JTAG debug port is typically connected to a JTAG connector on a processor board. The trace interface connects to a trace port external to the PPC465; the trace port is typically connected to a trace connector on a processor board.

12.2.1 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 Standard 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 comply with 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 $\overline{\text{TRST}}$ 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 $\overline{\text{TRST}}$, the JTAG debug port returns to the Test-Logic Reset state.

12.2.1.1 JTAG Connector

The PPC465 implements a JTAG interface to support system debugging. The interface enables the connection of an external debug tool. Detailed information on JTAG capabilities and how to connect an external debug tool is available from the JTAG debugger vendor.

12.2.1.2 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 12-1. JTAG Instructions

Instruction	Code	Comments
Extest	11110000	IEEE 1149.1 standard
	11111001	Reserved
Sample/Preload	11110010	IEEE 1149.1 standard
IDCode	11110011	IEEE 1149.1 standard
Private	xxxx0100	Private instructions
HighZ	11110101	IEEE 1149.1a-1993 optional
Clamp	11110110	IEEE 1149.1a-1993 optional
Bypass	11111111	IEEE 1149.1 standard
	11111011	Reserved

12.2.1.3 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, in, out, 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.

12.2.1.4 JTAG Register (SDR0_JTAGID)

SDR0_JTAGID is a Device Control 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 PPC460EX, PPC460EXr, PPC431EX, PPC431EXr, PPC460GT, PPC460SX or PPC460GTx Data Sheet for the JTAG ID number.

Figure 12-1. JTAG ID Register (SDR0_JTAGID)

0:3	VERS	Version	
4:7	LOC	Developer Location	
8:19	PART	Part Number	
20:31	MANF	Manufacturer Identifier	

12.2.2 Trace Port

The PPC465 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 and allows for user-extended trace functions. A software tool with trace capability 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.

12.3 Debug Modes

The following sections describe the various debug modes supported by the PPC465. Each of these debug modes supports a particular type of debug tool or debug task commonly used in embedded systems development. For all debug modes, the various debug event types are enabled by the setting of corresponding fields in Debug Control Register 0 (DBCR0), and upon their occurrence are recorded in the Debug Status Register (DBSR).

There are four debug modes:

- Internal debug mode
- External debug mode
- Debug wait mode
- Trace debug mode

The PowerPC Book-E architecture specification deals only with internal debug mode, and the relationship of Debug interrupts to the rest of the interrupt architecture. Internal debug mode is the mode which involves debug software running on the processor itself, typically in the form of the Debug interrupt handler. The other debug modes, on the other hand, are outside the scope of the architecture, and involve special-purpose debug hardware external to the PPC465 core, connected either to the JTAG interface (for external debug mode and debug wait mode) or the trace interface (for trace debug mode). Details of these interfaces and their operation are beyond the scope of this manual.

12.3.1 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 are considered exceptions, which, in addition to recording their status in the DBSR, generate Debug interrupts if and when such interrupts are enabled (Machine State Register (MSR) DE field is 1; see *Interrupts and Exceptions* on page 247 for a description of the MSR and Debug interrupts). When a Debug interrupt occurs, special debugger software at the interrupt handler can check processor status and other conditions related to the debug event, as well as alter processor resources using all of the instructions defined for the PPC465.

Internal debug mode relies on this interrupt handling software at the Debug interrupt vector to debug software problems. This mode, used while the processor executes instructions, enables debugging of both application programs and operating system software, including all of the non-critical class interrupt handlers.

In this mode, the debugger software can communicate with the outside world through a communications port, such as a serial port, external to the processor core.

To enable internal debug mode, the IDM field of DBCR0 must be set to 1 (DBCR0[IDM] = 1). This mode can be enabled in combination with external debug mode (see *External Debug Mode* below) and/or debug wait mode (see *Debug Wait Mode* on page 318).

12.3.2 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 record their status in the DBSR and then cause the processor to enter the *stop state*, in which normal instruction execution stops and architected processor resources and memory can be accessed and altered via the JTAG interface. While in the stop state, interrupts are temporarily disabled.

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.

External debug mode relies only on internal processor resources, and no Debug interrupt handling software, 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, including problems within the Debug interrupt handler itself, or within any other critical class interrupt handlers.

External debug mode is enabled by setting DBCR0[EDM] to 1. This mode can be enabled in combination with internal debug mode (see *Internal Debug Mode* on page 317) and/or debug wait mode (see *Debug Wait Mode* below). External debug mode takes precedence over internal debug mode however, in that debug events will first cause the processor to enter stop state rather than generating a Debug interrupt, although a Debug interrupt may be pending while the processor is in the stop state.

12.3.3 Debug Wait Mode

Debug wait mode is similar to external debug mode in that debug events cause the processor to enter the stop state. However, interrupts are still enabled while in debug wait mode, such that if and when an exception occurs for which the associated interrupt type is enabled, the processor will leave the stop state and generate the interrupt. This mode is useful for real-time hardware environments which cannot tolerate interrupts being disabled for an extended period of time. In such environments, if external debug mode were to be used, various I/O devices could operate incorrectly due to not being serviced in a timely fashion when they assert an interrupt request to the processor, if the processor happened to be in stop state at the time of the interrupt request.

When in debug wait mode, as with external debug mode, access to the architected processor resources and memory is via the JTAG interface.

Debug wait mode is enabled by setting both MSR[DWE] and the debug wait mode enable within the JTAG controller to 1. Since MSR[DWE] is automatically cleared upon any interrupt, debug wait mode is temporarily disabled upon an interrupt, and then can be automatically re-enabled when returning from the interrupt due to the restoration of the MSR value upon the execution of an **rfi**, **rfci,** or **rfmci** instruction.

While debug wait mode can be enabled in combination with external debug mode, external debug mode takes precedence and interrupts are temporarily disabled, thereby effectively nullifying the effect of debug wait mode. Similarly, debug wait mode can be enabled in combination with internal debug mode. However, if Debug interrupts are enabled (MSR[DE] is 1), then any debug event will lead to an exception and a corresponding Debug interrupt, which takes precedence over the stop state associated with debug wait mode. On the other hand, if Debug interrupts are disabled (MSR[DE] is 0), then debug wait mode will take effect and a debug event will cause the processor to enter stop state.

12.3.4 Trace Debug Mode

Trace debug mode is simply the *absence* of each of the other modes. That is, if internal debug mode, external debug mode, and debug wait mode are all disabled, then the processor is in trace debug mode. While in trace debug mode, all debug events are simply recorded in the DBSR, and are indicated over the trace interface from the PPC465 core. The processor does not enter the stop state, nor does a Debug interrupt occur.

12.4 Debug Events

There are several different kinds of debug events, each of which is enabled by a field in DBCR0 (except for the Unconditional debug event) and recorded in the DBSR. *Debug Modes* on page 317 describes the operation that results when a debug event occurs while operating in any of the debug modes.

Table 12-2 lists the various debug events recognized by the PPC465. Detailed explanations of each debug event type follow the table.

Table 12-2. Debug Events

Event	Description
Instruction Address Compare (IAC)	Caused by the attempted execution of an instruction for which the address matches the conditions specified by DBCR0, DBCR1, and the IAC1–IAC4 registers.
Data Address Compare (DAC)	Caused by the attempted execution of a load, store, or cache management instruction for which the data storage address matches the conditions specified by DBCR0, DBCR2, and the DAC1–DAC2 registers.
Data Value Compare (DVC)	Caused by the attempted execution of a load, store, or cache management instruction for which the data storage address matches the conditions specified by DBCR0, DBCR2, and the DAC1–DAC2 registers, and for which the referenced data matches the value specified by the DVC1–DVC2 registers.
Branch Taken (BRT)	Caused by the attempted execution of a branch instruction for which the branch conditions are met (that is, for a branch instruction that results in the re-direction of the instruction stream).
Trap (TRAP)	Caused by the attempted execution of a **tw** or **twi** instruction for which the trap conditions are met.
Return (RET)	Caused by the attempted execution of an **rfi**, **rfci,** or **rfmci** instruction.
Instruction Complete (ICMP)	Caused by the successful completion of the execution of any instruction.
Interrupt (IRPT)	Caused by the generation of an interrupt.
Unconditional (UDE)	Caused by the assertion of an unconditional debug event request from the JTAG interface to the PPC465.

12.4.1 Instruction Address Compare (IAC) Debug Event

IAC debug events occur when execution is attempted of an instruction for which the instruction address and other parameters match the IAC conditions specified by DBCR0, DBCR1, and the IAC registers. There are four IAC registers on the PPC465, IAC1–IAC4. Depending on the IAC mode specified by DBCR1, these IAC registers can be used to specify four independent, exact IAC addresses, or they can be configured in pairs (IAC1/IAC2 and IAC3/IAC4) in order to specify *ranges* of instruction addresses for which IAC debug events should occur.

12.4.1.1 IAC Debug Event Fields

There are several fields in DBCR0 and DBCR1 which are used to specify the IAC conditions, as follows:

IAC Event Enable Field

DBCR0[IAC1, IAC2, IAC3, IAC4] are the individual IAC event enables for each of the four IAC events: IAC1, IAC2, IAC3, and IAC4. For a given IAC event to occur, the corresponding IAC event enable bit in DBCR0 must be set. When a given IAC event occurs, the corresponding DBSR[IAC1, IAC2, IAC3, IAC4] bit is set.

IAC Mode Field

DBCR1[IAC12M, IAC34M] control the comparison mode for the IAC1/IAC2 and IAC3/IAC4 events, respectively. There are three comparison modes supported by the PPC465:

- Exact comparison mode (DBCR1[IAC12M/IAC34M] = 0b00)

  In this mode, the instruction address is compared to the value in the corresponding IAC register, and the IAC event occurs only if the comparison is an exact match.

- Range inclusive comparison mode (DBCR1[IAC12M/IAC34M] = 0b10)

  In this mode, the IAC1 or IAC2 event occurs only if the instruction address is *within* the range defined by the IAC1 and IAC2 register values, as follows: $IAC1 \le address < IAC2$. Similarly, the IAC3 or IAC4 event occurs only if the instruction address is *within* the range defined by the IAC3 and IAC4 register values, as follows: $IAC3 \le address < IAC4$.

  For a given IAC1/IAC2 or IAC3/IAC4 pair, when the instruction address falls within the specified range, either one or both of the corresponding IAC debug event bits will be set in the DBSR, as determined by which of the two corresponding IAC event enable bits are set in DBCR0. For example, when the IAC1/IAC2 pair are set to range inclusive comparison mode, and the instruction address falls within the defined range, then DBCR1[IAC1, IAC2] will determine whether one or the other or both of DBSR[IAC1, IAC2] are set. It is a programming error to set either of the IAC pairs to a range comparison mode (either inclusive or exclusive) without also enabling at least one of the corresponding IAC event enable bits in DBCR0.

  Note that the IAC range auto-toggle mechanism can “switch” the IAC range mode from inclusive to exclusive, and vice-versa. See *IAC Range Mode Auto-Toggle Field* on page 321.

- Range exclusive comparison mode (DBCR1[IAC12M/IAC34M] = 0b11)

  In this mode, the IAC1 or IAC2 event occurs only if the instruction address is *outside* the range defined by the IAC1 and IAC2 register values, as follows: $address < IAC1$ or $address \ge IAC2$. Similarly, the IAC3 or IAC4 event occurs only if the instruction address is *outside* the range defined by the IAC3 and IAC4 register values, as follows: $address < IAC3$ or $address \ge IAC4$.

  For a given IAC1/IAC2 or IAC3/IAC4 pair, when the instruction address falls outside the specified range, either one or both of the corresponding IAC debug event bits will be set in the DBSR, as determined by which of the two corresponding IAC event enable bits are set in DBCR0. For example, when the

IAC1/IAC2 pair are set to range exclusive comparison mode, and the instruction address falls outside the defined range, then DBCR1[IAC1, IAC2] will determine whether one or the other or both of DBSR[IAC1, IAC2] are set. It is a programming error to set either of the IAC pairs to a range comparison mode (either inclusive or exclusive) without also enabling at least one of the corresponding IAC event enable bits in DBCR0.

Note that the IAC range auto-toggle mechanism can “switch” the IAC range mode from inclusive to exclusive, and vice-versa. See *IAC Range Mode Auto-Toggle Field* on page 321.

The PowerPC Book-E architecture defines DBCR1[IAC12M/IAC34M] = 0b01 as IAC address bit mask mode, but that mode is not supported by the PPC465, and that value of the IAC12M/IAC34M fields is reserved.

IAC User/Supervisor Field

DBCR1[IAC1US, IAC2US, IAC3US, IAC4US] are the individual IAC user/supervisor fields for each of the four IAC events. The IAC user/supervisor fields specify what operating mode the processor must be in order for the corresponding IAC event to occur. The operating mode is determined by the Problem State field of the Machine State Register (MSR[PR]; see *User and Supervisor Modes* on page 77). When the IAC user/supervisor field is 0b00, the operating mode does not matter; the IAC debug event may occur independent of the state of MSR[PR]. When this field is 0b10, the processor must be operating in supervisor mode (MSR[PR] = 0). When this field is 0b11, the processor must be operating in user mode (MSR[PR] = 1). The IAC user/supervisor field value of 0b01 is reserved.

If a pair of IAC events (IAC1/IAC2 or IAC3/IAC4) are operating in range inclusive or range exclusive mode, it is a programming error (and the results of any instruction address comparison are undefined) if the corresponding pair of IAC user/supervisor fields are not set to the same value. For example, if IAC1/IAC2 are operating in one of the range modes, then both DBCR1[IAC1US] and DBCR1[IAC2US] must be set to the same value.

IAC Effective/Real Address Field

DBCR1[IAC1ER, IAC2ER, IAC3ER, IAC4ER] are the individual IAC effective/real address fields for each of the four IAC events. The IAC effective/real address fields specify whether the instruction address comparison should be performed using the effective, virtual, or real address (see *Memory Management* on page 219) for an explanation of these different types of addresses). When the IAC effective/real address field is 0b00, the comparison is performed using the effective address only—the IAC debug event may occur independent of the instruction address space (MSR[IS]). When this field is 0b10, the IAC debug event occurs only if the effective address matches the IAC conditions and is in virtual address space 0 (MSR[IS] = 0). Similarly, when this field is 0b11, the IAC debug event occurs only if the effective address matches the IAC conditions and is in virtual address space 1 (MSR[IS] = 1). Note that in these latter two modes, in which the virtual address space of the instruction is considered, it is not the entire virtual address which is considered. The Process ID, which forms the final part of the virtual address, is not considered. Finally, the IAC effective/real address field value of 0b01 is reserved, and corresponds to the PowerPC Book-E architected real address comparison mode, which is not supported by the PPC465.

If a pair of IAC events (IAC1/IAC2 or IAC3/IAC4) are operating in range inclusive or range exclusive mode, it is a programming error (and the results of any instruction address comparison are undefined) if the corresponding pair of IAC effective/real address fields are not set to the same value. For example, if IAC1/IAC2 are operating in one of the range modes, then both DBCR1[IAC1ER] and DBCR1[IAC2ER] must be set to the same value.

IAC Range Mode Auto-Toggle Field

DBCR1[IAC12AT, IAC34AT] control the auto-toggle mechanism for the IAC1/IAC2 and IAC3/IAC4 events, respectively. When the IAC mode for one of the pairs of IAC debug events is set to one of the range modes (either range inclusive or range exclusive), then the IAC range mode auto-toggle field corresponding to that pair of IAC debug events controls whether or not the range mode will automatically “toggle” from inclusive to exclusive, and vice-versa. When the IAC range mode auto-toggle field is set to 1, this automatic toggling is enabled; otherwise it is disabled. It is a programming error (and the results of any instruction address comparison are undefined) if an IAC range mode auto-toggle field is set to 1 without the corresponding IAC mode field being set to one of the range modes.

When auto-toggle is enabled for a pair of IAC debug events, then upon each occurrence of an IAC debug event within that pair the value of the corresponding auto-toggle status field in the DBSR (DBSR[IAC12ATS, IAC34ATS]) is reversed. That is, if the auto-toggle status field is 0 before the occurrence of the IAC debug event, then it will be changed to 1 at the same time that the IAC debug event is recorded in the DBSR. Conversely, if the auto-toggle status field is 1 before the occurrence of the IAC debug event, then it will be changed to 0 at the same time that the IAC debug event is recorded in the DBSR.

Furthermore, when auto-toggle is enabled, the auto-toggle status field of the DBSR affects the interpretation of the IAC mode field of DBCR1. If the auto-toggle status field is 0, then the IAC mode field is interpreted in the normal fashion, as defined in *IAC Mode Field* on page 320. That is, the IAC mode field value of 0b10 selects range inclusive mode, whereas the value of 0b11 selects range exclusive mode. On the other hand, when the auto-toggle status field is 1, then the interpretation of the IAC mode field is “reversed”. That is, the IAC mode field value of 0b10 selects range exclusive mode, whereas the value of 0b11 selects range inclusive mode.

The relationship of the IAC mode, IAC range mode auto-toggle, and IAC range mode auto-toggle status fields is summarized in *Table 12-3*.

Table 12-3. IAC Range Mode Auto-Toggle Summary

DBCR1	DBCR1	DBSR	IAC Mode
IAC12M/IAC34M	IAC12AT/IAC34AT	IAC12ATS/IAC34ATS	
0b10	0	—	Range Inclusive
0b10	1	0	Range Inclusive
0b10	1	1	Range Exclusive
0b11	0	—	Range Exclusive
0b11	1	0	Range Exclusive
0b11	1	1	Range Inclusive

The affect of the auto-toggle mechanism is to cause the IAC mode to switch back and forth between range inclusive mode and range exclusive mode, as each IAC range mode debug event occurs. For example, if the IAC mode is set to range inclusive, and auto-toggle is enabled, and the auto-toggle status field is 0, then the first IAC debug event will be a range inclusive event. Upon that event, the DBSR auto-toggle status field will be set to 1, and the next IAC debug event will then be a range exclusive event. Upon this next event, the DBSR auto-toggle status field will be set back to 0, such that the next IAC debug event will again be a range inclusive event.

This auto-toggling between range inclusive and range exclusive IAC modes is particularly helpful when enabling IAC range mode debug events in trace debug mode. A common debug operation is to detect when the instruction stream enters a particular region of the instruction address space (range inclusive mode). Once having entered the region of interest (a range inclusive event), it is common for the debugger to then want to be informed when that region is exited (a range exclusive event). By automatically toggling to range exclusive mode upon the occurrence of the range inclusive IAC debug event, this particular debug operation is facilitated. Furthermore, by not remaining in range inclusive mode upon entry to the region of interest, the debugger avoids a continuous stream of range inclusive

IAC debug events while the processor continues to execute instructions within that region, which can often be for a very long series of instructions.

12.4.1.2 IAC Debug Event Processing

When operating in external debug mode or debug wait mode, the occurrence of an IAC debug event is recorded in the corresponding bit of the DBSR and causes the instruction execution to be suppressed. The processor then enters the stop state and ceases the processing of instructions. The program counter will contain the address of the instruction which caused the IAC debug event. Similarly, when operating in internal debug mode with Debug interrupts enabled (MSR[DE] = 1), the occurrence of an IAC debug event is recorded in the DBSR and causes the instruction execution to be suppressed. A Debug interrupt then occurs with Critical Save/Restore Register 0 (CSRR0) set to the address of the instruction which caused the IAC debug event.

When operating in internal debug mode (and not also in external debug mode nor debug wait mode) with Debug interrupts disabled (MSR[DE] = 0), the behavior of IAC debug events depends on the IAC mode. If the IAC mode is set to exact comparison, then an IAC debug event can occur and will set the corresponding IAC field of the DBSR, along with the Imprecise Debug Event (IDE) field of the DBSR. The instruction execution is not suppressed, as no Debug interrupt will occur immediately. Instead, instruction execution continues, and a Debug interrupt will occur if and when MSR[DE] is set to 1, thereby enabling Debug interrupts, assuming software has not cleared the IAC debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the Debug interrupt handler software may query the DBSR[IDE] field to determine that the Debug interrupt has occurred imprecisely. On the other hand, if the IAC mode is set to either range inclusive or range exclusive mode, then IAC debug events cannot occur when operating in internal debug mode with MSR[DE] = 0, unless external debug mode and/or debug wait mode is also enabled.

When operating in trace mode, the occurrence of an IAC debug event simply sets the corresponding IAC field of the DBSR and is indicated over the trace interface, and instruction execution continues.

12.4.2 Data Address Compare (DAC) Debug Event

DAC debug events occur when execution is attempted of a load, store, or cache management instruction for which the data storage address and other parameters match the DAC conditions specified by DBCR0, DBCR2, and the DAC registers. There are two DAC registers on the PPC465, DAC1 and DAC2. Depending on the DAC mode specified by DBCR2, these DAC registers can be used to specify two independent, exact DAC addresses, or they can be configured to operate as a pair. When operating as a pair, then can specify either a *range* of data storage addresses for which DAC debug events should occur, or a combination of an address and an *address bit mask* for selective comparison with the data storage address.

Note that for integer load and store instructions, and for cache management instructions, the address that is used in the DAC comparison is the starting data address calculated as part of the instruction execution. As explained in the instruction definitions for the cache management instructions, the target operand of these instructions is an aligned cache block, which on the PPC465 is 32 bytes. Therefore, the storage reference for these instructions effectively ignores the low-order five bits of the calculated data address, and the entire aligned 32-byte cache block—which starts at the calculated data address as modified with the low-order five bits set to 0b00000—is accessed. However, the DAC comparison does not take into account this implicit 32-byte alignment of the storage reference of a cache management instruction, and instead the DAC comparison considers the entire data address, as calculated according to the instruction definition.

On the other hand, for auxiliary processor load and store instructions, the AP interface can specify that the PPC465 should *force* the storage access to be aligned on an operand-size boundary, by zeroing the appropriate number of low-order address bits. In such a case, the DAC comparison is performed against this modified, alignment-forced address, rather than the original address as calculated according to the instruction definition.

12.4.2.1 DAC Debug Event Fields

There are several fields in DBCR0 and DBCR2 which are used to specify the DAC conditions, as follows:

DAC Event Enable Field

DBCR0[DAC1R, DAC1W, DAC2R, DAC2W] are the individual DAC event enables for the two DAC events DAC1 and DAC2. For each of the two DAC events, there is one enable for DAC read events, and another for DAC write events. Load, **dcbt**, **dcbtst**, **icbi**, and **icbt** instructions may cause DAC read events, while store, **dcbst**, **dcbf**, **dcbi**, and **dcbz** instructions may cause DAC write events (see *DAC Debug Events Applied to Various Instruction Types* on page 327 for more information on these instructions and the types of DAC debug events they may cause). For a given DAC event to occur, the corresponding DAC event enable bit in DBCR0 for the particular operation type must be set. When a DAC event occurs, the corresponding DBSR[DAC1R, DAC1W, DAC2R, DAC2W] bit is set. These same DBSR bits are shared by DVC debug events (see *Data Value Compare (DVC) Debug Event* on page 328).

DAC Mode Field

DBCR2[DAC12M] controls the comparison mode for the DAC1 and DAC2 events. There are four comparison modes supported by the PPC465:

- Exact comparison mode (DBCR2[DAC12M] = 0b00)

  In this mode, the data address is compared to the value in the corresponding DAC register, and the DAC event occurs only if the comparison is an exact match.

- Address bit mask mode (DBCR2[DAC12M] = 0b01)

  In this mode, the DAC1 or DAC2 event occurs only if the data address matches the value in the DAC1 register, as masked by the value in the DAC2 register. That is, the DAC1 register specifies an address value, and the DAC2 register specifies an *address bit mask* which determines which bit of the data address should participate in the comparison to the DAC1 value. For every bit set to 1 in the DAC2 register, the corresponding data address bit must match the value of the same bit position in the DAC1 register. For every bit set to 0 in the DAC2 register, the corresponding address bit comparison does not affect the result of the DAC event determination.

  This comparison mode is useful for detecting accesses to a particular byte address, when the accesses may be of various sizes. For example, if the debugger is interested in detecting accesses to byte address 0x0000 0003, then these accesses may occur due to a byte access to that specific address, or due to a half word access to address 0x0000 0002, or due to a word access to address 0x0000 0000. By using address bit mask mode and specifying that the low-order two bits of the address should be ignored (that is, setting the address bit mask in DAC2 to 0xFFFF FFFC), the debugger can detect each of these types of access to byte address 0x0000 0003.

  When the data address matches the address bit mask mode conditions, either one or both of the DAC debug event bits corresponding to the operation type (read or write) will be set in the DBSR, as determined by which of the corresponding two DAC event enable bits are set in DBCR0. That is, when an address bit mask mode DAC debug event occurs, the setting of DBCR2[DAC1R, DAC1W, DAC2R, DAC2W] will determine whether one or the other or both of the DBSR[DAC1R, DAC1W, DAC2R, DAC2W] bits corresponding to the operation type are set. It is a programming error to set the DAC mode field to address bit mask mode without also enabling at least one of the four DAC event enable bits in DBCR0.

- Range inclusive comparison mode (DBCR2[DAC12M] = 0b10)

  In this mode, the DAC1 or DAC2 event occurs only if the data address is *within* the range defined by the DAC1 and DAC2 register values, as follows: DAC1 $\leq$ address $<$ DAC2.

When the data address falls within the specified range, either one or both of the DAC debug event bits corresponding to the operation type (read or write) will be set in the DBSR, as determined by which of the corresponding two DAC event enable bits are set in DBCR0. That is, when a range inclusive mode DAC debug event occurs, the setting of DBCR2[DAC1R, DAC1W, DAC2R, DAC2W] will determine whether one or the other or both of the DBSR[DAC1R, DAC1W, DAC2R, DAC2W] bits corresponding to the operation type are set. It is a programming error to set the DAC mode field to a range comparison mode (either inclusive or exclusive) without also enabling at least one of the four DAC event enable bits in DBCR0.

- Range exclusive comparison mode (DBCR2[DAC12M] = 0b11)

  In this mode, the DAC1 or DAC2 event occurs only if the data address is *outside* the range defined by the DAC1 and DAC2 register values, as follows: address $<$ DAC1 or address $\geq$ DAC2.

  When the data address falls outside the specified range, either one or both of the DAC debug event bits corresponding to the operation type (read or write) will be set in the DBSR, as determined by which of the corresponding two DAC event enable bits are set in DBCR0. That is, when a range exclusive mode DAC debug event occurs, the setting of DBCR2[DAC1R, DAC1W, DAC2R, DAC2W] will determine whether one or the other or both of the DBSR[DAC1R, DAC1W, DAC2R, DAC2W] bits corresponding to the operation type are set. It is a programming error to set the DAC mode field to a range comparison mode (either inclusive or exclusive) without also enabling at least one of the four DAC event enable bits in DBCR0.

DAC User/Supervisor Field

DBCR2[DAC1US, DAC2US] are the individual DAC user/supervisor fields for the two DAC events. The DAC user/supervisor fields specify what operating mode the processor must be in order for the corresponding DAC event to occur. The operating mode is determined by the Problem State field of the Machine State Register (MSR[PR]; see *User and Supervisor Modes* on page 77). When the DAC user/supervisor field is 0b00, the operating mode does not matter—the DAC debug event may occur independent of the state of MSR[PR]. When this field is 0b10, the processor must be operating in supervisor mode (MSR[PR] = 0). When this field is 0b11, the processor must be operating in user mode (MSR[PR] = 1). The DAC user/supervisor field value of 0b01 is reserved.

If the DAC mode is set to one of the “paired” modes (address bit mask mode, or one of the two range modes), it is a programming error (and the results of any data address comparison are undefined) if DBCR2[DAC1US] and DBCR2[DAC2US] are not set to the same value.

DAC Effective/Real Address Field

DBCR2[DAC1ER, DAC2ER] are the individual DAC effective/real address fields for the two DAC events. The DAC effective/real address fields specify whether the instruction address comparison should be performed using the effective, virtual, or real address (see *Memory Management* on page 219) for an explanation of these different types of addresses). When the DAC effective/real address field is 0b00, the comparison is performed using the effective address only; the DAC debug event may occur independent of the data address space (MSR[DS]). When this field is 0b10, the DAC debug event occurs only if the effective address matches the DAC conditions and is in virtual address space 0 (MSR[DS] = 0). Similarly, when this field is 0b11, the DAC debug event occurs only if the effective address matches the DAC conditions and is in virtual address space 1 (MSR[DS] = 1). Note that in these latter two modes, in which the virtual address space of the data is considered, it is not the entire virtual address which is considered. The Process ID, which forms the final part of the virtual address, is not considered. Finally, the DAC effective/real address field value of 0b01 is reserved, and corresponds to the PowerPC Book-E architected real address comparison mode, which is not supported by the PPC465.

If the DAC mode is set to one of the “paired” modes (address bit mask mode, or one of the two range modes), it is a programming error (and the results of any data address comparison are undefined) if DBCR2[DAC1ER] and DBCR2[DAC2ER] are not set to the same value.

DVC Byte Enable Field

DBCR2[DVC1BE, DVC2BE] are the individual data *value* compare (DVC) byte enable fields for the two DVC events. These fields must be disabled (by being set to 4b0000) in order for the corresponding DAC debug event to be enabled. In other words, when any of the DVC byte enable field bits for a given DVC event are set to 1, the corresponding DAC event is disabled, and the various DAC field conditions are used in conjunction with the DVC field conditions to determine whether a DVC event should occur. See *Data Value Compare (DVC) Debug Event* on page 328 for more information on DVC events.

12.4.2.2 DAC Debug Event Processing

The behavior of the PPC465 upon a DAC debug event depends on the setting of DBCR2[DAC12A]. This field of DBCR2 controls whether DAC debug events are processed in a *synchronous* (DBCR2[DAC12A] = 0) or an *asynchronous* (DBCR2[DAC12A] = 1) fashion.

DBCR2[DAC12A] = 0 (Synchronous Mode)

When operating in external debug mode or debug wait mode, the occurrence of a DAC debug event is recorded in the corresponding bit of the DBSR and causes the instruction execution to be suppressed. The processor then enters the stop state and ceases the processing of instructions. The program counter will contain the address of the instruction which caused the DAC debug event. Similarly, when operating in internal debug mode with Debug interrupts enabled (MSR[DE] = 1), the occurrence of a DAC debug event is recorded in the DBSR and causes the instruction execution to be suppressed. A Debug interrupt will occur with CSRR0 set to the address of the instruction which caused the DAC debug event.

When operating in internal debug mode (and not also in external debug mode nor debug wait mode) with Debug interrupts disabled (MSR[DE] = 0), then a DAC debug event will set the corresponding DAC field of the DBSR, along with the Imprecise Debug Event (IDE) field of the DBSR. The instruction execution is not suppressed, as no Debug interrupt will occur immediately. Instead, instruction execution continues, and a Debug interrupt will occur if and when MSR[DE] is set to 1, thereby enabling Debug interrupts, assuming software has not cleared the DAC debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the Debug interrupt handler software may query the DBSR[IDE] field to determine that the Debug interrupt has occurred imprecisely.

When operating in trace mode, the occurrence of a DAC debug event simply sets the corresponding DAC field of the DBSR and is indicated over the trace interface, and instruction execution continues. DBCR2[DAC12A] does not affect the processing of DAC debug events when operating in trace mode.

Engineering Note: When DAC debug events are enabled in any debug mode other than trace mode, and DBCR2[DAC12A] is set to 0 (synchronous mode), in order for the PPC465 to deal with a DAC-related Debug interrupt in a synchronous fashion, the processing of all potential DAC debug event-causing instructions (loads, stores, and cache management instructions) is impacted by one processor cycle. This one cycle impact occurs whether or not the instruction is actually causing a DAC debug event. Overall processor performance is thus significantly impacted if synchronous mode DAC debug events are enabled. In order to maintain normal processor performance while DAC debug events are enabled *and in the absence of any actual DAC debug events*, software should set DBCR2[DAC12A] to 1.

DBCR2[DAC12A] = 1 (Asynchronous Mode)

When operating in external debug mode or debug wait mode, the occurrence of a DAC debug event is recorded in the corresponding bit of the DBSR and causes the processor to enter stop state and cease processing instructions. However, the determination and processing of the DAC debug event is not handled synchronously with respect to the instruction execution. That is, the processor may process the DAC debug event and enter the stop state either before or after the completion of the instruction causing the event. If the DAC debug event is processed *before* the completion of the instruction causing the event, then upon entering the stop state the program counter will contain the address of that instruction, and that instruction's execution will have been suppressed. Conversely, if the DAC debug event is processed *after* the completion of the instruction causing the event, then the program counter will contain the address of some instruction after the one which caused the event. Whether or not the DAC debug event processing occurs before or after the completion of the instruction depends on the particular circumstances surrounding the instruction's execution, the details of which are generally beyond the scope of this document.

Similarly, when operating in internal debug mode with Debug interrupts enabled (MSR[DE] = 1), the occurrence of a DAC debug event is recorded in the DBSR and will generate a Debug interrupt with CSRR0 set to the address of the instruction which caused the DAC debug event, or to the address of some subsequent instruction, depending upon whether the event is processed before or after the instruction completes.

When operating in internal debug mode (and not also in external debug mode nor debug wait mode) with Debug interrupts disabled (MSR[DE] = 0), then a DAC debug event will set the corresponding DAC field of the DBSR, along with the Imprecise Debug Event (IDE) field of the DBSR. Instruction execution continues, and a Debug interrupt will occur if and when MSR[DE] is set to 1, thereby enabling Debug interrupts, assuming software has not cleared the DAC debug event status from the DBSR in the meantime. Upon such a "delayed" interrupt, the Debug interrupt handler software may query the DBSR[IDE] field to determine that the Debug interrupt has occurred imprecisely.

When operating in trace mode, the occurrence of a DAC debug event simply sets the corresponding DAC field of the DBSR and is indicated over the trace interface, and instruction execution continues. DBCR2[DAC12A] does not affect the processing of DAC debug events when operating in trace mode.

12.4.2.3 DAC Debug Events Applied to Instructions that Result in Multiple Storage Accesses

Certain misaligned load and store instructions are handled by making multiple, independent storage accesses. Similarly, load and store multiple and string instructions which access more than one register result in more than one storage access. *Load and Store Alignment* on page 140 provides a detailed description of the circumstances that lead to such multiple storage accesses being made as the result of the execution of a single instruction.

Whenever the execution of a given instruction results in multiple storage accesses, the data address of each access is independently considered for whether or not it will cause a DAC debug event.

12.4.2.4 DAC Debug Events Applied to Various Instruction Types

Various special cases apply to the cache management instructions, the store word conditional indexed (**stwcx.**) instruction, and the load and store string indexed (**lswx**, **stswx**) instructions, with regards to DAC debug events. These special cases are as follows:

dcbz, **dcbi**

The **dcbz** and **dcbi** instructions are considered "stores" with respect to both storage access control and DAC debug events. The **dcbz** instruction directly changes the contents of a given storage location, whereas the **dcbi** instruction can indirectly change the contents of a given storage location by invalidating data which has been modified within the data cache, thereby "restoring" the value of the location to the "old" contents of memory. As "store" operations, they may cause DAC write debug events.

dcbst, **dcbf**

The **dcbst** and **dcbf** instructions are considered "loads" with respect to storage access control, since they do not change the contents of a given storage location. They may merely cause the data at that storage location to be moved from the data cache out to memory. However, in a debug environment, the fact that these instructions may lead to write operations on the external interface is typically the event of interest. Therefore, these instructions are considered “stores” with respect to DAC debug events, and may cause DAC write debug events.

dcbt, **dcbtst**, **icbt**

The *touch* instructions are considered “loads” with respect to both storage access control and DAC debug events. However, these instructions are treated as no-ops if they reference caching inhibited storage locations, or if they cause Data Storage or Data TLB Miss exceptions. Consequently, if a *touch* instruction is being treated as a no-op for one of these reasons, then it does not cause a DAC read debug event. However, if a *touch* instruction is not being treated as a no-op for one of these reasons, it may cause a DAC read debug event.

dcba

The **dcba** instruction is treated as a no-op by the PPC465, and thus will not cause a DAC debug event.

icbi

The **icbi** instruction is considered a “load” with respect to both storage access control and DAC debug events, and thus may cause a DAC read debug event.

dccci, **dcread**, **iccci**, **icread**

The **dccci** and **iccci** instructions do not generate an address, but rather they affect the entire data and instruction cache, respectively. Similarly, the **dcread** and **icread** instructions do not generate an address, but rather an “index” which is used to select a particular location in the respective cache, without regard to the storage address represented by that location. Therefore, none of these instructions cause DAC debug events.

stwcx.

If the execution of a **stwcx.** instruction would otherwise have caused a DAC write debug event, but the processor does not have the reservation from a **lwarx** instruction, then the DAC write debug event does not occur since the storage location does not get written.

lswx, **stswx**

DAC debug events do not occur for **lswx** or **stswx** instructions with a length of 0 (XER[TBC] = 0), since these instructions do not actually access storage.

12.4.3 Data Value Compare (DVC) Debug Event

DVC debug events occur when execution is attempted of a load, store, or **dcbz** instruction for which the data storage address and other parameters match the DAC conditions specified by DBCR0, DBCR2, and the DAC registers, and for which the data accessed matches the DVC conditions specified by DBCR2 and the DVC registers. In other words, in order for a DVC debug event to occur, the conditions for a DAC debug event must first be met, and then the data must also match the DVC conditions. *Data Address Compare (DAC) Debug Event* on page 323 describes the DAC conditions. In addition to the DAC conditions, there are two DVC registers on the PPC465, DVC1 and DVC2. The DVC registers can be used to specify two independent, 4-byte data values, which are selectively compared against the data being accessed by a given load, store, or cache management instruction.

When a DVC event occurs, the corresponding DBSR[DAC1R, DAC1W, DAC2R, DAC2W] bit is set. These same DBSR bits are shared by DAC debug events.

12.4.3.1 DVC Debug Event Fields

In addition to the DAC debug event fields described in *DAC Debug Event Fields* on page 324, and the DVC registers themselves, there are two fields in DBCR2 which are used to specify the DVC conditions, as follows:

DVC Byte Enable Field

DBCR2[DVC1BE, DVC2BE] are the individual DVC byte enable fields for the two DVC events. When one or the other (or both) of these fields is disabled (by being set to 4b0000), the corresponding DVC debug event is disabled (the corresponding DAC debug event may still be enabled, as determined by the DAC debug event enable field of DBCR0). When either one or both of these fields is enabled (by being set to a non-zero value), then the corresponding DVC debug event is enabled.

Each bit of a given DVC byte enable field corresponds to a byte position within an aligned word of memory. For a given aligned word of memory, the byte offsets (or “byte lanes”) within that word are numbered 0, 1, 2, and 3, starting from the left-most (most significant) byte of the word. Accordingly, bits 0:3 of a given DVC byte enable field correspond to bytes 0:3 of an aligned word of memory being accessed.

For an access to “match” the DVC conditions for a given byte, the access must be actually transferring data on that given byte position *and* the data must match the corresponding byte value within the DVC register.

For each storage access, the DVC comparison is made against the bytes that are being accessed within the aligned word of memory containing the starting byte of the transfer. For example, consider a load word instruction with a starting data address of x01. The four bytes from memory are located at addresses 0x01–0x04, but the aligned word of memory containing the starting byte consists of addresses 0x00–0x03. Thus the only bytes being accessed within the aligned word of memory containing the starting byte are the bytes at addresses 0x01–0x03, and only these bytes are considered in the DVC comparison. The byte transferred from address 0x04 is not considered.

DVC Mode Field

DBCR2[DVC1M, DVC2M] are the individual DVC mode fields for the two DVC events. Each one of these fields specifies the particular data value comparison mode for the corresponding DVC debug event. There are three comparison modes supported by the PPC465:

- AND comparison mode (DBCR2[DVC1M, DVC2M] = 0b01)

  In this mode, all data byte lanes enabled by a DVC byte enable field must be being accessed and must match the corresponding byte data value in the corresponding DVC1 or DVC2 register.

- OR comparison mode (DBCR2[DVC1M, DVC2M] = 0b10)

  In this mode, at least one data byte lane that is enabled by a DVC byte enable field must be being accessed and must match the corresponding byte data value in the corresponding DVC1 or DVC2 register.

- AND-OR comparison mode (DBCR2[DVC1M, DVC2M] = 0b11)

  In this mode, the four byte lanes of an aligned word are divided into two pairs, with byte lanes 0 and 1 being in one pair, and byte lanes 2 and 3 in the other pair. The DVC comparison mode for each pair of byte lanes operates in AND mode, and then the results of these two AND mode comparisons are ORed together to determine whether a DVC debug event occurs. In other words, a DVC debug event occurs if either one or both of the pairs of byte lanes satisfy the AND mode comparison requirements.

This mode may be used to cause a DVC debug event upon an access of a particular half word data value in either of the two half words of a word in memory.

12.4.3.2 DVC Debug Event Processing

The behavior of the PPC465 upon a DVC debug event depends on the setting of DBCR2[DAC12A]. This field of DBCR2 controls whether DVC debug events are processed in a *synchronous* (DBCR2[DAC12A] = 0) or an *asynchronous* (DBCR2[DAC12A] = 1) fashion. The processing of DVC debug events is the same as it is for DAC debug events. See *DAC Debug Event Processing* on page 326 for more information.

12.4.3.3 DVC Debug Events Applied to Instructions that Result in Multiple Storage Accesses

Certain misaligned load and store instructions are handled by making multiple, independent storage accesses. Similarly, load and store multiple and string instructions which access more than one register result in more than one storage access. *Load and Store Alignment* on page 140 provides a detailed description of the circumstances that lead to such multiple storage accesses being made as the result of the execution of a single instruction.

Whenever the execution of a given instruction results in multiple storage accesses, the address and data of each access is independently considered for whether or not it will cause a DVC debug event.

12.4.3.4 DVC Debug Events Applied to Various Instruction Types

Various special cases apply to the cache management instructions, the store word conditional indexed (**stwcx.**) instruction, and the load and store string indexed (**lswx**, **stswx**) instructions, with regards to DVC debug events. These special cases are as follows:

dcbz

The **dcbz** instruction is the only cache management instruction which can cause a DVC debug event. **dcbz** is the only such instruction which actually writes new data to a storage location (in this case, an entire 32-byte data cache line is written to zeroes).

stwcx.

If the execution of a **stwcx.** instruction would otherwise have caused a DVC write debug event, but the processor does not have the reservation from a **lwarx** instruction, then the DVC write debug event does not occur since the storage location does not get written.

lswx, **stswx**

DVC debug events do not occur for **lswx** or **stswx** instructions with a length of 0 (XER[TBC] = 0), since these instructions do not actually access storage.

12.4.4 Branch Taken (BRT) Debug Event

BRT debug events occur when BRT debug events are enabled (DBCR0[BRT] = 1) and execution is attempted of a branch instruction for which the branch condition(s) are satisfied, such that the instruction stream will be redirected to the target address of the branch.

When operating in external debug mode or debug wait mode, the occurrence of a BRT debug event is recorded in DBSR[BRT] and causes the instruction execution to be suppressed. The processor then enters the stop state and ceases the processing of instructions. The program counter will contain the address of the branch instruction which caused the BRT debug event. Similarly, when operating in internal debug mode with Debug interrupts enabled

(MSR[DE] = 1), the occurrence of a BRT debug event is recorded in DBSR[BRT] and causes the instruction execution to be suppressed. A Debug interrupt will occur with CSRR0 set to the address of the branch instruction which caused the BRT debug event.

When operating in internal debug mode (and not also in external debug mode nor debug wait mode) with Debug interrupts disabled (MSR[DE] = 0), then BRT debug events cannot occur. Since taken branches are a very common operation and thus likely to be frequently executed within the critical class interrupt handlers (which typically have MSR[DE] set to 0), allowing BRT debug events under these conditions would lead to an undesirable number of delayed (and hence imprecise) Debug interrupts.

When operating in trace mode, the occurrence of a BRT debug event is simply recorded in DBSR[BRT] and is indicated over the trace interface, and instruction execution continues.

12.4.5 Trap (TRAP) Debug Event

TRAP debug events occur when TRAP debug events are enabled (DBCR0[TRAP] = 1) and execution is attempted of a trap (**tw**, **twi**) instruction for which the trap condition is satisfied.

When operating in external debug mode or debug wait mode, the occurrence of a TRAP debug event is recorded in DBSR[TRAP] and causes the instruction execution to be suppressed. The processor then enters the stop state and ceases the processing of instructions. The program counter will contain the address of the trap instruction which caused the TRAP debug event. Similarly, when operating in internal debug mode with Debug interrupts enabled (MSR[DE] = 1), the occurrence of a TRAP debug event is recorded in DBSR[TRAP] and causes the instruction execution to be suppressed. A Debug interrupt will occur with CSRR0 set to the address of the trap instruction which caused the TRAP debug event.

When operating in internal debug mode (and not also in external debug mode nor debug wait mode) with Debug interrupts disabled (MSR[DE] = 0), the occurrence of a TRAP debug event will set DBSR[TRAP], along with the Imprecise Debug Event (IDE) field of the DBSR. Although a Debug interrupt will not occur immediately, the instruction execution is suppressed as a Trap exception type Program interrupt will occur instead. A Debug interrupt will also occur later, if and when MSR[DE] is set to 1, thereby enabling Debug interrupts, assuming software has not cleared the TRAP debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the Debug interrupt handler software may query the DBSR[IDE] field to determine that the Debug interrupt has occurred imprecisely.

When operating in trace mode, the occurrence of a TRAP debug event is simply recorded in DBSR[TRAP] and is indicated over the trace interface, and instruction execution continues.

12.4.6 Return (RET) Debug Event

RET debug events occur when RET debug events are enabled (DBCR0[RET] = 1) and execution is attempted of a return (**rfi**, **rfci,** or **rfmci**) instruction.

When operating in external debug mode or debug wait mode, the occurrence of a RET debug event is recorded in DBSR[RET] and causes the instruction execution to be suppressed. The processor then enters the stop state and ceases the processing of instructions. The program counter will contain the address of the return instruction which caused the RET debug event. Similarly, when operating in internal debug mode with Debug interrupts enabled (MSR[DE] = 1), the occurrence of a RET debug event is recorded in DBSR[RET] and causes the instruction execution to be suppressed. A Debug interrupt will occur with CSRR0 set to the address of the return instruction which caused the RET debug event.

When operating in internal debug mode (and not also in external debug mode nor debug wait mode) with Debug interrupts disabled (MSR[DE] = 0), then RET debug events can occur only for **rfi** instructions, and not for **rfci** or **rfmci** instructions. Since the **rfci** or **rfmci** instruction is typically used to return from a critical class interrupt handler (including the Debug interrupt itself), and MSR[DE] is typically 0 at the time of the return, the **rfci** or **rfmci** must not be allowed to cause a RET debug event under these conditions, or else it would not be possible to return from the critical class interrupts.

For the **rfi** instruction only, if a RET debug event occurs under these conditions (internal debug mode enabled, external debug mode and debug wait mode disabled, and MSR[DE] = 0), then DBSR[RET] is set, along with the Imprecise Debug Event (IDE) field of the DBSR. The instruction execution is not suppressed, as no Debug interrupt will occur immediately. Instead, instruction execution continues, and a Debug interrupt will occur if and when MSR[DE] is set to 1, thereby enabling Debug interrupts, assuming software has not cleared the RET debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the Debug interrupt handler software may query the DBSR[IDE] field to determine that the Debug interrupt has occurred imprecisely.

When operating in trace mode, the occurrence of a RET debug event is simply recorded in DBSR[RET] and is indicated over the trace interface, and instruction execution continues.

12.4.7 Instruction Complete (ICMP) Debug Event

ICMP debug events occur when ICMP debug events are enabled (DBCR0[ICMP] = 1) and the PPC465 completes the execution of any instruction.

When operating in external debug mode or debug wait mode, the occurrence of an ICMP debug event is recorded in DBSR[ICMP] and causes the processor to enter the stop state and cease processing instructions. The program counter will contain the address of the instruction which would have executed next, had the ICMP debug event not occurred. Note that if the instruction whose completion caused the ICMP debug event was a branch instruction (and the branch conditions were satisfied), then upon entering the stop state the program counter will contain the target of the branch, and not the address of the instruction that is sequentially after the branch. Similarly, if the ICMP debug event is caused by the execution of a return (**rfi**, **rfci,** or **rfmci**) instruction, then upon entering the stop state the program counter will contain the address being *returned to*, and not the address of the instruction which is sequentially after the return instruction.

When operating in internal debug mode with Debug interrupts enabled (MSR[DE] = 1), the occurrence of an ICMP debug event is recorded in DBSR[ICMP] and a Debug interrupt will occur with CSRR0 set to the address of the instruction which would have executed next, had the ICMP debug event not occurred. Note that there is a special case of MSR[DE] = 1 at the time of the execution of the instruction causing the ICMP debug event, but that instruction itself sets MSR[DE] to 0. This special case is described in more detail in *Debug Interrupt* on page 278, in the subsection on the setting of CSRR0.

When operating in internal debug mode (and not also in external debug mode nor debug wait mode) with Debug interrupts disabled (MSR[DE] = 0), then ICMP debug events cannot occur. Since the code at the beginning of the critical class interrupt handlers (including the Debug interrupt itself) must execute at least temporarily with MSR[DE] = 0, there would be no way to avoid causing additional ICMP debug events and setting DBSR[IDE], if ICMP debug events were allowed to occur under these conditions.

The PPC465 does not support the use of the ICMP debug event when operating in trace mode. Software must not enable ICMP debug events unless one of the other debug modes is enabled as well.

12.4.8 Interrupt (IRPT) Debug Event

IRPT debug events occur when IRPT debug events are enabled (DBCR0[IRPT] = 1) and an interrupt occurs.

When operating in external debug mode or debug wait mode, the occurrence of an IRPT debug event is recorded in DBSR[IRPT] and causes the processor to enter the stop state and cease processing instructions. The program counter will contain the address of the instruction which would have executed next, had the IRPT debug event not occurred. Since the IRPT debug event is caused by the occurrence of an interrupt, by definition this address is that of the first instruction of the interrupt handler for the interrupt type which caused the IRPT debug event.

When operating in internal debug mode with external debug mode and debug wait mode both disabled (and regardless of the value of MSR[DE]), an IRPT debug event can only occur due to a non-critical class interrupt. Critical class interrupts (Machine Check, Critical Input, Watchdog Timer, and Debug interrupts) cannot cause IRPT debug events in internal debug mode (unless also in external debug mode or debug wait mode), as otherwise the Debug interrupt which would occur as the result of the IRPT debug event would by necessity always be imprecise, since the critical class interrupt which would be causing the IRPT debug event would itself be causing MSR[DE] to be set to 0.

For a non-critical class interrupt which is causing an IRPT debug event while internal debug mode is enabled and external debug mode and debug wait mode are both disabled, the occurrence of the IRPT debug event is recorded in DBSR[IRPT]. If MSR[DE] is 1 at the time of the IRPT debug event, then a Debug interrupt occurs with CSRR0 set to the address of the instruction which would have executed next, had the IRPT debug event not occurred. Since the IRPT debug event is caused by the occurrence of some other interrupt, by definition this address is that of the first instruction of the interrupt handler for the interrupt type which caused the IRPT debug event. If MSR[DE] is 0 at the time of the IRPT debug event, then the Imprecise Debug Event (IDE) field of the DBSR is also set and a Debug interrupt does not occur immediately. Instead, instruction execution continues, and a Debug interrupt will occur if and when MSR[DE] is set to 1, thereby enabling Debug interrupts, assuming software has not cleared the IRPT debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the Debug interrupt handler software may query the DBSR[IDE] field to determine that the Debug interrupt has occurred imprecisely.

When operating in trace mode, the occurrence of an IRPT debug event is simply recorded in DBSR[IRPT] and is indicated over the trace interface, and instruction execution continues.

12.4.9 Unconditional Debug Event (UDE)

UDE debug events occur when a debug tool asserts the unconditional debug event request via the JTAG interface. The UDE debug event is the only event which does not have a corresponding enable field in DBCR0.

When operating in external debug mode or debug wait mode, the occurrence of a UDE debug event is recorded in DBSR[UDE] and causes the processor to enter the stop state and cease processing instructions. The program counter will contain the address of the instruction which would have executed next, had the UDE debug event not occurred. Similarly, when operating in internal debug mode with Debug interrupts enabled (MSR[DE] = 1), the occurrence of a UDE debug event is recorded in DBSR[UDE] and a Debug interrupt will occur with CSRR0 set to the address of the instruction which would have executed next, had the UDE debug event not occurred.

When operating in internal debug mode (and not also in external debug mode nor debug wait mode) with Debug interrupts disabled (MSR[DE] = 0), the occurrence of a UDE debug event will set DBSR[UDE], along with the Imprecise Debug Event (IDE) field of the DBSR. The Debug interrupt will not occur immediately. Instead, instruction execution continues, and a Debug interrupt will occur if and when MSR[DE] is set to 1, thereby enabling Debug interrupts, assuming software has not cleared the UDE debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the Debug interrupt handler software may query the DBSR[IDE] field to determine that the Debug interrupt has occurred imprecisely.

When operating in trace mode, the occurrence of a UDE debug event simply sets DBSR[UDE] and is indicated over the trace interface, and instruction execution continues.

12.4.10 Debug Event Summary

Table 12-4 summarizes each of the debug event types, and the effect of debug mode and MSR[DE] on their occurrence.

Table 12-4. Debug Event Summary

External Debug Mode	Debug Wait Mode	Internal Debug Mode	MSR DE	Debug Events								
				IAC	DAC	DVC	BRT	TRAP	RET	ICMP	IRPT	UDE
Enabled	—	—	—	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
—	Enabled	—	—	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Disabled	Disabled	Enabled	1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Note 1	Yes
Disabled	Disabled	Enabled	0	Note 2	Yes	Yes	No	Yes	Note 3	No	Note 1	Yes
Disabled	Disabled	Disabled	—	Yes	Yes	Yes	Yes	Yes	Yes	Note 4	yes	Yes

Table Notes

1. IRPT debug events may only occur for non-critical class interrupts when operating in internal debug mode with external debug mode and debug wait mode both disabled.
2. IAC debug events may not occur in internal debug mode with MSR[DE] = 0 and with external debug mode and debug wait mode both disabled, and the IAC mode set to range inclusive or range exclusive. They *may* occur if the IAC mode is set to exact.
3. RET debug events may not occur for **rfci** or **rfmci** instructions when operating in internal debug mode with MSR[DE] = 0 and with external debug mode and debug wait mode both disabled. They may only occur in this mode for the **rfi** instruction.
4. ICMP debug events are not permitted when operating in trace debug mode. Software must not enable ICMP debug events unless one of the other debug modes is enabled.

12.5 Debug Reset

Software can initiate an immediate reset operation by setting DBCR0[RST] to a non-zero value. The results of a reset operation within the PPC465 core are described in *Reset and Initialization* in the chip user's manual. The results of a reset operation on the rest of the chip and/or system is dependent on the particular type of reset operation (core, chip, or system reset), and on the particular chip and system implementation. See the chip user's manual for details.

12.6 Debug Timer Freeze

In order to maintain the semblance of "real time" operation while a system is being debugged, DBCR0[FT] can be set to 1, which will cause all of the timers within the PPC465 core to stop incrementing or decrementing for as long as a debug event bit is set in the DBSR, or until DBCR0[FT] is set to 0. See *Timer Facilities* on page 307 for more information on the operation of the PPC465 core timers.

12.7 Debug Registers

Various Special Purpose Registers (SPRs) are used to enable the debug modes, to configure and record debug events, and to communicate with debug tool hardware and software. These debug registers may be accessed either through software running on the processor or through the JTAG debug port of the PPC465.

Programming Note: It is the responsibility of software to synchronize the context of any changes to the debug facility registers. Specifically, when changing the contents of any of the debug facility registers, software must execute an **isync** instruction both *before* and *after* the changes to these registers, to ensure that all preceding instructions use the *old* values of the registers, and that all succeeding instructions use the *new* values. In addition, when changing any of the debug facility register fields related to the DAC and/or DVC debug events, software must execute an **msync** instruction *before* making the changes, to ensure that all storage accesses complete using the *old* context of these register fields.

12.7.1 Debug Control Register 0 (DBCR0)

DBCR0 is an SPR that is used to enable debug modes and events, reset the processor, and control timer operation when debugging. DBCR0 can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 12-2. Debug Control Register 0 (DBCR0)

0	EDM	External Debug Mode 0 Disable external debug mode. 1 Enable external debug mode.	
1	IDM	Internal Debug Mode 0 Disable internal debug mode. 1 Enable internal debug mode.	
2:3	RST	Reset 00 No action 01 Core reset 10 Chip reset 11 System reset	**Attention:** Writing 01, 10, or 11 to this field causes a processor reset to occur.
4	ICMP	Instruction Completion Debug Event 0 Disable instruction completion debug event. 1 Enable instruction completion debug event.	Instruction completions do not cause instruction completion debug events if MSR[DE] = 0 in internal debug mode, unless also in external debug mode or debug wait mode.
5	BRT	Branch Taken Debug Event 0 Disable branch taken debug event. 1 Enable branch taken debug event.	Taken branches do not cause branch taken debug events if MSR[DE] = 0 in internal debug mode, unless also in external debug mode or debug wait mode.
6	IRPT	Interrupt Debug Event 0 Disable interrupt debug event. 1 Enable interrupt debug event.	Critical interrupts do not cause interrupt debug events in internal debug mode, unless also in external debug mode or debug wait mode.
7	TRAP	Trap Debug Event 0 Disable trap debug event. 1 Enable trap debug event.	
8	IAC1	Instruction Address Compare (IAC) 1 Debug Event 0 Disable IAC 1 debug event. 1 Enable IAC 1 debug event.	
9	IAC2	IAC 2 Debug Event 0 Disable IAC 2 debug event. 1 Enable IAC 2 debug event.	
10	IAC3	IAC 3 Debug Event 0 Disable IAC 3 debug event. 1 Enable IAC 3 debug event.	

11	IAC4	IAC 4 Debug Event 0 Disable IAC 4 debug event. 1 Enable IAC 4 debug event.	
12	DAC1R	Data Address Compare (DAC) 1 Read Debug Event 0 Disable DAC 1 read debug event. 1 Enable DAC 1 read debug event.	
13	DAC1W	DAC 1 Write Debug Event 0 Disable DAC 1 write debug event. 1 Enable DAC 1 write debug event.	
14	DAC2R	DAC 2 Read Debug Event 0 Disable DAC 2 read debug event. 1 Enable DAC 2 read debug event.	
15	DAC2W	DAC 2 Write Debug Event 0 Disable DAC 2 write debug event. 1 Enable DAC 2 write debug event.	
16	RET	Return Debug Event 0 Disable return (**rfi**/**rfci**/**rfmci**) debug event. 1 Enable return (**rfi**/**rfci**/**rfmci**) debug event.	**rfci**/**rfmci** does not cause a return debug event if MSR[DE] = 0 in internal debug mode, unless also in external debug mode or debug wait mode.
17:30		Reserved	
31	FT	Freeze timers on debug event 0 Timers are not frozen. 1 Freeze timers if a DBSR field associated with a debug event is set.	

12.7.2 Debug Control Register 1 (DBCR1)

DBCR1 is an SPR that is used to configure IAC debug events. DBCR1 can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 12-3. Debug Control Register 1 (DBCR1)

0:1	IAC1US	Instruction Address Compare (IAC) 1 User/Supervisor 00 Both 01 Reserved 10 Supervisor only (MSR[PR] = 0) 11 User only (MSR[PR] = 1)	
2:3	IAC1ER	IAC 1 Effective/Real 00 Effective (MSR[IS] = don’t care) 01 Reserved 10 Virtual (MSR[IS] = 0) 11 Virtual (MSR[IS] = 1)	
4:5	IAC2US	IAC 2 User/Supervisor 00 Both 01 Reserved 10 Supervisor only (MSR[PR] = 0) 11 User only (MSR[PR] = 1)	
6:7	IAC2ER	IAC 2 Effective/Real 00 Effective (MSR[IS] = don’t care) 01 Reserved 10 Virtual (MSR[IS] = 0) 11 Virtual (MSR[IS] = 1)	

8:9	IAC12M	IAC 1/2 Mode 00 Exact match 01 Reserved 10 Range inclusive 11 Range exclusive	Match if address[0:29] = IAC 1/2[0:29]; two independent compares Match if IAC1 ≤ address < IAC2 Match if address < IAC1 OR address ≥ IAC2
10:14		Reserved	
15	IAC12AT	IAC 1/2 Auto-Toggle Enable 0 Disable IAC 1/2 auto-toggle 1 Enable IAC 1/2 auto-toggle	
16:17	IAC3US	IAC 3 User/Supervisor 00 Both 01 Reserved 10 Supervisor only (MSR[PR] = 0) 11 User only (MSR[PR] = 1)	
18:19	IAC3ER	IAC 3 Effective/Real 00 Effective (MSR[IS] = don’t care) 01 Reserved 10 Virtual (MSR[IS] = 0) 11 Virtual (MSR[IS] = 1)	
20:21	IAC4US	IAC 4 User/Supervisor 00 Both 01 Reserved 10 Supervisor only (MSR[PR] = 0) 11 User only (MSR[PR] = 1)	
22:23	IAC4ER	IAC 4 Effective/Real 00 Effective (MSR[IS] = don’t care) 01 Reserved 10 Virtual (MSR[IS] = 0) 11 Virtual (MSR[IS] = 1)	
24:25	IAC34M	IAC 3/4 Mode 00 Exact match 01 Reserved 10 Range inclusive 11 Range exclusive	Match if address[0:29] = IAC 3/4[0:29]; two independent compares Match if IAC3 ≤ address < IAC4 Match if address < IAC3 OR address ≥ IAC4
26:30		Reserved	
31	IAC34AT	IAC3/4 Auto-Toggle Enable 0 Disable IAC 3/4 auto-toggle 1 Enable IAC 3/4 auto-toggle	

12.7.3 Debug Control Register 2 (DBCR2)

DBCR2 is an SPR that is used to configure DAC and DVC debug events. DBCR2 can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 12-4. Debug Control Register 2 (DBCR2)

0:1	DAC1US	Data Address Compare (DAC) 1 User/Supervisor 00 Both 01 Reserved 10 Supervisor only (MSR[PR] = 0) 11 User only (MSR[PR] = 1)	
2:3	DAC1ER	DAC 1 Effective/Real 00 Effective (MSR[DS] = don't care) 01 Reserved 10 Virtual (MSR[DS] = 0) 11 Virtual (MSR[DS] = 1)	
4:5	DAC2US	DAC 2 User/Supervisor 00 Both 01 Reserved 10 Supervisor only (MSR[PR] = 0) 11 User only (MSR[PR] = 1)	
6:7	DAC2ER	DAC 2 Effective/Real 00 Effective (MSR[DS] = don't care) 01 Reserved 10 Virtual (MSR[DS] = 0) 11 Virtual (MSR[DS] = 1)	
8:9	DAC12M	DAC 1/2 Mode 00 Exact match 01 Address bit mask 10 Range inclusive 11 Range exclusive	Match if address[0:31] = DAC 1/2[0:31]; two independent compares Match if address = DAC1; only compare bits corresponding to 1 bits in DAC2 Match if DAC1 $\leq$ address < DAC2 Match if address < DAC1 OR address $\geq$ DAC2
10	DAC12A	DAC 1/2 Asynchronous 0 Debug interrupt caused by DAC1/2 exception will be synchronous 1 Debug interrupt caused by DAC1/2 exception will be asynchronous	
11		Reserved	
12:13	DVC1M	Data Value Compare (DVC) 1 Mode 00 Reserved 01 AND all bytes enabled by DVC1BE 10 OR all bytes enabled by DVC1BE 11 AND-OR pairs of bytes enabled by DVC1BE	(0 AND 1) OR (2 AND 3)
14:15	DVC2M	DVC 2 Mode 00 Reserved 01 AND all bytes enabled by DVC2BE 10 OR all bytes enabled by DVC2BE 11 AND-OR pairs of bytes enabled by DVC2BE	(0 AND 1) OR (2 AND 3)
16:19		Reserved	
20:23	DVC1BE	DVC 1 Byte Enables 0:3	
24:27		Reserved	
28:31	DVC2BE	DVC 2 Byte Enables 0:3	

12.7.4 Debug Status Register (DBSR)

The DBSR contains status on debug events as well as information on the type of the most recent reset. The status bits are set by the occurrence of debug events, while the reset type information is updated upon the occurrence of any of the three reset types.

The DBSR is read into a GPR using **mfspr**. Clearing the DBSR is performed using **mtspr** by placing a 1 in the GPR source register in all bit positions which are to be cleared in the DBSR, and a 0 in all other bit positions. The data written from the GPR to the DBSR is not direct data, but a mask. A 1 clears the bit and a 0 leaves the corresponding DBSR bit unchanged.

Figure 12-5. Debug Status Register (DBSR)

0	IDE	Imprecise Debug Event 0 Debug event occurred while MSR[DE] = 1 1 Debug event occurred while MSR[DE] = 0	For synchronous debug events in internal debug mode, this field indicates whether the corresponding Debug interrupt occurs precisely or imprecisely
1	UDE	Unconditional Debug Event 0 Event didn't occur 1 Event occurred	
2:3	MRR	Most Recent Reset 00 No reset has occurred since this field was last cleared by software. 01 Core reset 10 Chip reset 11 System reset	This field is set upon any processor reset to a value indicating the type of reset.
4	ICMP	Instruction Completion Debug Event 0 Event didn't occur 1 Event occurred	
5	BRT	Branch Taken Debug Event 0 Event didn't occur 1 Event occurred	
6	IRPT	Interrupt Debug Event 0 Event didn't occur 1 Event occurred	
7	TRAP	Trap Debug Event 0 Event didn't occur 1 Event occurred	
8	IAC1	IAC 1 Debug Event 0 Event didn't occur 1 Event occurred	
9	IAC2	IAC 2 Debug Event 0 Event didn't occur 1 Event occurred	
10	IAC3	IAC 3 Debug Event 0 Event didn't occur 1 Event occurred	
11	IAC4	IAC 4 Debug Event 0 Event didn't occur 1 Event occurred	
12	DAC1R	DAC 1 Read Debug Event 0 Event didn't occur 1 Event occurred	

13	DAC1W	DAC 1 Write Debug Event 0 Event didn't occur 1 Event occurred	
14	DAC2R	DAC 2 Read Debug Event 0 Event didn't occur 1 Event occurred	
15	DAC2W	DAC 2 Write Debug Event 0 Event didn't occur 1 Event occurred	
16	RET	Return Debug Event 0 Event didn't occur 1 Event occurred	
17:29		Reserved	
30	IAC12ATS	IAC 1/2 Auto-Toggle Status 0 Range is not reversed from value specified in DBCR1[IAC12M] 1 Range is reversed from value specified in DBCR1[IAC12M]	
31	IAC34ATS	IAC 3/4 Auto-Toggle Status 0 Range is not reversed from value specified in DBCR1[IAC34M] 1 Range is reversed from value specified in DBCR1[IAC34M]	

12.7.5 Instruction Address Compare Registers (IAC1:IAC4)

The four IAC registers specify the addresses upon which IAC debug events should occur. Each of the IAC registers can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 12-6. Instruction Address Compare Registers (IAC1:IAC4)

0:29		Instruction Address Compare (IAC) word address	
30:31		Reserved	

12.7.6 Data Address Compare Registers (DAC1:DAC2)

The two DAC registers specify the addresses upon which DAC (and/or DVC) debug events should occur. Each of the DAC registers can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 12-7. Data Address Compare Registers (DAC1:DAC2)

0:31		Data Address Compare (DAC) byte address	

12.7.7 Data Value Compare Registers (DVC1–DVC2)

The DVC registers specify the data values upon which DVC debug events should occur. Each of the DVC registers can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 12-8. Data Value Compare Registers (DVC1:DVC2)

0:31		Data value to compare	

12.7.8 Debug Data Register (DBDR)

The DBDR can be used for communication between software running on the processor and debug tool hardware and software. The DBDR can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Figure 12-9. Debug Data Register (DBDR)

0:31		Debug Data	

13. Instruction Set

Descriptions of the PPC465 instructions follow. Each description contains the following elements:

- Instruction names (mnemonic and full)
- Instruction syntax
- Instruction format diagram
- Pseudocode description
- Prose description
- Registers altered

Where appropriate, instruction descriptions list invalid instruction forms and exceptions, and provide programming notes.

Table 13-1 summarizes the PPC465 instruction set by category.

Table 13-1. Instruction Categories

Category	Sub-Category	Instruction Types
Integer	Integer Storage Access	load, store
	Integer Arithmetic	add, subtract, multiply, divide, negate
	Integer Logical	and, andc, or, orc, xor, nand, nor, xnor, extend sign, count leading zeros
	Integer Compare	compare, compare logical
	Integer Trap	trap
	Integer Rotate	rotate and insert, rotate and mask
	Integer Shift	shift left, shift right, shift right algebraic
	Integer Select	select operand
Branch		branch, branch conditional, branch to link, branch to count
Processor Control	Condition Register Logical	crand, crandc, cror, crorc, crnand, crnor, crxor, crxnor
	Register Management	move to/from SPR, move to/from DCR, move to/from MSR, write to external interrupt enable bit, move to/from CR
	System Linkage	system call, return from interrupt, return from critical interrupt, return from machine check interrupt
	Processor Synchronization	instruction synchronize
Storage Control	Cache Management	data allocate, data invalidate, data touch, data zero, data flush, data store, instruction invalidate, instruction touch
	TLB Management	read, write, search, synchronize
	Storage Synchronization	memory synchronize, memory barrier
Allocated	Allocated Arithmetic	multiply-accumulate, negative multiply-accumulate, multiply half word
	Allocated Logical	detect left-most zero byte
	Allocated Cache Management	data congruence-class invalidate, instruction congruence-class invalidate
	Allocated Cache Debug	data read, instruction read

13.1 Instruction Set Portability

To support embedded real-time applications, the PPC465 implements the defined instruction set of the Book-E Enhanced PowerPC Architecture, with the exception of those operations which are defined for 64-bit implementations only, and those which are defined as floating-point operations. Support for the floating-point operations is provided via the auxiliary processor interface, while the 64-bit operations are not supported at all. See *Instruction Classes* on page 53 for more information on the support for defined instructions within the PPC465.

The PPC465 also implements a number of instructions that are not part of PowerPC Book-E architecture, but are included as part of the PPC465. Architecturally, they are considered allocated instructions, as they use opcodes which are within the allocated class of instructions, which the PowerPC Book-E architecture identifies as being available for implementation-dependent and/or application-specific purposes. However, all of the allocated instructions which are implemented within the PPC465 are “standard” for PowerPC 400 Series family of embedded controllers, and are not unique to the PPC465.

The allocated instructions implemented within the PPC465 are divided into four sub-categories, and are shown in *Table 13-2*. Programs using these instructions may not be portable to other PowerPC Book-E implementations.

Table 13-2. Allocated Instructions

Arithmetic			Logical	Cache Management	Cache Debug
Multiply-Accumulate	**Negative Multiply-Accumulate**	**Multiply Half word**			
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**[.]	**dlmzb**[.]	**dccci** **iccci**	**dcread** **icread**

13.2 Instruction Formats

For more detailed information about instruction formats, including a summary of instruction field usage and instruction format diagrams for the PPC465, see *Section A.1 Instruction Formats* on page 609.

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 field as well. The remaining instruction bits contain additional fields. All instruction fields belong to one of the following categories:

- Defined

  These instruction fields 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 specifiers and immediate values, each of which may contain any one of a number of values. The instruction format diagrams specify the field names of 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 specified value, the instruction is illegal and an Illegal Instruction exception type Program interrupt 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 PPC465 will execute all invalid instruction forms without causing an Illegal Instruction exception.

13.3 Pseudocode

The pseudocode that appears in the instruction descriptions provides a semi-formal language for describing instruction operations.

The pseudocode uses the following notation:

+	Twos complement addition
%	Remainder of an integer division; (33 % 32) = 1.
$\overset{u}{<}, \overset{u}{>}$	Unsigned comparison relations
(GPR(r))	The contents of GPR r, where $0 \leq r \leq 31$.
(RA|0)	The contents of the register RA or 0, if the RA field is 0.
(Rx)	The contents of a GPR, where *x* is A, B, S, or T
0bn	A binary number
0xn	A hexadecimal number
<, >	Signed comparison relations
=	Assignment
=, ≠	Equal, not equal relations
CEIL(x)	Least integer ≥ x.
CIA	Current instruction address; the 32-bit address of the instruction being described by a sequence of pseudocode. This address is used to set the next instruction address (NIA). Does not correspond to any architected register.
DCR(DCRN)	A Device Control Register (DCR) specified by the DCRF field in an **mfdcr** or **mtdcr** instruction
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.
EXTS(x)	The result of extending *x* on the left with sign bits.
FLD	An instruction or register field
FLD_b	A bit in a named instruction or register field
$FLD_{b,b,\ldots}$	A list of bits, by number or name, in a named instruction or register field
$FLD_{b:b}$	A range of bits in a named instruction or register field
GPR(r)	General Purpose Register (GPR) r, where $0 \leq r \leq 31$.
GPRs	RA, RB, . . .
MASK(MB,ME)	Mask having 1s in positions MB through ME (wrapping if MB > ME) and 0s elsewhere.
MS(addr, n)	The number of bytes represented by *n* at the location in main storage represented by *addr.*

NIA	Next instruction address; the 32-bit address of the next instruction to be executed. In pseudocode, a successful branch is indicated by assigning a value to NIA. For instructions that do not branch, the NIA is CIA +4.
PC	Program counter.
REG[FLD, FLD . . .]	A list of fields in a named register
REG[FLD:FLD]	A range of fields in a named register
REG[FLD]	A field in a named register
REG_b	A bit in a named register
$\text{REG}_{b,b,\ldots}$	A list of bits, by number or name, in a named register
$\text{REG}_{b:b}$	A range of bits in a named register
RESERVE	Reserve bit; indicates whether a process has reserved a block of storage.
ROTL((RS),n)	Rotate left; the contents of RS are shifted left the number of bits specified by *n*.
SPR(SPRN)	A Special Purpose Register (SPR) specified by the SPRF field in an **mfspr** or **mtspr** instruction
$c_{0:3}$	A four-bit object used to store condition results in compare instructions.
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.
if...then...else...	Conditional execution; if *condition* then *a* else *b*, where *a* and *b* represent one or more pseudocode statements. Indenting indicates the ranges of *a* and *b*. If *b* is null, the else does not appear.
instruction(EA)	An instruction operating on a data or instruction cache block associated with an EA.
leave	Leave innermost do loop or do loop specified in a leave statement.
n	A decimal number
^{n}b	The bit or bit value *b* is replicated *n* times.
xx	Bit positions which are don't-cares.
||	Concatenation
$\times$	Multiplication
$\div$	Division yielding a quotient
$\oplus$	Exclusive-OR (XOR) logical operator
–	Twos complement subtraction, unary minus
$\neg$	NOT logical operator
$\wedge$	AND logical operator
$\vee$	OR logical operator

13.3.1 Operator Precedence

Table 13-3 lists the pseudocode operators and their associativity in descending order of precedence:

Table 13-3. Operator Precedence

Operators	Associativity
REG_b, REG[FLD], function evaluation	Left to right
${}^{n}b$	Right to left
$\neg$, – (unary minus)	Right to left
$\times$, $\div$	Left to right
+, –	Left to right
$\|$	Left to right
=, $\neq$, <, >, $\overset{u}{<}$, $\overset{u}{>}$	Left to right
$\wedge$, $\oplus$	Left to right
$\vee$	Left to right
$\leftarrow$	None

13.4 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). Some instructions also change other registers, but the details of the changes are not included in the instruction descriptions. Common examples of these kinds of register changes include the Condition Register (CR) and the Integer Exception Register (XER). For discussion of the CR, see *Condition Register (CR)* on page 66. For discussion of the XER, see *Integer Exception Register (XER)* on page 70.

13.5 Alphabetical Instruction Listing

The following pages list the instructions, both defined and allocated, which are implemented within the PPC465.

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) ← (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] if Rc contains 1
- XER[SO, OV] if OE contains 1

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] if Rc contains 1
- XER[SO, OV] if OE contains 1

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) $\leftarrow$ (RA) + (RB) + XER[CA]
if (RA) + (RB) + 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, 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] if Rc contains 1
- XER[SO, OV] if OE contains 1

addi RT, RA, IM

14	RT	RA	IM
0	6	11	16 … 31

(RT) ← (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.

Table 13-4. 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

```
$(RT) \leftarrow (RA) + EXTS(IM)$
if $(RA) + EXTS(IM) \overset{u}{>} 2^{32} - 1$ then
    $XER[CA] \leftarrow 1$
else
    $XER[CA] \leftarrow 0$
```

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]

Table 13-5. 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**	

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]

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.**.

Table 13-6. 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
```

Table 13-7. 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) $\leftarrow$ (RA) + XER[CA] + (–1)
if (RA) + XER[CA] + 0xFFFF FFFF $\overset{u}{>}$ $2^{32} – 1$ then
   XER[CA] $\leftarrow$ 1
else
   XER[CA] $\leftarrow$ 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] if Rc contains 1
- XER[SO, OV] if OE contains 1

Invalid Instruction Forms

- Reserved fields

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) ← (RA) + XER[CA]
if (RA) + XER[CA] $\overset{u}{>}$ $2^{32}$ – 1 then
   XER[CA] ← 1
else
   XER[CA] ← 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] if Rc contains 1
- XER[SO, OV] if OE contains 1

Invalid Instruction Forms

- Reserved fields

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] if Rc contains 1

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] if Rc contains 1

andi. RA, RS, IM

28	**RS**	**RA**	**IM**	
0	6	11	16	31

$(RA) \leftarrow (RS) \land (^{16}0 \parallel IM)$

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]

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.**.

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]

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.**.

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 ← $\text{target}_{6:29}$
    NIA ← EXTS(LI || $^{2}0$)
else
    LI ← $(\text{target} - \text{CIA})_{6:29}$
    NIA ← CIA + EXTS(LI || $^{2}0$)
if LK = 1 then
    (LR) ← CIA + 4
PC ← NIA
```

The next instruction address (NIA) is the effective address of the branch target. 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 the current instruction address (CIA). If the AA field contains 1, the base address is 0.

Instruction execution resumes with the instruction at the NIA.

If the LK field contains 1, then (CIA + 4) is placed into the LR.

Registers Altered

- LR if LK contains 1

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 $BO_2$ = 0 then
    CTR $\leftarrow$ CTR – 1
if ($BO_2$ = 1 $\vee$ ((CTR = 0) = $BO_3$)) $\wedge$ ($BO_0$ = 1 $\vee$ ($CR_{BI}$ = $BO_1$)) then
    if AA = 1 then
        BD $\leftarrow$ $target_{16:29}$
        NIA $\leftarrow$ EXTS(BD || $^{2}0$)
    else
        BD $\leftarrow$ $(target – CIA)_{16:29}$
        NIA $\leftarrow$ CIA + EXTS(BD || $^{2}0$)
else
    NIA $\leftarrow$ CIA + 4
if LK = 1 then
    (LR) $\leftarrow$ CIA + 4
PC $\leftarrow$ NIA
```

If BO_2 contains 0, the CTR decrements, and the decremented value is tested for 0 as part of the branch condition. In this case, BO_3 indicates whether the test for 0 must be true or false in order for the branch to be taken. If BO_2 contains 1, then the CTR is neither decremented nor tested as part of the branch condition.

If BO_0 contains 0, then the CR bit specified by the BI field is compared to BO_1 as part of the branch condition. If BO_0 contains 1, then the CR is not tested as part of the branch condition, and the BI field is ignored.

The next instruction address (NIA) is either the effective address of the branch target, or the address of the instruction after the branch, depending on whether the branch is taken or not. The branch target address 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 the current instruction address (CIA). If the AA field contains 1, the base address is 0.

BO_4 affects branch prediction, a performance-improvement feature. See *Branch Prediction* on page 65 for a complete discussion.

Instruction execution resumes with the instruction at the NIA.

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

Table 13-8. Extended Mnemonics for bc, bca, bcl, bcla

Mnemonic	Operands	Function	Other Registers Altered
bdnz	target	Decrement CTR; branch if CTR $\neq$ 0. *Extended mnemonic for* **bc 16,0,target**	
bdnza	target	*Extended mnemonic for* **bca 16,0,target**	
bdnzl	target	*Extended mnemonic for* **bcl 16,0,target**	(LR) $\leftarrow$ CIA + 4.
bdnzla	target	*Extended mnemonic for* **bcla 16,0,target**	(LR) $\leftarrow$ CIA + 4.
bdnzf	cr_bit, target	Decrement CTR. Branch if CTR $\neq$ 0 AND CR_{cr_bit} = 0. *Extended mnemonic for* **bc 0,cr_bit,target**	
bdnzfa	cr_bit, target	*Extended mnemonic for* **bca 0,cr_bit,target**	
bdnzfl	cr_bit, target	*Extended mnemonic for* **bcl 0,cr_bit,target**	(LR) $\leftarrow$ CIA + 4.
bdnzfla	cr_bit, target	*Extended mnemonic for* **bcla 0,cr_bit,target**	(LR) $\leftarrow$ CIA + 4.
bdnzt	cr_bit, target	Decrement CTR. Branch if CTR $\neq$ 0 AND CR_{cr_bit} = 1. *Extended mnemonic for* **bc 8,cr_bit,target**	
bdnzta	cr_bit, target	*Extended mnemonic for* **bca 8,cr_bit,target**	
bdnztl	cr_bit, target	*Extended mnemonic for* **bcl 8,cr_bit,target**	(LR) $\leftarrow$ CIA + 4.
bdnztla	cr_bit, target	*Extended mnemonic for* **bcla 8,cr_bit,target**	(LR) $\leftarrow$ CIA + 4.
bdz	target	Decrement CTR; branch if CTR = 0. *Extended mnemonic for* **bc 18,0,target**	
bdza	target	*Extended mnemonic for* **bca 18,0,target**	
bdzl	target	*Extended mnemonic for* **bcl 18,0,target**	(LR) $\leftarrow$ CIA + 4.
bdzla	target	*Extended mnemonic for* **bcla 18,0,target**	(LR) $\leftarrow$ 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**	
bdzfa	cr_bit, target	*Extended mnemonic for* **bca 2,cr_bit,target**	
bdzfl	cr_bit, target	*Extended mnemonic for* **bcl 2,cr_bit,target**	(LR) $\leftarrow$ CIA + 4.
bdzfla	cr_bit, target	*Extended mnemonic for* **bcla 2,cr_bit,target**	(LR) $\leftarrow$ CIA + 4.

Table 13-8. Extended Mnemonics for bc, bca, bcl, bcla (continued)

Mnemonic	Operands	Function	Other Registers Altered
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) $\leftarrow$ CIA + 4.
bdztla		*Extended mnemonic for* **bcla 10,cr_bit,target**	(LR) $\leftarrow$ 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) $\leftarrow$ CIA + 4.
beqla		*Extended mnemonic for* **bcla 12,4∗cr_field+2,target**	(LR) $\leftarrow$ 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
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

Table 13-8. Extended Mnemonics for bc, bca, bcl, bcla (continued)

Mnemonic	Operands	Function	Other Registers Altered
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.
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.

Table 13-8. Extended Mnemonics for bc, bca, bcl, bcla (continued)

Mnemonic	Operands	Function	Other Registers Altered
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.
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 $(BO_0 = 1 \vee (CR_{BI} = BO_1))$ then
    NIA $\leftarrow CTR_{0:29} \| {}^2 0$
else
    NIA $\leftarrow$ CIA + 4
if LK = 1 then
    (LR) $\leftarrow$ CIA + 4
PC $\leftarrow$ NIA
```

If BO_0 contains 0, then the CR bit specified by the BI field is compared to BO_1 as part of the branch condition. If BO_0 contains 1, then the CR is not tested as part of the branch condition, and the BI field is ignored.

The next instruction address (NIA) is either the effective address of the branch target, or the address of the instruction after the branch, depending on whether the branch is taken or not. The branch target address is formed by concatenating two 0-bits to the right of the 30 most significant bits of the CTR.

BO_4 affects branch prediction, a performance-improvement feature. See *Branch Prediction* on page 65 for a complete discussion.

Instruction execution resumes with the instruction at the NIA.

If the LK field contains 1, then (CIA + 4) is placed into the LR.

Registers Altered

- LR if LK contains 1

Invalid Instruction Forms

- Reserved fields
- If BO_2 contains 0, the instruction form is invalid, and the result of the instruction (in particular, the branch target address and whether or not the branch is taken) is undefined. The architecture does not permit the combination of decrementing the CTR as part of the branch condition, together with using the CTR as the branch target address.

Table 13-9. 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) $\leftarrow$ CIA + 4.

Table 13-9. 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 13-9. 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 ← CTR – 1
if $(BO_2 = 1 \vee ((CTR = 0) = BO_3)) \wedge (BO_0 = 1 \vee (CR_{BI} = BO_1))$ then
    NIA ← $LR_{0:29} \,||\, {}^{2}0$
else
    NIA ← CIA + 4
if LK = 1 then
    (LR) ← CIA + 4
PC ← NIA
```

If BO_2 contains 0, the CTR decrements, and the decremented value is tested for 0 as part of the branch condition. In this case, BO_3 indicates whether the test for 0 must be true or false in order for the branch to be taken. If BO_2 contains 1, then the CTR is neither decremented nor tested as part of the branch condition.

If BO_0 contains 0, then the CR bit specified by the BI field is compared to BO_1 as part of the branch condition. If BO_0 contains 1, then the CR is not tested as part of the branch condition, and the BI field is ignored.

The next instruction address (NIA) is either the effective address of the branch target, or the address of the instruction after the branch, depending on whether the branch is taken or not. The branch target address is formed by concatenating two 0-bits to the right of the 30 most significant bits of the LR.

BO_4 affects branch prediction, a performance-improvement feature. See *Branch Prediction* on page 65 for a complete discussion.

Instruction execution resumes with the instruction at the NIA.

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

Table 13-10. 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 13-10. Extended Mnemonics for bclr, bclrl (continued)

Mnemonic	Operands	Function	Other Registers Altered
bdnzlr		Decrement CTR. Branch if CTR $\neq$ 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 $\neq$ 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 $\neq$ 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 13-10. 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 13-10. 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

```
$c_{0:3} \leftarrow {}^40$
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]
$n \leftarrow$ BF
CR[CRn] $\leftarrow c_{0:3}$
```

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

PowerPC Book-E architecture defines this instruction as **cmp BF,L,RA,RB**, where L selects operand size for 64-bit implementations. For all 32-bit implementations, L = 0 is required (L = 1 is an invalid form); hence for the PPC465 use of the extended mnemonic **cmpw BF,RA,RB** is recommended.

Table 13-11. 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

PowerPC Book-E Architecture defines this instruction as **cmpi BF,L,RA,IM**, where L selects operand size for 64-bit implementations. For all 32-bit implementations, L = 0 is required (L = 1 is an invalid form); hence for the PPC465 use of the extended mnemonic **cmpwi BF,RA,IM** is recommended.

Table 13-12. 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 $\leftarrow$ BF
CR[CRn] $\leftarrow c_{0:3}$
```

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

PowerPC Book-E Architecture defines this instruction as **cmpl BF,L,RA,RB**, where L selects operand size for 64-bit implementations. For all 32-bit implementations, L = 0 is required (L = 1 is an invalid form); hence for PPC465 use of the extended mnemonic **cmplw BF,RA,RB** is recommended.

Table 13-13. 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) $\overset{u}{<}$ (${}^{16}0$ || IM) then $c_0 \leftarrow 1$
if (RA) $\overset{u}{>}$ (${}^{16}0$ || IM) then $c_1 \leftarrow 1$
if (RA) $=$ (${}^{16}0$ || 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

PowerPC Book-E Architecture defines this instruction as **cmpli BF,L,RA,IM**, where L selects operand size for 64-bit implementations. For all 32-bit implementations, L = 0 is required (L = 1 is an invalid form); hence for the PPC465 use of the extended mnemonic **cmplwi BF,RA,IM** is recommended.

Table 13-14. 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] if Rc contains 1

Invalid Instruction Forms

- Reserved fields

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.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR_{BT}

Invalid Instruction Forms

- Reserved fields

crandc BT, BA, BB

19	BT	BA	BB	129	▨
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \wedge \neg CR_{BB}$

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.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR_{BT}

Invalid Instruction Forms

- Reserved fields

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.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR_{BT}

Invalid Instruction Forms

- Reserved fields

Table 13-15. 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.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR_{BT}

Invalid Instruction Forms

- Reserved fields

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.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR_{BT}

Invalid Instruction Forms

- Reserved fields

Table 13-16. 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

$CR_{BT} \leftarrow CR_{BA} \lor CR_{BB}$

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.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR_{BT}

Invalid Instruction Forms

- Reserved fields

Table 13-17. 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.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR_{BT}

Invalid Instruction Forms

- Reserved fields

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.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR_{BT}

Invalid Instruction Forms

- Reserved fields

Table 13-18. 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

dcba is treated as a no-op by the PPC465.

dcbf RA, RB

31		RA	RB	86	
0	6	11	16	21	31

```
EA ← (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 memory page referenced by 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 265 for more information.

This instruction is considered a “store” with respect to data address compare (DAC) Debug exceptions. See *Debug Interrupt* on page 278 for more information.

This instruction may cause a Cache Locking type of Data Storage exception. See *Data Storage Interrupt* on page 265 for more information.

dcbi RA, RB

31		RA	RB	470	
0	6	11	16	21	31

EA ← (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 memory page referenced by 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 265 for more information.

This instruction is considered a “store” with respect to data address compare (DAC) Debug exceptions. See *Debug Interrupt* on page 278 for more information.

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 memory page referenced by 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 265 for more information.

This instruction is considered a “store” with respect to data address compare (DAC) Debug exceptions. See *Debug Interrupt* on page 278 for more information.

dcbt RA, RB

31		**RA**	**RB**	**278**	
0	6	11	16	21	31

```
EA ← (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 memory page referenced by 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 memory page referenced by the EA is marked as caching inhibited, no operation is performed.

This instruction is not allowed to cause Data Storage interrupts nor Data TLB Error interrupts. If execution of the instruction causes either of these types of exception, then no operation is performed, and no interrupt 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 265 for more information.

This instruction is considered a “load” with respect to data address compare (DAC) Debug exceptions. See *Debug Interrupt* on page 278 for more information.

dcbtst RA, RB

31		RA	RB	246	
0	6	11	16	21	31

```
EA ← (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 memory page referenced by the EA address is marked as cacheable, the data block is loaded into the data cache.

If the data block at the EA is in the data cache, or if the memory page referenced by the EA is marked as caching inhibited, no operation is performed.

This instruction is not allowed to cause Data Storage interrupts nor Data TLB Error interrupts. If execution of the instruction causes either of these types of exception, then no operation is performed, and no interrupt 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** is intended to bring a cache block into the data cache in a manner which will permit future instructions to store to that block efficiently. For example, in an implementation which supports the “MESI” cache coherency protocol, the block would be brought into the cache in “Exclusive” mode, allowing the block to be stored to without having to broadcast any coherency operations on the system bus. However, since the PPC465 does not support hardware-enforcement of multiprocessor coherency, there is no distinction between a block being brought in for a read or a write, and hence 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 265 for more information.

This instruction is considered a “load” with respect to data address compare (DAC) Debug exceptions. See *Debug Interrupt* on page 278 for more information.

dcbz RA, RB

31		RA	RB	1014	
0	6	11	16	21	31

```
EA ← (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 memory page referenced by the EA is marked as cacheable and non-write-through, the data in the cache block is set to 0 and marked as *dirty* (modified).

If the data block at the EA is not in the data cache and the memory page referenced by the EA is marked as cacheable and non-write-through, a cache block is established and set to 0 and marked as dirty. Note that nothing is read from main storage, as described in the programming note.

If the memory page referenced by the EA is marked as either write-through or as caching inhibited, 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 Data Machine Check exception could occur under these circumstances.

If **dcbz** is attempted to an EA in a memory page which is marked as caching inhibited or as write-through, the software alignment exception handler should emulate the instruction by storing zeros to the block referenced by the EA. The store instructions in the emulation software will cause main storage to be updated (and possibly the cache, if the EA is in a page marked as write-through).

Exceptions

An alignment exception occurs if the EA is marked as caching inhibited or as write-through.

This instruction is considered a “store” with respect to Data Storage exceptions. See *Data Storage Interrupt* on page 265 for more information.

This instruction is considered a “store” with respect to data address compare (DAC) Debug exceptions. See *Debug Interrupt* on page 278 for more information.

dccci RA, RB

31		RA	RB	454	
0	6	11	16	21	31

DCCCI

This instruction flash invalidates the entire data cache array. The RA and RB operands are not used; previous implementations used these operands to calculate an effective address (EA) which specified the particular block or blocks to be invalidated. The instruction form (including the specification of RA and RB operands) 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 data cache array before caching is enabled.

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

dcread   RT, RA, RB

31	RT	RA	RB	486	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
INDEX ← $EA_{17:26}$
WORD ← $EA_{27:29}$
(RT) ← (data cache data)[INDEX,WORD]
DCDBTRH ← (data cache tag high)[INDEX]
DCDBTRL ← (data cache tag low)[INDEX]
```

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.

$EA_{17:26}$ selects a line of tag and data from the data cache. $EA_{27:29}$ selects a word from the 8-word data portion of the selected cache line, and this word is read into register RT. $EA_{30:31}$ must be 0b00; if not, the value placed in register RT is undefined.

The tag portion of the selected cache line is read into the DCDBTRH and DCDBTRL registers, as follows:

Register[bit(s)]	Tag Field	Name	
DCDBTRH[0:23]	TRA	Tag Real Address	Bits 0:23 of the lower 32 bits of the 36-bit real address associated with this cache line
DCDBTRH[24]	V	Valid	The valid indicator for the cache line (1 indicates valid)
DCDBTRH[25:27]		reserved	Reserved fields are read as 0s
DCDBTRH[28:31]	TERA	Tag Extended Real Address	Upper 4 bits of the 36-bit real address associated with this cache line
DCDBTRL[0:23]		reserved	Reserved fields are read as 0s
DCDBTRL[24:27]	D	Dirty Indicators	The “dirty” (modified) indicators for each of the four doublewords in the cache line
DCDBTRL[28]	U0	U0 Storage Attribute	The U0 storage attribute for the memory page associated with this cache line
DCDBTRL[29]	U1	U1 Storage Attribute	The U0 storage attribute for the memory page associated with this cache line
DCDBTRL[30]	U2	U2 Storage Attribute	The U0 storage attribute for the memory page associated with this cache line
DCDBTRL[31]	U3	U3 Storage Attribute	The U0 storage attribute for the memory page associated with this cache line

This instruction can be used by a debug tool to determine the contents of the data cache, without knowing the specific addresses of the lines which are currently contained within the cache.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT
- DCDBTRH
- DCDBTRL

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged.

The PPC465 does not support the use of the **dcread** instruction when the data cache controller is still in the process of performing cache operations associated with previously executed instructions (such as line fills and line flushes). Also, the PPC465 does not automatically synchronize context between a **dcread** instruction and the subsequent **mfspr** instructions that read the results of the **dcread** instruction into GPRs. In order to guarantee that the **dcread** instruction operates correctly, and that the **mfspr** instructions obtain the results of the **dcread** instruction, a sequence such as the following must be used:

```
msync                        # ensure that all previous cache operations have completed
dcread      regT,regA,regB   # read cache information; the contents of GPR A and GPR B are
                             # added and the result used to specify a cache line index to be read;
                             # the data word is moved into GPR T and the tag information is read
                             # into DCDBTRH and DCDBTRL
isync                        # ensure dcread completes before attempting to read results
mfdcdbtrh   regD             # move high portion of tag into GPR D
mfdcdbtrl   regE             # move low portion of tag into GPR E
```

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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]$_{0:2}$ are undefined. Either invalid division operation sets XER[OV, SO] (and CR[CR0]$_3$ if Rc contains 1) to 1 if the OE field contains 1.

Registers Altered

- RT
- CR[CR0] 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.

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]$_{0:2}$ are also undefined. The invalid division operation also sets XER[OV, SO] (and CR[CR0]$_3$ if Rc contains 1) to 1 if the OE field contains 1.

Registers Altered

- RT
- CR[CR0] 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 0.

dlmzb	RA, RS, RB	Rc=0
dlmzb.	RA, RS, RB	Rc=1

31	RS	RA	RB	78	Rc
0	6	11	16	21	31

```
d ← (RS) || (RB)
i, x, y ← 0
do while (x < 8) ∧ (y = 0)
    x ← x + 1
    if $d_{i:i+7}$ = 0 then
        y ← 1
    else
        i ← i + 8
(RA) ← x
XER[TBC] ← x
if Rc = 1 then
    $CR[CR0]_3$ ←XER[SO]
    if y = 1 then
        if x < 5 then
            $CR[CR0]_{0:2}$ ← 0b010
        else
            $CR[CR0]_{0:2}$ ← 0b100
    else
        $CR[CR0]_{0:2}$ ← 0b001
```

The contents of registers RS and RB are concatenated to form an 8-byte operand. The operand is searched for the leftmost byte in which each bit is 0 (a 0-byte).

Bytes in the operand are numbered from left to right starting with 1. If a 0-byte is found, its byte number is placed into XER[TBC] and register RA. Otherwise, the number 8 is placed into XER[TBC] and register RA.

If the Rc field contains 1, XER[SO] is copied to $CR[CR0]_3$ and $CR[CR0]_{0:2}$ are updated as follows:

- If no 0-byte is found, $CR[CR0]_{0:2}$ is set to 0b001.
- If the leftmost 0-byte is in the first 4 bytes (in the RS register), $CR[CR0]_{0:2}$ is set to 0b010.
- If the leftmost 0-byte is in the last 4 bytes (in the RB register), $CR[CR0]_{0:2}$ is set to 0b100.

Registers Altered

- XER[TBC]
- RA
- CR[CR0] if Rc contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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] if Rc contains 1

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] if Rc contains 1

Invalid Instruction Forms

- Reserved fields

extsh	RA, RS	Rc=0
extsh.	RA, RS	Rc=1

31	**RS**	**RA**		**922**	**Rc**
0	6	11	16	21	31

(RA) ← $\text{EXTS(RS)}_{16:31}$

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] if Rc contains 1

Invalid Instruction Forms

- Reserved fields

icbi RA, RB

31		RA	RB	982	
0	6	11	16	21	31

```
EA ← (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.

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 management instructions use MSR[DS], not MSR[IS], as part of the virtual address. Also, the instruction cache on the PPC465 is “virtually-tagged”, which means that the EA is converted to a virtual address (VA), and the VA is compared against the cache tag field. See *Instruction Cache Synonyms* on page 133 for more information on the ramifications of virtual tagging on software.

Exceptions

Instruction Storage interrupts and Instruction TLB Error interrupts are associated with exceptions which occur during instruction *fetching*, not during instruction *execution*. *Execution* of instruction cache management instructions may cause Data Storage or Data TLB Error exceptions.

This instruction is considered a “load” with respect to Data Storage exceptions. See *Data Storage Interrupt* on page 265 for more information.

This instruction is considered a “load” with respect to data address compare (DAC) Debug exceptions. See *Debug Interrupt* on page 278 for more information.

This instruction may cause a Cache Locking type of Data Storage exception. See *Data Storage Interrupt* on page 265 for more information.

icbt RA, RB

31		RA	RB	22	
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 the memory page referenced by the EA is marked as cacheable, the instruction block is fetched into the instruction cache.

If the instruction block at the EA is in the instruction cache, or if the memory page referenced by the EA is marked as caching inhibited, no operation is performed.

If the memory page referenced by the EA is marked as “no-execute” for the current operating mode (user mode or supervisor mode, as specified by MSR[PR]), no operation is performed.

This instruction is not allowed to cause Data Storage interrupts nor Data TLB Error interrupts. If execution of the instruction causes either of these types of exception, then no operation is performed, and no interrupt occurs.

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 management instructions use MSR[DS], not MSR[IS], as part of the virtual address. Also, the instruction cache on the PPC465 is “virtually-tagged”, which means that the EA is converted to a virtual address (VA), and the VA is compared against the cache tag field. See *Instruction Cache Synonyms* on page 133 for more information on the ramifications of virtual tagging on software.

Exceptions

Instruction Storage interrupts and Instruction TLB Error interrupts are associated with exceptions which occur during instruction *fetching*, not during instruction *execution*. *Execution* of instruction cache management instructions may cause Data Storage or Data TLB Error exceptions, but are not allowed to cause the associated interrupt. Instead, if such an exception occurs, then no operation is performed.

This instruction is considered a “load” with respect to Data Storage exceptions. See *Data Storage Interrupt* on page 265 for more information.

This instruction is considered a “load” with respect to data address compare (DAC) Debug exceptions. See *Debug Interrupt* on page 278 for more information.

iccci RA, RB

31		**RA**	**RB**	**966**	
0	6	11	16	21	31

ICCCI

This instruction flash invalidates the entire instruction cache array. The RA and RB operands are not used; previous implementations used these operands to calculate an effective address (EA) which specified the particular block or blocks to be invalidated. The instruction form (including the specification of RA and RB operands) 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 instruction cache array before caching is enabled.

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

icread RA, RB

31		RA	RB	998	
0	6	11	16	21	31

EA ← (RA|0) + (RB)
INDEX ← $EA_{17:26}$
WORD ← $EA_{27:29}$
ICDBDR ← (instruction cache data)[INDEX,WORD]
ICDBTRH ← (instruction cache tag high)[INDEX]
ICDBTRL ← (instruction cache tag low)[INDEX]

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.

$EA_{17:26}$ selects a line of tag and data (instructions) from the instruction cache. $EA_{27:29}$ selects a 32-bit instruction from the 8-instruction data portion of the selected cache line, and this instruction is read into the ICDBDR. $EA_{30:31}$ are ignored, as are $EA_{0:16}$.

The tag portion of the selected cache line is read into the ICDBTRH and ICDBTRL registers, as follows:

Register[bit(s)]	Tag Field	Name	
ICDBTRH[0:23]	TEA	Tag Effective Address	Bits 0:23 of the 32-bit effective address associated with this cache line
ICDBTRH[24]	V	Valid	The valid indicator for the cache line (1 indicates valid)
ICDBTRH[25:31]		reserved	Reserved fields are read as 0s
ICDBTRL[0:21]		reserved	Reserved fields are read as 0s
ICDBTRL[22]	TS	Translation Space	The address space portion of the virtual address associated with this cache line.
ICDBTRL[23]	TD	Translation ID (TID) Disable	TID Disable field for the memory page associated with this cache line
ICDBTRL[24:31]	TID	Translation ID	TID field portion of the virtual address associated with this cache line

The instruction cache on PPC465 is “virtually-tagged”, which means that the tag field contains the virtual address, which consists of the TEA, TS, and TID fields. See *Memory Management* on page 219 for more information on the function of the TS, TD, and TID fields.

This instruction can be used by a debug tool to determine the contents of the instruction cache, without knowing the specific addresses of the lines which are currently contained within the cache.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- ICDBDR
- ICDBTRH

- ICDBTRL

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged.

The PPC465 does not automatically synchronize context between an **icread** instruction and the subsequent **mfspr** instructions which read the results of the **icread** instruction into GPRs. In order to guarantee that the **mfspr** instructions obtain the results of the **icread** instruction, a sequence such as the following must be used:

```
icread      regA,regB     # read cache information (the contents of GPR A and GPR B are
                          # added and the result used to specify a cache line index to be read)
isync                     # ensure icread completes before attempting to read results
mficdbdr    regC          # move instruction information into GPR C
mficdbtrh   regD          # move high portion of tag into GPR D
mficdbtrl   regE          # move low portion of tag into GPR E
```

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

isel RT, RA, RB, CRb

31	RT	RA	RB	CRb	15	
0	6	11	16	21	26	31

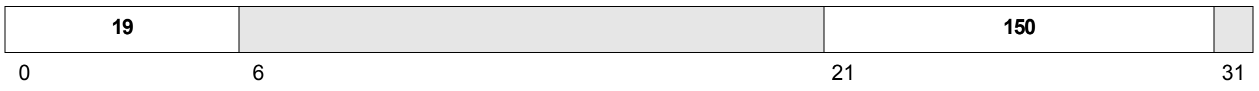
```
if CR[CRb] = 1 then
    (RT) ← (RA|0)
else
    (RT) ← (RB)
```

If CR[CRb] = 0, register RT is written with the contents of register RB.
If CR[CRb] = 1 and RA $\neq$ 0, register RT is written with the contents of register RA.
If CR[CRb] = 1 and RA = 0, register RT is written with 0.

Registers Altered

- RT

isync



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. Furthermore, all instructions preceding the **isync** are guaranteed to be unaffected by any context changes initiated by instructions after the **isync**.

No subsequent instructions are initiated by the processor until **isync** completes. Finally, execution of **isync** causes the processor to discard any prefetched instructions (prefetched from the cache, not instructions that are in the cache or on their way into the cache), with the effect that subsequent instructions are fetched and executed in the context established by the instructions preceding **isync**.

isync causes any caching inhibited instruction fetches from memory to be aborted and any data associated with them to be discarded. Cacheable instruction fetches from memory are not aborted however, as these should be handled by the **icbi** instructions which must precede the **isync** if software wishes to invalidate any cached instructions.

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 79.

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
    msync                        # wait until the data actually reaches the memory
    icbi      addr1              # invalidate the instruction if it is in the cache (or in the # process
of being fetched into the cache)
    msync                        # wait until the icbi completes
    isync                        # discard and refetch any instructions (including
                                 # possibly the instruction at addr1) which may have
                                 # already been fetched from the cache and be in the
                                 # pipeline after the isync
```

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

lbzu RT, D(RA)

35	RT	RA	D
0	6	11	16 … 31

```
$EA \leftarrow (RA|0) + EXTS(D)$
$(RA) \leftarrow EA$
$(RT) \leftarrow {}^{24}0 \parallel MS(EA,1)$
```

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

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

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

lha RT, D(RA)

42	RT	RA	D	
0	6	11	16	31

```
EA ← (RA|0) + EXTS(D)
(RT) ← 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

lhau

RT, D(RA)

43	**RT**	**RA**	**D**	
0	6	11	16	31

```
EA ← (RA|0) + EXTS(D)
(RA) ← EA
(RT) ← 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 when the RA field is 0 and is the contents of register RA otherwise. 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

lhaux RT, RA, RB

31	**RT**	**RA**	**RB**	**375**	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
(RA) ← EA
(RT) ← 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 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

lhax RT, RA, RB

31	RT	RA	RB	343	
0	6	11	16	21	31

EA ← (RA|0) + (RB)
(RT) ← 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

lhbrx RT, RA, RB

31	RT	RA	RB	790	
0	6	11	16	21	31

```
$EA \leftarrow (RA|0) + (RB)$
$(RT) \leftarrow {}^{16}0 \;||\; BYTE\_REVERSE(MS(EA,2))$
```

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 byte-reversed from the default byte ordering for the memory page referenced by the EA. 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

Programming Note

Byte ordering is generally controlled by the Endian (E) storage attribute (see *Memory Management* on page 219). The load byte reverse instructions provide a mechanism for data to be loaded from a memory page using the opposite byte ordering from that specified by the Endian storage attribute.

lhz RT, D(RA)

40	RT	RA	D	
0	6	11	16	31

$\text{EA} \leftarrow (\text{RA}|0) + \text{EXTS(D)}$
$(\text{RT}) \leftarrow {}^{16}0 \;||\; \text{MS(EA,2)}$

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

lhzu RT, D(RA)

41	RT	RA	D	
0	6	11	16	31

```
$EA \leftarrow (RA|0) + EXTS(D)$
$(RA) \leftarrow EA$
$(RT) \leftarrow {}^{16}0 \; || \; MS(EA,2)$
```

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 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

lhzux RT, RA, RB

31	**RT**	**RA**	**RB**	**311**	▨
0	6	11	16	21	31

```
$EA \leftarrow (RA|0) + (RB)$
$(RA) \leftarrow EA$
$(RT) \leftarrow {}^{16}0 \ || \ MS(EA,2)$
```

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 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

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

lmw RT, D(RA)

46	RT	RA	D
0	6	11	16 … 31

```
EA ← (RA|0) + EXTS(D)
r ← RT
do while r ≤ 31
   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).

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.

Programming Note

This instruction can be restarted, meaning that it could be interrupted after having already updated some of the target registers, and then re-executed from the beginning (after returning from the interrupt), in which case the registers which had already been loaded prior to the interrupt will be loaded a second time. Note that if RA is in the range of registers to be loaded (an invalid form; see above) and is also one of the registers which is loaded prior to the interrupt, then when the instruction is restarted the re-calculated EA will be incorrect, since RA will no longer contain the original base address. Hence the definition of this as an invalid form which software must avoid.

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
      GPR(r)) ← 0
   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}.

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

Programming Note

This instruction can be restarted, meaning that it could be interrupted after having already updated some of the target registers, and then re-executed from the beginning (after returning from the interrupt), in which case the registers which had already been loaded prior to the interrupt will be loaded a second time. Note that if RA is in the range of registers to be loaded (an invalid form; see above) and is also one of the registers which is loaded prior to the interrupt, then when the instruction is restarted the re-calculated EA will be incorrect, since RA will no longer contain the original base address. Hence the definition of this as an invalid form which software must avoid.

lswx RT, RA, RB

31	RT	RA	RB	533	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
CNT ← XER[TBC]
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
      GPR(r)) ← 0
   $GPR(r)_{i:i+7}$) ← MS(EA,1)
   i ← i + 8
   if i = 32 then
      i ← 0
   EA ← EA + 1
   n ← n – 1
```

An effective address (EA) is 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}.

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

This instruction can be restarted, meaning that it could be interrupted after having already updated some of the target registers, and then re-executed from the beginning (after returning from the interrupt), in which case the registers which had already been loaded prior to the interrupt will be loaded a second time. Note that if RA or RB is in the range of registers to be loaded (an invalid form; see above) and is also one of the registers which is loaded prior to the interrupt, then when the instruction is restarted the re-calculated EA will be incorrect, since the affected register will no longer contain the original base address or index. Hence the definition of these as invalid forms which software must avoid.

If XER[TBC] = 0, the contents of register RT are undefined and **lswx** is treated as a no-op. Furthermore, if the EA is such that a Data Storage, Data TLB Error, or Data Address Compare Debug exception occurs, **lswx** is treated as a no-op and no interrupt occurs as a result of the exception.

lwarx

RT, RA, RB

31	RT	RA	RB	20	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
RESERVE ← 1
(RT) ← 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

The **lwarx** and **stwcx.** instructions are typically 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 multiple processes. Only **lwarx** can set the reservation bit to 1. **stwcx.** sets the reservation bit to 0 upon its completion, whether or not **stwcx.** actually stored (RS) to memory. CR[CR0]$_2$ 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 the register as required
      stwcx.     # attempt to store the semaphore; reset reservation
      bne loop   # some other process intervened and cleared the reservation prior to the above
                 # stwcx.; try again
```

The PowerPC Book-E architecture specifies that the EA for the **lwarx** instruction must be word-aligned (that is, a multiple of 4 bytes); otherwise, the result is undefined. Although the PPC465 will execute **lwarx** regardless of the EA alignment, in order for the operation of the pairing of **lwarx** and **stwcx.** to produce the desired result, software must ensure that the EA for both instructions is word-aligned. This requirement is due to the manner in which misaligned storage accesses may be broken up into separate, aligned accesses by the PPC465.

PPC465 supports **lwarx** and **stwcx.** in a memory coherent manner for pages marked coherent (M=1) and write-through (WL1=1) pages, as specified in the following table, and generates an exception whenever a lwarx and stwcx. to W=1 or IL1=1 pages (Table 13-19 on page 430).

Table 13-19. Page Attributes and lwarx & stwcx. Operations

W	WL1	IL1D	IL2D	I	Operation
0	0	0	0	0	Coherency not supported
0	1	0	0	0	Fully supported
-	-	1	0	0	When L1D and L2 caches are in Write-through, or L1D or L2 caching is inhibited, lwarx and stwcx. are not supported even in the memory coherence required storage space, and therefore, a Storage synchronization exception will be generated. Refer to Shared Storage and Storage Attributes and Data Storage Interrupt sections of Book E.
-	-	0	1	0	
-	-	1	1	1	
1	-	-			

lwarx and **stwcx.** are expected to be word aligned when L2CCR1[L2COBE] bit is set indicating L2C being present. All misaligned **lwarx** and **stwcx.** operations result in an Alignment exception as permitted by PowerPC Book-E.

Caution: When the entire cache array is set to FAM (FAMES=10 for 256KB), the atomic access instructions, **lwarx** and **stwcx.** will not function correctly and instead will always fail.

For the **lwarx** instruction, if the address is not aligned on a word boundary, it is undefined whether or not a reservation has been set. if CCR1[L2COBE] is set to 0. However, if CCR1[L2COBE] is set to 1, **lwarx** instruction generates an alignment exception in this case.

Table 7-8, “Required Configuration Settings for Coherent Operation,” on page -174.

lwbrx RT, RA, RB

31	**RT**	**RA**	**RB**	**534**	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
(RT) ← BYTE_REVERSE(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 byte-reversed from the default byte ordering for the memory page referenced by the EA. 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

Programming Note

Byte ordering is generally controlled by the Endian (E) storage attribute (see *Memory Management* on page 219). The load byte reverse instructions provide a mechanism for data to be loaded from a memory page using the opposite byte ordering from that specified by the Endian storage attribute.

lwz RT, D(RA)

0	6	11	16	31
32	RT	RA	D	

```
EA ← (RA|0) + EXTS(D)
(RT) ← 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

lwzu RT, D(RA)

33	RT	RA	D	
0	6	11	16	31

```
EA ← (RA|0) + EXTS(D)
(RA) ← EA
(RT) ← 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 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

lwzux RT, RA, RB

31	RT	RA	RB	55	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
(RA) ← EA
(RT) ← 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 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

lwzx

RT, RA, RB

31	RT	RA	RB	23	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
(RT) ← 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

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

$\text{prod}_{0:31} \leftarrow (\text{RA})_{16:31} \times (\text{RB})_{0:15}$ signed

$\text{temp}_{0:32} \leftarrow \text{prod}_{0:31} + (\text{RT})$

$(\text{RT}) \leftarrow \text{temp}_{1:32}$

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 RT is updated with the low-order 32 bits of the signed sum.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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.

If the signed sum can be represented in 32 bits, then RT is updated with the low-order 32 bits of the signed sum.

If the signed sum cannot be represented in 32 bits, then RT is updated with a value which is “saturated” to the nearest representable value. That is, if the signed sum is less than -2^{31}, then RT is updated with -2^{31}. Likewise, if the signed sum is greater than $2^{31} - 1$, then RT is updated with $2^{31} - 1$.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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

$\text{prod}_{0:31} \leftarrow (\text{RA})_{16:31} \times (\text{RB})_{0:15}$ unsigned

$\text{temp}_{0:32} \leftarrow \text{prod}_{0:31} + (\text{RT})$

$(\text{RT}) \leftarrow (\text{temp}_{1:32} \vee {}^{32}\text{temp}_0)$

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.

If the unsigned sum can be represented in 32 bits, then RT is updated with the low-order 32 bits of the unsigned sum.

If the unsigned sum cannot be represented in 32 bits, then RT is updated with a value which is “saturated” to the maximum representable value of $2^{32} - 1$.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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 RT is updated with the low-order 32 bits of the unsigned sum.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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

$prod_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ 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 RT is updated with the low-order 32 bits of the signed sum.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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 $(RT) \leftarrow (RT_0 \,||\, ^{31}(\neg RT_0))$
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.

If the signed sum can be represented in 32 bits, then RT is updated with the low-order 32 bits of the signed sum.

If the signed sum cannot be represented in 32 bits, then RT is updated with a value which is “saturated” to the nearest representable value. That is, if the signed sum is less than -2^{31}, then RT is updated with -2^{31}. Likewise, if the signed sum is greater than $2^{31} - 1$, then RT is updated with $2^{31} - 1$.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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.

If the unsigned sum can be represented in 32 bits, then RT is updated with the low-order 32 bits of the unsigned sum.

If the unsigned sum cannot be represented in 32 bits, then RT is updated with a value which is "saturated" to the maximum representable value of $2^{32} - 1$.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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 RT is updated with the low-order 32 bits of the unsigned sum.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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

$\text{prod}_{0:31} \leftarrow (\text{RA})_{16:31} \times (\text{RB})_{16:31}$ signed

$\text{temp}_{0:32} \leftarrow \text{prod}_{0:31} + (\text{RT})$

$(\text{RT}) \leftarrow \text{temp}_{1:32}$

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 RT is updated with the low-order 32 bits of the signed sum.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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 $(RT) \leftarrow (RT_0 \,\|\, {}^{31}(\neg RT_0))$
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.

If the signed sum can be represented in 32 bits, then RT is updated with the low-order 32 bits of the signed sum.

If the signed sum cannot be represented in 32 bits, then RT is updated with a value which is “saturated” to the nearest representable value. That is, if the signed sum is less than -2^{31}, then RT is updated with -2^{31}. Likewise, if the signed sum is greater than $2^{31} - 1$, then RT is updated with $2^{31} - 1$.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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

$\text{prod}_{0:31} \leftarrow (\text{RA})_{16:31} \times (\text{RB})_{16:31}$ unsigned

$\text{temp}_{0:32} \leftarrow \text{prod}_{0:31} + (\text{RT})$

$(\text{RT}) \leftarrow (\text{temp}_{1:32} \vee {}^{32}\text{temp}_0)$

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.

If the unsigned sum can be represented in 32 bits, then RT is updated with the low-order 32 bits of the unsigned sum.

If the unsigned sum cannot be represented in 32 bits, then RT is updated with a value which is “saturated” to the maximum representable value of $2^{32} – 1$.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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

$\text{prod}_{0:31} \leftarrow (\text{RA})_{16:31} \times (\text{RB})_{16:31}$ unsigned

$\text{temp}_{0:32} \leftarrow \text{prod}_{0:31} + (\text{RT})$

$(\text{RT}) \leftarrow \text{temp}_{1:32}$

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 RT is updated with the low-order 32 bits of the unsigned sum.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

mbar

31	MO		854	
0	6	11	21	31

The **mbar** instruction ensures that all loads and stores preceding **mbar** complete with respect to main storage before any loads and stores following **mbar** access main storage. As implemented in the PPC465, the MO field of **mbar** is ignored and treated as 0, providing a storage ordering function for all storage access instructions executed by the processor. Other processors implementing the **mbar** instruction may support one or more non-zero MO settings, specifying different subsets of storage accesses to be ordered by the **mbar** instruction in those implementations.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

Architecturally, **mbar** merely orders storage accesses, and does not perform execution nor context synchronization (see *Synchronization* on page 79). Therefore, non-storage access instructions after **mbar** could complete before the storage access instructions which were executed prior to **mbar** have actually completed their storage accesses. The **msync** instruction, on the other hand, *is* execution synchronizing, and *does* guarantee that all storage accesses initiated by instructions executed prior to the **msync** have completed before any instructions after the **msync** begin execution. However, the PPC465 implements the **mbar** instruction identically to the **msync** instruction, and thus both are execution synchronizing.

Software should nevertheless use the correct instruction (**mbar** or **msync**) as called for by the specific ordering and synchronizing requirements of the application, in order to guarantee portability to other implementations.

See *Storage Ordering and Synchronization* on page 80 for additional information on the use of the **msync** and **mbar** instructions.

Table 13-20. Extended Mnemonics for mbar

Mnemonic	Operands	Function	Other Registers Altered
mbar	None	Memory Barrier. *Extended mnemonic for* **mbar 0**	

Architecture Note

mbar replaces the PowerPC **eieio** instruction. **mbar** uses the same opcode as **eieio**; PowerPC applications which used **eieio** will get the function of **mbar** when executed on a PowerPC Book-E implementation. **mbar** is architecturally “stronger” than **eieio**, in that **eieio** forced separate ordering amongst different *categories* of storage accesses, while **mbar** forces such ordering amongst *all* storage accesses as a single category.

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

mcrxr BF

31	BF		512	
0	6	9	21	31

```
$n \leftarrow BF$
$(CR[CRn]) \leftarrow XER_{0:3}$
$XER_{0:3} \leftarrow {}^{4}0$
```

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.

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

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

mfdcr

RT, DCRN

31	RT	DCRF	323	
0	6	11	21	31

$\text{DCRN} \leftarrow \text{DCRF}_{5:9} \parallel \text{DCRF}_{0:4}$
$(\text{RT}) \leftarrow (\text{DCR}(\text{DCRN}))$

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

Programming Notes

Execution of this instruction is privileged.

The DCR number (DCRN) specified in the assembler language coding of the **mfdcr** instruction refers to a DCR number. The assembler handles the unusual register number encoding to generate the DCRF field.

mfdcrx RT, RA

31	RT	RA		259	
0	6	11	16	21	31

DCRN ← RA
GPR(RT) ← DCREG(DCRN)

The contents of the register RA denote a DCR.

The contents of the designated DCR are placed into GPR[RT].

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Programming Notes

Execution of this instruction is privileged and restricted to supervisor mode only.

The DCR number (DCRN) specified in the register RA refers to a DCR number (see User's Manual for a list of the Device Control Registers supported by the implementation).

Architecture Note

The specific numbers and definitions of any DCRs are outside the scope of the PowerPC Book-E architecture and the PPC465 core. Any DCRs are defined as part of the chip-level product incorporating the PPC465 core.

Note: Mfdcrx is supported by PPC440H/PPC465/PPC465 and not supported by PPC440A4 or PPC440G5.

mfdcrux RT, RA

31	RT	RA	▨	291	▨
0	6	11	16	21	31

```
DCRN ← (RA)
GPR(RT) ← DCREG(DCRN)
```

The contents of the register RA denote a DCR.

The contents of the designated DCR are placed into GPR[RT].

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Programming Notes

Execution of this instruction is not privileged and is intended for user-mode accessible DCRs only. Whether a particular DCR is accessible (in either user or supervisor mode) via **mfdcrux** depends on the implementation and the system.

The DCR number (DCRN) specified in the register RA refers to a DCR number (see User's Manual for a list of the Device Control Registers supported by the implementation).

Note: Mfdcrux is supported by PPC440H6/PPC464 and not supported by PPC440A4 or PPC440G5.

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.

mfspr RT, SPRN

31	**RT**	**SPRF**	**339**	▨
0	6	11	21	31

```
SPRN ← $SPRF_{5:9}$ || $SPRF_{0:4}$
(RT) ← (SPR(SPRN))
```

The contents of the SPR specified by the SPRF field are placed into register RT. See *Special Purpose Registers Sorted by SPR Number* on page 599 for a listing of SPR mnemonics and corresponding SPRN 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 *Privileged SPRs* on page 78 for a list of privileged SPRs.

The SPR number (SPRN) specified in the assembler language coding of the **mfspr** instruction refers to an SPR number. The assembler handles the unusual register number encoding to generate the SPRF field.

Table 13-21. Extended Mnemonics for mfspr

Mnemonic	Operands	Function
mfccr0 **mfccr1** **mfcsrr0** **mfcsrr1** **mfctr** **mfdac1** **mfdac2** **mfdbcr0** **mfdbcr1** **mfdbcr2** **mfdbdr** **mfdbsr** **mfdcdbtrh** **mfdcdbtrl** **mfdear** **mfdec** **mfdnv0** **mfdnv1** **mfdnv2** **mfdnv3** **mfdtv0** **mfdtv1** **mfdtv2** **mfdtv3** **mfdvc1** **mfdvc2** **mfdvlim** **mfesr** **mfiac1** **mfiac2** **mfiac3** **mfiac4** **mficdbdr** **mficdbtrh** **mficdbtrl** **mfinv0** **mfinv1** **mfinv2** **mfinv3** **mfitv0** **mfitv1** **mfitv2** **mfitv3** **mfivlim** **mfivor0** **mfivor1** **mfivor2** **mfivor3** **mfivor4** **mfivor5** **mfivor6** **mfivor7** **mfivor8** **mfivor9** **mfivor10** **mfivor11** **mfivor12** **mfivor13** **mfivor14** **mfivor15** **mfivpr** **mflr** **mfmcsr** **mfmcsrr0** **mfmcsrr1** **mfmmucr**	RT	Move from special purpose register SPRN. *Extended mnemonic for* **mfspr RT,SPRN** See *Special Purpose Registers Sorted by SPR Number* on page 599 for a list of valid SPRN values.

Table 13-21. Extended Mnemonics for mfspr (continued)

Mnemonic	Operands	Function
mfpid **mfpir** **mfpvr** **mfsprg0** **mfsprg1** **mfsprg2** **mfsprg3** **mfsprg4** **mfsprg5** **mfsprg6** **mfsprg7** **mfsrr0** **mfsrr1** **mftbl** **mftbu** **mftcr** **mftsr** **mfusprg0** **mfxer**		

msync

31		598	
0	6	21	31

The **msync** instruction guarantees that all instructions initiated by the processor preceding **msync** will complete before **msync** completes, and that no subsequent instructions will be initiated by the processor until after **msync** completes. **msync** also will not complete until all storage accesses associated with instructions preceding **msync** have completed.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None.

Invalid Instruction Forms

- Reserved fields

Programming Notes

The **msync** instruction is execution synchronizing (see *Execution Synchronization* on page 80), and guarantees that all storage accesses initiated by instructions executed prior to the **msync** have completed before any instructions after the **msync** begin execution. On the other hand, architecturally the **mbar** instruction merely *orders* storage accesses, and does *not* perform execution synchronization. Therefore, non-storage access instructions after **mbar** *could* complete before the storage access instructions which were executed prior to **mbar** have actually completed their storage accesses. However, the PPC465 implements the **mbar** instruction identically to the **msync** instruction, and thus both are execution synchronizing.

Software should nevertheless use the correct instruction (**mbar** or **msync**) as called for by the specific ordering and synchronizing requirements of the application, in order to guarantee portability to other implementations.

See *Storage Ordering and Synchronization* on page 80 for additional information on the use of the **msync** and **mbar** instructions.

Architecture Note

mbar replaces the PowerPC **eieio** instruction. **mbar** uses the same opcode as **eieio**; PowerPC applications which used **eieio** will get the function of **mbar** when executed on a PowerPC Book-E implementation. **mbar** is architecturally “stronger” than **eieio**, in that **eieio** forced separate ordering amongst different *categories* of storage accesses, while **mbar** forces such ordering amongst *all* storage accesses as a single category.

msync replaces the PowerPC **sync** instruction. **msync** uses the same opcode as **sync**; PowerPC applications which used **sync** get the function of **msync** when executed on a PowerPC Book-E implementation. **msync** is architecturally identical to the version of **sync** specified by an earlier version of the PowerPC architecture.

mtcrf FXM, RS

31	**RS**	▨	**FXM**	▨	**144**	▨
0	6	11	12	20	21	31

mask $\leftarrow$ ${}^4(FXM_0) \parallel {}^4(FXM_1) \parallel \ldots \parallel {}^4(FXM_6) \parallel {}^4(FXM_7)$
(CR) $\leftarrow$ $((RS) \land mask) \lor ((CR) \land \neg mask)$

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:

Table 13-22. FXM Bit Field Correspondence

FXM Bit Number	CR Bits Affected
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

Table 13-23. Extended Mnemonics for mtcrf

Mnemonic	Operands	Function
mtcr	RS	Move to CR. *Extended mnemonic for* **mtcrf 0xFF,RS**

mtdcr DCRN, RS

31	RS	DCRF	451	
0	6	11	21	31

DCRN ← $\text{DCRF}_{5:9}$ || $\text{DCRF}_{0:4}$
(DCR(DCRN)) ← (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

Programming Note

Execution of this instruction is privileged.

The DCR number (DCRN) specified in the assembler language coding of the **mtdcr** instruction refers to a DCR number. The assembler handles the unusual register number encoding to generate the DCRF field.

mtdcrx RA, RS

31	RS	RA		387	
0	6	11	16	21	31

DCRN = (RA)
DCREG(DCRN) = GPR[RS]

The contents of the register RA denote a DCR.

The contents of the designated GPR[RS] are placed into the designated DCR.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

Execution of this instruction is not privileged and restricted to supervisor mode only.

The DCR number (DCRN) specified in the register RA refers to a DCR number (see User’s Manual for a list of the Device Control Registers supported by the implementation).

Note: Mtdcrx is supported by PPC440H6/PPC465- and not supported by PPC440A4 or PPC440G5.

mtdcrux RA, RS

31	RS	RA		419	
0	6	11	16	21	31

DCRN ← (RA)
DCREG(DCRN) ← GPR[RS]

The contents of the register RA denote a DCR.

The contents of the designated GPR[RS] are placed into the designated DCR.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

Execution of this instruction is not privileged and is intended for user-mode accessible DCRs only. Whether a particular DCR is accessible (in either user or supervisor mode) via **mfdcrux** depends on the implementation and the system.

The DCR number (DCRN) specified in the register RA refers to a DCR number (see User’s Manual for a list of the Device Control Registers supported by the implementation).

Note: Mtdcrux is supported by PPC440H6/PPC465 and not supported by PPC440A4 or PPC440G5.

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 (see *Execution Synchronization* on page 80).

mtspr SPRN, RS

31	RS	SPRF	467	
0	6	11	21	31

SPRN $\leftarrow$ $SPRF_{5:9}$ || $SPRF_{0:4}$
(SPR(SPRN)) $\leftarrow$ (RS)

The contents of register RS are placed into register RT. See *Special Purpose Registers Sorted by SPR Number* on page 599 for a listing of SPR mnemonics and corresponding SPRN 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 contains 1. See *Privileged SPRs* on page 78 for a list of privileged SPRs.

The SPR number (SPRN) specified in the assembler language coding of the **mtspr** instruction refers to an SPR number. The assembler handles the unusual register number encoding to generate the SPRF field.

Table 13-24. Extended Mnemonics for mtspr

Mnemonic	Operands	Function
mtccr0 **mtccr1** **mtcsrr0** **mtcsrr1** **mtctr** **mtdac1** **mtdac2** **mtdbcr0** **mtdbcr1** **mtdbcr2** **mtdbdr** **mtdbsr** **mtdear** **mtdec** **mtdecar** **mtdnv0** **mtdnv1** **mtdnv2** **mtdnv3** **mtdtv0** **mtdtv1** **mtdtv2** **mtdtv3** **mtdvc1** **mtdvc2** **mtdvlim** **mtesr** **mtiac1** **mtiac2** **mtiac3** **mtiac4** **mtinv0** **mtinv1** **mtinv2** **mtinv3** **mtitv0** **mtitv1** **mtitv2** **mtitv3** **mtivlim** **mtivor0** **mtivor1** **mtivor2** **mtivor3** **mtivor4** **mtivor5** **mtivor6** **mtivor7** **mtivor8** **mtivor9** **mtivor10** **mtivor11** **mtivor12** **mtivor13** **mtivor14** **mtivor15** **mtivpr** **mtlr** **mtmcsr** **mtmcsrr0** **mtmcsrr1** **mtmmucr** **mtpid**	RT	Move to special purpose register SPRN. *Extended mnemonic for* **mtspr RT,SPRN** See *Special Purpose Registers Sorted by SPR Number* on page 599 for a list of valid SPRN values.

Table 13-24. Extended Mnemonics for mtspr (continued)

Mnemonic	Operands	Function
mtsprg0 **mtsprg1** **mtsprg2** **mtsprg3** **mtsprg4** **mtsprg5** **mtsprg6** **mtsprg7** **mtsrr0** **mtsrr1** **mttbl** **mttbu** **mttcr** **mttsr** **mtusprg0** **mtxer**		

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, considering both source operands as signed integers. The 32-bit result is placed into register RT.

Registers Altered

- RT
- CR[CR0] if Rc contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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, considering both source operands as unsigned integers. The 32-bit result is placed into register RT.

Registers Altered

- RT
- CR[CR0] if Rc contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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, considering both source operands as signed integers. The 32-bit result is placed into register RT.

Registers Altered

- RT
- CR[CR0] if Rc contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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, considering both source operands as unsigned integers. The 32-bit result is placed into register RT.

Registers Altered

- RT
- CR[CR0] if Rc contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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] if Rc contains 1

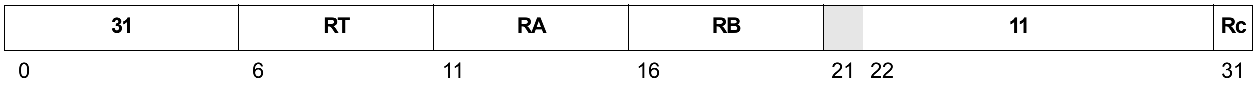
Programming Note

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.

Invalid Instruction Forms

- Reserved fields

mulhwu	RT, RA, RB	Rc=0
mulhwu.	RT, RA, RB	Rc=1



$prod_{0:63} \leftarrow$ (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] 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.

Invalid Instruction Forms

- Reserved fields

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, considering both source operands as signed integers. The 32-bit result is placed into register RT.

Registers Altered

- RT
- CR[CR0] if Rc contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

mullhwu	RT, RA, RB	Rc=0
mullhwu.	RT, RA, RB	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, considering both source operands as unsigned integers. The 32-bit result is placed into register RT.

Registers Altered

- RT
- CR[CR0] if Rc contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

mulli RT, RA, IM

7	RT	RA	IM	
0	6	11	16	31

$\text{prod}_{0:47} \leftarrow (\text{RA}) \times \text{IM}$
$(\text{RT}) \leftarrow \text{prod}_{16:47}$

The 48-bit product of register RA and the 16-bit IM field is formed. 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.

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] 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, however, is calculated specifically for a 64-bit signed product, and is dependent upon interpretation of the source operands as signed numbers.

nand	RA, RS, RB	Rc=0
nand.	RA, RS, RB	Rc=1

31	RT	RA	RB	476	Rc
0	6	11	16	21	31

$(RA) \leftarrow \neg((RS) \wedge (RB))$

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] if Rc contains 1

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) ← ¬(RA) + 1

The twos complement of the contents of register RA are placed into register RT.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE=1

Invalid Instruction Forms

- Reserved fields

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 signed product is subtracted from the contents of RT and RT is updated with the low-order 32 bits of the result.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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

```
$\text{nprod}_{0:31} \leftarrow -((\text{RA})_{16:31} \times (\text{RB})_{0:15}$ signed
$\text{temp}_{0:32} \leftarrow \text{nprod}_{0:31} + (\text{RT})$
if $((\text{nprod}_0 = \text{RT}_0) \wedge (\text{RT}_0 \neq \text{temp}_1))$ then $(\text{RT}) \leftarrow (\text{RT}_0 \,\|\, {}^{31}(\neg\text{RT}_0))$
else $(\text{RT}) \leftarrow \text{temp}_{1:32}$
```

The low-order halfword of RA is multiplied by the high-order halfword of RB. The signed product is subtracted from the contents of RT.

If the result of the subtraction can be represented in 32 bits, then RT is updated with the low-order 32 bits of the result.

If the result of the subtraction cannot be represented in 32 bits, then RT is updated with a value which is “saturated” to the nearest representable value. That is, if the result is less than -2^{31}, then RT is updated with -2^{31}. Likewise, if the result is greater than $2^{31} - 1$, then RT is updated with $2^{31} - 1$.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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

$\text{nprod}_{0:31} \leftarrow -((\text{RA})_{0:15} \times (\text{RB})_{0:15})$ signed

$\text{temp}_{0:32} \leftarrow \text{nprod}_{0:31} + (\text{RT})$

$(\text{RT}) \leftarrow \text{temp}_{1:32}$

The high-order halfword of RA is multiplied by the high-order halfword of RB. The signed product is subtracted from the contents of RT and RT is updated with the low-order 32 bits of the result.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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 $(RT) \leftarrow (RT_0 \,\|\, {}^{31}(\neg RT_0))$
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 subtracted from the contents of RT.

If the result of the subtraction can be represented in 32 bits, then RT is updated with the low-order 32 bits of the result.

If the result of the subtraction cannot be represented in 32 bits, then RT is updated with a value which is “saturated” to the nearest representable value. That is, if the result is less than -2^{31}, then RT is updated with -2^{31}. Likewise, if the result is greater than $2^{31} - 1$, then RT is updated with $2^{31} - 1$.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

nmaclhw	RT, RA, RB	OE=0, Rc=0
nmaclhw.	RT, RA, RB	OE=0, Rc=1
nmaclhwo	RT, RA, RB	OE=1, Rc=0
nmaclhwo.	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 signed product is subtracted from the contents of RT and RT is updated with the low-order 32 bits of the result.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

nmaclhws	RT, RA, RB	OE=0, Rc=0
nmaclhws.	RT, RA, RB	OE=0, Rc=1
nmaclhwso	RT, RA, RB	OE=1, Rc=0
nmaclhwso.	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 $(RT) \leftarrow (RT_0 \parallel {}^{31}(\neg RT_0))$
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 subtracted from the contents of RT.

If the result of the subtraction can be represented in 32 bits, then RT is updated with the low-order 32 bits of the result.

If the result of the subtraction cannot be represented in 32 bits, then RT is updated with a value which is “saturated” to the nearest representable value. That is, if the result is less than -2^{31}, then RT is updated with -2^{31}. Likewise, if the result is greater than $2^{31} - 1$, then RT is updated with $2^{31} - 1$.

Registers Altered

- RT
- CR[CR0] if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is implementation-specific and programs which use this instruction may not be portable to other PowerPC Book-E implementations. See *Instruction Set Portability* on page 344.

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] if Rc contains 1

Table 13-25. Extended Mnemonics for nor, nor.

Mnemonic	Operands	Function	Other Registers Altered
not	RA, RS	Complement register. $(RA) \leftarrow \neg(RS)$ *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] if Rc contains 1

Table 13-26. Extended Mnemonics for or, or.

Mnemonic	Operands	Function	Other Registers Altered
mr	RT, RS	Move register. $(RT) \leftarrow (RS)$ *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) ← (RS) ∨ ¬(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] if Rc contains 1

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

Table 13-27. Extended Mnemonics for ori

Mnemonic	Operands	Function	Other Registers Altered
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

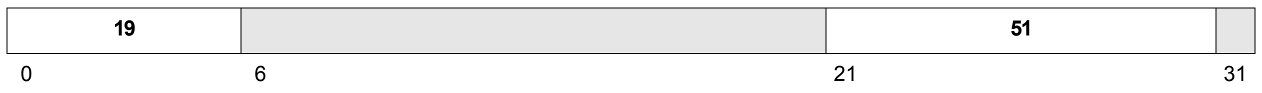
$(RA) \leftarrow (RS) \vee (IM \parallel {}^{16}0)$

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

rfci



(PC) ← (CSRR0)
(MSR) ← (CSRR1)

This instruction is used to return from a critical interrupt.

The program counter (PC) is restored with the contents of CSRR0 and the MSR is restored with the contents of CSRR1.

Instruction execution returns to the address contained in the PC.

Registers Altered

- MSR

Programming Note

Execution of this instruction is privileged and context-synchronizing (see *Context Synchronization* on page 79).

rfi

19		50	
0	6	21	31

(PC) ← (SRR0)
(MSR) ← (SRR1)

This instruction is used to return from a non-critical interrupt.

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 (see *Context Synchronization* on page 79).

rfmci

19		**38**	
0	6	21	31

(PC) ← (MCSRR0)
(MSR) ← (MCSRR1)

This instruction is used to return from a machine check interrupt.

The program counter (PC) is restored with the contents of MCSRR0 and the MSR is restored with the contents of MCSRR1.

Instruction execution returns to the address contained in the PC.

Registers Altered

- MSR

Programming Note

Execution of this instruction is privileged and context-synchronizing (see “Context Synchronization” on page 79).

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] if Rc contains 1

Table 13-28. 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 ← ROTL((RS), SH)
m ← MASK(MB, ME)
(RA) ← r ∧ 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] if Rc contains 1

Table 13-29. 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. ($n \le b < 32$) $(RA)_{b-n:31-n} \leftarrow (RS)_{b:31}$ $(RA)_{32-n:31} \leftarrow {}^{n}0$ $(RA)_{0:b-n-1} \leftarrow {}^{b-n}0$ *Extended mnemonic for* **rlwinm RA,RS,n,b−n,31−n**	
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 13-29. 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. $(RA) \leftarrow$ 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. $(RA) \leftarrow$ 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

```
$r \leftarrow \mathrm{ROTL}((RS), (RB)_{27:31})$
$m \leftarrow \mathrm{MASK}(MB, ME)$
$(RA) \leftarrow r \wedge m$
```

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] if Rc contains 1

Table 13-30. Extended Mnemonics for rlwnm, rlwnm.

Mnemonic	Operands	Function	Other Registers Altered
rotlw	RA, RS, RB	Rotate left. $(RA) \leftarrow \mathrm{ROTL}((RS), (RB)_{27:31})$ *Extended mnemonic for* **rlwnm RA,RS,RB,0,31**	
rotlw.		*Extended mnemonic for* **rlwnm. RA,RS,RB,0,31**	CR[CR0]

sc

17		1	
0	6	30	31

```
SRR1 ← MSR
SRR0 ← 4 + address of sc instruction
PC ← $IVPR_{0:15}$ || $IVOR8_{16:27}$ || $^{4}0$
MSR[WE, EE, PR, FP, FE0, FE1, DWE, DS, IS] ← $^{9}0$
```

A System Call exception is generated, and a System Call interrupt occurs (see *System Call Interrupt* on page 273 for more information on System Call interrupts). 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 interrupt vector address. The interrupt vector address is formed by concatenating the high halfword of the Interrupt Vector Prefix Register (IVPR), bits 16:27 of the Interrupt Vector Offset Register 8 (IVOR8), and 0b0000.

The MSR[WE, EE, PR, FP, FE0, FE1, DWE, DS, IS] bits are set to 0.

Program execution continues at the new address in the PC.

Registers Altered

- SRR0
- SRR1
- MSR[WE, EE, PR, FP, FE0, FE1, DWE, DS, IS]

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is context-synchronizing (see *Context Synchronization* on page 79).

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 ← $(RB)_{26:31}$
r ← ROTL((RS), n)
if n < 32 then
    m ← MASK(0, 31 – n)
else
    m ← $^{32}0$
(RA) ← r $\wedge$ m
```

The contents of register RS are shifted left by the number of bits specified by the contents of register $RB_{26:31}$. 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.

Note that if RB_{26} = 1, then the shift amount is 32 bits or more, and thus all bits are shifted out such that register RA is set to zero.

Registers Altered

- RA
- CR[CR0] if Rc contains 1

sraw	RA, RS, RB	Rc=0
sraw.	RA, RS, RB	Rc=1

31	RS	RA	RB	792	Rc
0	6	11	16	21	31

```
$n \leftarrow (RB)_{26:31}$
$r \leftarrow ROTL((RS), 32 - n)$
if $n < 32$ 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 $RB_{26:31}$. Bits shifted out of the least significant bit are lost. Bit 0 of Register RS 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.

Note that if RB_{26} = 1, then the shift amount is 32 bits or more, and thus all bits are shifted out such that register RA and XER[CA] are set to bit 0 of register RS.

Registers Altered

- RA
- XER[CA]
- CR[CR0] if Rc contains 1

srawi	RA, RS, SH	Rc=0
srawi.	RA, RS, SH	Rc=1

31	RS	RA	SH	824	Rc
0	6	11	16	21	31

```
$n \leftarrow SH$
$r \leftarrow ROTL((RS), 32 - n)$
$m \leftarrow MASK(n, 31)$
$s \leftarrow (RS)_0$
$(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$
$XER[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$
```

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] if Rc contains 1

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 ← $(RB)_{26:31}$
r ← ROTL((RS), 32 – n)
if n < 32 then
    m ← MASK(n, 31)
else
    m ← $^{32}0$
(RA) ← r $\wedge$ m
```

The contents of register RS are shifted right by the number of bits specified the contents of register $RB_{26:31}$. 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.

Note that if RB_{26} = 1, then the shift amount is 32 bits or more, and thus all bits are shifted out such that register RA is set to zero.

Registers Altered

- RA
- CR[CR0] if Rc contains 1

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

stbu RS, D(RA)

39	RS	RA	D	
0	6	11	16	31

```
$EA \leftarrow (RA|0) + EXTS(D)$
$MS(EA, 1) \leftarrow (RS)_{24:31}$
$(RA) \leftarrow EA$
```

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.

The EA is placed into register RA.

Registers Altered

- RA

Invalid Instruction Forms

RA = 0

stbux RS, RA, RB

31	RS	RA	RB	247	
0	6	11	16	21	31

```
$EA \leftarrow (RA|0) + (RB)$
$MS(EA, 1) \leftarrow (RS)_{24:31}$
$(RA) \leftarrow EA$
```

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.

The least significant byte of register RS is stored into the byte at the EA.

The EA is placed into register RA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA

Invalid Instruction Forms

- Reserved fields

RA = 0

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

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

sthbrx RS, RA, RB

31	RS	RA	RB	918	
0	6	11	16	21	31

$EA \leftarrow (RA|0) + (RB)$
$MS(EA, 2) \leftarrow BYTE\_REVERSE((RS)_{16:31})$

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 byte-reversed from the default byte ordering for the memory page referenced by the EA. The resulting halfword is stored at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

Byte ordering is generally controlled by the Endian (E) storage attribute (see *Memory Management* on page 219). The store byte reverse instructions provide a mechanism for data to be stored to a memory page using the opposite byte ordering from that specified by the Endian storage attribute.

sthu RS, D(RA)

45	RS	RA	D	
0	6	11	16	31

```
$EA \leftarrow (RA|0) + EXTS(D)$
$MS(EA, 2) \leftarrow (RS)_{16:31}$
$(RA) \leftarrow EA$
```

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.

The EA is placed into register RA.

Registers Altered

- RA

Invalid Instruction Forms

RA = 0

sthux RS, RA, RB

31	RS	RA	RB	439	▨
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
MS(EA, 2) ← $(RS)_{16:31}$
(RA) ← 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.

The least significant halfword of register RS is stored into the halfword at the EA.

The EA is placed into register RA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA

Invalid Instruction Forms

- Reserved fields
- RA = 0

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

stmw RS, D(RA)

47	RS	RA	D	
0	6	11	16	31

```
EA ← (RA|0) + EXTS(D)
r ← RS
do while r ≤ 31
   MS(EA, 4) ← (GPR(r))
   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 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

Programming Note

This instruction can be restarted, meaning that it could be interrupted after having already stored some of the register values to memory, and then re-executed from the beginning (after returning from the interrupt), in which case the registers which had already been stored prior to the interrupt will be stored a second time.

Store String Word Immediate

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) and wrapping to GPR(0) as necessary, 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

This instruction can be restarted, meaning that it could be interrupted after having already stored some of the register values to memory, and then re-executed from the beginning (after returning from the interrupt), in which case the registers which had already been stored prior to the interrupt will be stored a second time.

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) and wrapping to GPR(0) as necessary, 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

This instruction can be restarted, meaning that it could be interrupted after having already stored some of the register values to memory, and then re-executed from the beginning (after returning from the interrupt), in which case the registers which had already been stored prior to the interrupt will be stored a second time.

If XER[TBC] = 0, no GPRs are stored to memory, and **stswx** is treated as a no-op. Furthermore, if the EA is such that a Data Storage, Data TLB Error, or Data Address Compare Debug exception occurs, **stswx** is treated as a no-op and no interrupt occurs as a result of the exception.

stw RS, D(RA)

36	RS	RA	D	
0	6	11	16	31

```
EA ← (RA|0) + EXTS(D)
MS(EA, 4) ← (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

stwbrx RS, RA, RB

31	RS	RA	RB	662	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
MS(EA, 4) ← BYTE_REVERSE($(RS)_{0:31}$)
```

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 word in register RS is byte-reversed from the default byte ordering for the memory page referenced by the EA. The resulting word is stored at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

Byte ordering is generally controlled by the Endian (E) storage attribute (see *Memory Management* on page 219). The store byte reverse instructions provide a mechanism for data to be stored to a memory page using the opposite byte ordering from that specified by the Endian storage attribute.

stwcx. RS, RA, RB

31	RS	RA	RB	150	1
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
if RESERVE = 1 then
   MS(EA, 4) ← (RS)
   RESERVE ← 0
   (CR[CR0]) ← $^{2}0$ || 1 || XER[SO]
else
   (CR[CR0]) ← $^{2}0$ || 0 || 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]$_{0:1}$ are cleared
- CR[CR0]$_2$ is set to indicate whether or not the store was performed (1 indicates that it was)
- CR[CR0]$_3$ is set to the contents of the XER[SO] bit

Registers Altered

- CR[CR0]

Programming Notes

The **lwarx** and **stwcx.** instructions are typically 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 multiple processes. Only **lwarx** can set the reservation bit to 1. **stwcx.** sets the reservation bit to 0 upon its completion, whether or not **stwcx.** actually stored (RS) to memory. CR[CR0]$_2$ 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 the register as required
      stwcx.       # attempt to store the semaphore; reset reservation
      bne loop     # some other process intervened and cleared the reservation prior to the above
                   # stwcx.; try again
```

PPC465 supports **lwarx** and **stwcx.** in a memory coherent manner for pages marked coherent (M=1) and write-through (WL1=1) pages, as specified in the following table, and generates an exception whenever a lwarx and stwcx. to W=1 or IL1=1 pages (Table 13-31 on page 518).

Table 13-31. Page Attributes and lwarx & stwcx. Operations

W	WL1	IL1D	IL2D	I	**Operation**
0	0	0	0	0	Coherency not supported
0	1	0	0	0	Fully supported
-	-	1	0	0	When L1D and L2 caches are in Write-through, or L1D or L2 caching is inhibited, lwarx and stwcx. are not supported even in the memory coherence required storage space, and therefore, a Storage synchronization exception will be generated. Refer to Shared Storage and Storage Attributes and Data Storage Interrupt sections of Book E.
-	-	0	1	0	
-	-	1	1	1	
1	-	-			

The PowerPC Book-E architecture specifies that the EA for the **lwarx** instruction must be word-aligned (that is, a multiple of 4 bytes); otherwise, the result is undefined. Although the PPC465 executes **stwcx.** when CCR1[L2COBE] = 0 and CCR1[L2COBE] = 0 regardless of the EA alignment, if the operation of the pairing of **lwarx** and **stwcx.** is to produce the desired result, software must ensure that the EAs for both instructions are word-aligned. This requirement is due to the manner in which misaligned storage accesses may be broken up into separate, aligned accesses by the PPC465.

The PowerPC Book-E architecture also specifies that it is implementation-dependent as to whether a Data Storage, Data TLB Error, Alignment, or Debug interrupt occurs when the reservation bit is off at the time of execution of an **stwcx.** instruction, and when the conditions are such that a non-**stwcx.** store-type storage access instruction would have resulted in such an interrupt. The PPC465 implements **stwcx.** such that Data Storage and Debug (DAC and/or DVC exception type) interrupts do not occur when the reservation bit is off at the time of execution of the **stwcx.** Instead, the **stwcx.** instruction completes without causing the interrupt and without storing to memory, and CR[CR0] is updated to indicate the failure of the **stwcx.**

On the other hand, the PPC465 causes a Data TLB Error interrupt if a Data TLB Miss exception occurs during execution of a **stwcx.** instruction, regardless of the state of the reservation. Similarly, the PPC465 causes an Alignment interrupt if the EA of the **stwcx.** operand is not word-aligned when CCR0[FLSTA] is 1, or when CCR1[L2COBE] is 1, regardless of the state of the reservation “Core Configuration Register 0 (CCR0)” on page 73 for more information on the Force Load/Store Alignment function).

In addition, a storage access control exception, DSI, is also generated differently from PPC440's. A storage access control exception occurs when a storage access instruction attempts to access a location in storage that is not access permitted while CCR1[L2COBE] is 1, regardless of the state of the reservation. However, if CCR1[L2COBE] is 0, this DSI only occurs if both a storage access control violation and the reservation exist.

In addition, the L2 cache must be configured to be non-zero in size, snooping must be enabled and FAM must not be configured to the full size of the array or **stwcx.** instructions will fail but no alignment interrupt will occur.

lwarx and **stwcx.** are expected to be word aligned when L2CCR1[L2COBE] bit is set indicating L2C being present. All misaligned **lwarx** and **stwcx.** operations result in an Alignment exception as permitted by PowerPC Book-E.

Caution: When the entire cache array is set to FAM (FAMES=10 for 256KB), the atomic access instructions,**lwarx** and **stwcx.** will not function correctly and instead will always fail.

For the **stwcx.** instruction, if the address is not aligned on a word boundary, then the value in CR[CR0] is undefined, as is whether or not the reservation (if one existed) has been cleared if CCR1[L2COBE] is set to 0. However, if CCR1[L2COBE] bit is set to 1, **stwcx.** instruction will generate an alignment exception in this case. CR[CR0] will not be changed.

See *Table 7-8, “Required Configuration Settings for Coherent Operation,” on page -174.*

stwu RS, D(RA)

37	RS	RA	D
0	6	11	16 … 31

```
EA ← (RA|0) + EXTS(D)
MS(EA, 4) ← (RS)
(RA) ← EA
```

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 into the word at the EA.

The EA is placed into register RA.

Registers Altered

- RA

Invalid Instruction Forms

RA = 0

stwux RS, RA, RB

31	RS	RA	RB	183	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
MS(EA, 4) ← (RS)
(RA) ← 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.

The contents of register RS are stored into the word at the EA.

The EA is placed into register RA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA

Invalid Instruction Forms

- Reserved fields
- RA = 0

stwx RS, RA, RB

31	RS	RA	RB	151	
0	6	11	16	21	31

EA ← (RA|0) + (RB)
MS(EA,4) ← (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

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] if Rc contains 1
- XER[SO, OV] if OE contains 1

Table 13-32. 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) $\leftarrow$ $\neg$(RA) + (RB) + 1
if $\neg$(RA) + (RB) + 1 $\overset{u}{>}$ $2^{32} - 1$ then
   XER[CA] $\leftarrow$ 1
else
   XER[CA] $\leftarrow$ 0
```

The sum of the ones complement of register RA, register RB, and 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] if Rc contains 1
- XER[SO, OV] if OE contains 1

Table 13-33. Extended Mnemonics for subfc, subfc., subfco, subfco.

Mnemonic	Operands	Function	Other Registers Altered
subc	RT, RA, RB	Subtract (RB) from (RA). (RT) $\leftarrow$ $\neg$(RB) + (RA) + 1. Place carry-out in XER[CA]. *Extended mnemonic for* **subfc RT,RB,RA**	
subc.	RT, RA, RB	*Extended mnemonic for* **subfc. RT,RB,RA**	CR[CR0]
subco	RT, RA, RB	*Extended mnemonic for* **subfco RT,RB,RA**	XER[SO, OV]
subco.	RT, RA, RB	*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] if Rc contains 1
- XER[SO, OV] if OE contains 1

subfic RT, RA, IM

8	RT	RA	IM
0	6	11	16 … 31

```
(RT) $\leftarrow \neg$(RA) + EXTS(IM) + 1
if $\neg$(RA) + EXTS(IM) + 1 $\overset{u}{>} 2^{32} - 1$ then
    XER[CA] $\leftarrow$ 1
else
    XER[CA] $\leftarrow$ 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]

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) ← ¬(RA) – 1 + XER[CA]
if ¬(RA) + 0xFFFF FFFF + XER[CA] $\overset{u}{>}$ $2^{32}$ – 1 then
    XER[CA] ← 1
else
    XER[CA] ← 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] if Rc contains 1
- XER[SO, OV] if OE contains 1
- XER[CA]

Invalid Instruction Forms

- Reserved fields

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] if Rc contains 1
- XER[SO, OV] if OE contains 1

Invalid Instruction Forms

- Reserved fields

tlbre RT, RA, WS

31	RT	RA	WS	946	
0	6	11	16	21	31

```
tlbentry ← TLB[$(RA)_{26:31}$]
if WS =0
    $(RT)_{0:27}$ ← tlbentry[EPN,V,TS,SIZE]
    if CCR0[CRPE] = 0
        $(RT)_{28:31}$ ← $^{4}0$
    else
        $(RT)_{28:31}$ ← TPAR
    MMUCR[STID] ← tlbentry[TID]
else if WS = 1
    $(RT)_{0:21}$ ← tlbentry[RPN]
    if CCR0[CRPE] = 0
        $(RT)_{22:23}$ ← $^{2}0$
    else
        $(RT)_{22:23}$ ← PAR1
    $(RT)_{24:27}$ ← $^{4}0$
    $(RT)_{28:31}$ ← tlbentry[ERPN]
else if WS = 2
    if CCR0[CRPE] = 0
        $(RT)_{0:1}$ ← $^{2}0$
    else
        $(RT)_{0:1}$ ← PAR2
    $(RT)_{2:15}$ ← $^{14}0$
    $(RT)_{11:24}$ ← tlbentry[WL1,IL1I,IL1D,IL2I,IL2D,U0,U1,U2,U3,W,I,M,G,E]
    $(RT)_{25}$ ← 0
    $(RT)_{26:31}$ ← tlbentry[UX,UW,UR,SX,SW,SR]
else (RT), MMUCR[STID] ← undefined
```

The contents of the specified portion of the selected TLB entry are placed into register RT (and also MMUCR[STID] if WS = 0).

The parity bits in the TLB entry (TPAR, PAR1, and PAR2) are placed into the register RT if and only if the Cache Read Parity Enable bit, CCR0[CRPE], is set to 1.

The contents of RA are used as an index into the TLB. If this value is greater than the index of the highest numbered TLB entry (63), the results are undefined.

The WS field specifies which portion of the TLB entry is placed into RT. If WS = 0, the TID field of the selected TLB entry is read into MMUCR[STID]. See *Memory Management* on page 219 for descriptions of the TLB entry fields.

If the value of the WS field is greater than 2, 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
- MMUCR[STID] (if WS = 0)

Invalid Instruction Forms

- Reserved fields
- Invalid WS value

Programming Notes

Execution of this instruction is privileged.

The PPC465 does not automatically synchronize the context of the MMUCR[STID] field between a **tlbre** instruction which updates the field, and a **tlbsx**[**.**] instruction which uses it as a source operand. Therefore, software must execute an **isync** instruction between the execution of a **tlbre** instruction and a subsequent **tlbsx**[.] instruction to ensure that the **tlbsx**[**.**] instruction will use the new value of MMUCR[STID]. On the other hand, the PPC465 *does* automatically synchronize the context of MMUCR[STID] between **tlbre** and **tlbwe**, as well as between **tlbre** and **mfspr** which specifies the MMUCR as the source SPR, so no **isync** is required in these cases.

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
    $CR[CR0]_0$ ← 0
    $CR[CR0]_1$ ← 0
    $CR[CR0]_3$ ← XER[SO}
if  Valid TLB entry matching EA and MMUCR[STID,STS] is in the TLB then
    (RT) ← Index of matching TLB Entry
    if Rc = 1
        $CR[CR0]_2$ ← 1
else
    (RT) ← Undefined
    if Rc = 1
        $CR[CR0]_2$ ← 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 MMUCR[STID,STS]. See *Memory Management* on page 219 for descriptions of the TLB fields and how they participate in the determination of a match. If a matching entry is found, its index (0 - 63) is placed into bits 26:31 of RT, and bits 0:25 are set to 0. If no match is found, the contents of RT are undefined.

The record bit (Rc) specifies whether the results of the search will affect CR[CR0] as shown above, such that $CR[CR0]_2$ can be tested if there is a possibility that the search may fail.

Registers Altered

- CR[CR0] if Rc contains 1

Invalid Instruction Forms

- None

Programming Notes

Execution of this instruction is privileged.

The PPC465 does not automatically synchronize the context of the MMUCR[STID] field between a **tlbre** instruction which updates the field, and a **tlbsx[.]** instruction which uses it as a source operand. Therefore, software must execute an **isync** instruction between the execution of a **tlbre** instruction and a subsequent **tlbsx[.]** instruction to ensure that the **tlbsx[.]** instruction will use the new value of MMUCR[STID]. On the other hand, the PPC465 *does* automatically synchronize the context of MMUCR[STID] between **tlbre** and **tlbwe**, as well as between **tlbre** and **mfspr** which specifies the MMUCR as the source SPR, so no **isync** is required in these cases.

tlbsync

31		566	
0	6	21	31

The **tlbsync** instruction is provided by the PowerPC Book-E architecture to support synchronization of TLB operations between processors in a coherent multi-processor system. Since the PPC465 does not support coherent multi-processing, this instruction performs no operation, and is provided only to facilitate code portability.

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 privileged. Translation is not required to be active during the execution of this instruction.

Since the PPC465 does not support tightly-coupled multiprocessor systems, **tlbsync** performs no operation.

tlbwe RS, RA, WS

31	RS	RA	WS	978	
0	6	11	16	21	31

```
tlbentry ← TLB[$(RA)_{26:31}$]
if WS = 0
    tlbentry[EPN,V,TS,SIZE] ← $(RS)_{0:27}$
    tlbentry[TID] ← MMUCR[STID]
else if WS = 1
    tlbentry[RPN] ← $(RS)_{0:21}$
    tlbentry[ERPN] ← $(RS)_{28:31}$
else if WS = 2
    tlbentry[WL1,IL1I,IL1D,IL2I,IL2D,U0,U1,U2,U3,W,I,M,G,E] ← $(RS)_{11:24}$
    tlbentry[UX,UW,UR,SX,SW,SR] ← $(RS)_{26:31}$
else tlbentry ← undefined
```

The contents of the specified portion of the selected TLB entry are replaced with the contents of register RS (and also MMUCR[STID] if WS = 0).

Parity check bits are automatically calculated and stored in the TLB entry as the tlbwe is executed. The contents of the RS register in the TPAR, PAR1, and PAR2 fields (for WS=0,1,or 2, respectively) is ignored by tlbwe; the parity is calculated from the other data bits being written to the TLB entry.

The contents of RA are used as an index into the TLB. If this value is greater than the index of the highest numbered TLB entry (63), the results are undefined.

The WS field specifies which portion of the TLB entry is replaced by the contents of RS. If WS = 0, the TID field of the selected TLB entry is replaced by the value in MMUCR[STID]. See *Memory Management* on page 219 for descriptions of the TLB entry fields.

If the value of the WS field is greater than 2, 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 Note

Execution of this instruction is privileged.

tw TO, RA, RB

31	TO	RA	RB	4	
0	6	11	16	21	31

```
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)   )
    SRR0 $\leftarrow$ address of **tw** instruction
    SRR1 $\leftarrow$ MSR
    ESR[PTR] $\leftarrow$ 1 (other bits cleared)
    MSR[WE, EE, PR, FP, FE0, FE1, DWE, DS, IS]) $\leftarrow$ ${}^{9}0$
    PC $\leftarrow$ $IVPR_{0:15}$ || $IVOR6_{16:27}$ || ${}^{4}0$
else no operation
```

Register RA is compared with register RB. If any comparison condition enabled by the TO field is true, a Trap exception type Program interrupt occurs as follows (see *Program Interrupt* on page 270 for more information on Program interrupts). The contents of the MSR are copied into SRR1 and the address of the **tw** instruction) is placed into SRR0. ESR[PTR] is set to 1 and the other bits ESR bits cleared to indicate the type of exception causing the Program interrupt.

The program counter (PC) is then loaded with the interrupt vector address. The interrupt vector address is formed by concatenating the high halfword of the Interrupt Vector Prefix Register (IVPR), bits 16:27 of the Interrupt Vector Offset Register 6 (IVOR6), and 0b0000.

MSR[WE, EE, PR, FP, FE0, FE1, DWE, DS, IS] are set to 0.

Program execution continues at the new address in the PC.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- SRR0 (if trap condition is met)
- SRR1 (if trap condition is met)
- MSR[WE, EE, PR, FP, FE0, FE1, DWE, DS, IS] (if trap condition is met)
- ESR (if trap condition is met)

Invalid Instruction Forms

- Reserved fields

Programming Notes

This instruction can be inserted into the execution stream by a debugger to implement breakpoints, and is not typically used by application code.

The enabling of trap debug events may affect the interrupt type caused by the execution of **tw** instruction. Specifically, trap instructions may be enabled to cause Debug interrupts instead of Program interrupts. See *Trap (TRAP) Debug Event* on page 331 for more details.

Table 13-34. 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**	
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

```
if (   (($(RA)$ $<$ EXTS(IM) $\wedge$ $TO_0$ = 1)   $\vee$
       (($(RA)$ $>$ EXTS(IM) $\wedge$ $TO_1$ = 1)   $\vee$
       (($(RA)$ $=$ EXTS(IM) $\wedge$ $TO_2$ = 1)   $\vee$
       (($(RA)$ $\overset{u}{<}$ EXTS(IM) $\wedge$ $TO_3$ = 1)   $\vee$
       (($(RA)$ $\overset{u}{>}$ EXTS(IM) $\wedge$ $TO_4$ = 1)   )
    SRR0 $\leftarrow$ address of twi instruction
    SRR1 $\leftarrow$ MSR
    ESR[PTR] $\leftarrow$ 1 (other bits cleared)
    MSR[WE, EE, PR, FP, FE0, FE1, DWE, DS, IS]) $\leftarrow$ $^{9}0$
    PC $\leftarrow$ $IVPR_{0:15}$ || $IVOR6_{16:27}$ || $^{4}0$
else no operation
```

Register RA is compared with the sign-extended IM field. If any comparison condition selected by the TO field is true, a Trap exception type Program interrupt occurs as follows (see *Program Interrupt* on page 270 for more information on Program interrupts). The contents of the MSR are copied into SRR1 and the address of the **twi** instruction) is placed into SRR0. ESR[PTR] is set to 1 and the other bits ESR bits cleared to indicate the type of exception causing the Program interrupt.

The program counter (PC) is then loaded with the interrupt vector address. The interrupt vector address is formed by concatenating the high halfword of the Interrupt Vector Prefix Register (IVPR), bits 16:27 of the Interrupt Vector Offset Register 6 (IVOR6), and 0b0000.

MSR[WE, EE, PR, FP, FE0, FE1, DWE, DS, IS] are set to 0.

Program execution continues at the new address in the PC.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- SRR0 (if trap condition is met)
- SRR1 (if trap condition is met)
- MSR[WE, EE, PR, FP, FE0, FE1, DWE, DS, IS] (if trap condition is met)
- ESR (if trap condition is met)

Invalid Instruction Forms

- Reserved fields

Programming Notes

This instruction can be inserted into the execution stream by a debugger to implement breakpoints, and is not typically used by application code.

The enabling of trap debug events may affect the interrupt type caused by the execution of **tw** instruction. Specifically, trap instructions may be enabled to cause Debug interrupts instead of Program interrupts. See *Trap (TRAP) Debug Event* on page 331 for more details.

Table 13-35. 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**	
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

$\text{MSR[EE]} \leftarrow (\text{RS})_{16}$

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 Notes

Execution of this instruction is privileged.

This instruction is typically used as part of a code sequence which can provide the equivalent of an atomic read-modify-write of the MSR, as follows:

```
mfmsr Rn     #save EE in Rn[16]
wrteei 0     #Turn off EE (leaving other bits unchanged)
•            #Code with EE disabled
•
•
wrtee Rn     #restore EE without affecting any MSR changes that occurred in the disabled code
```

wrteei

E

31		E		163	
0	6	16	17	21	31

MSR[EE] ← 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 Notes

Execution of this instruction is privileged.

This instruction is typically used as part of a code sequence which can provide the equivalent of an atomic read-modify-write of the MSR, as follows:

```
mfmsr Rn    #save EE in Rn[16]
wrteei 0    #Turn off EE (leaving other bits unchanged)
•           #Code with EE disabled
•
•
wrtee Rn    #restore EE without affecting any MSR changes that occurred in the disabled code
```

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

- RA
- CR[CR0] if Rc contains 1

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

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

14. Floating Point Instruction Set

Descriptions of the PPC465 floating point instructions follow. Each description contains the following elements:

- Instruction names (mnemonic and full)
- Instruction syntax
- Instruction format diagram
- Pseudocode description
- Prose description
- Registers altered

Where appropriate, instruction descriptions list exceptions and invalid instruction forms, and provide programming notes.

Table 14-1 summarizes the PPC465 instruction set by category.

Table 14-1. Instruction Categories

Category	Subcategory	Instruction Types
Floating Point	Computational	Elementary arithmetic, multiply-add, rounding and converting, square root and reciprocal estimate
	Noncomputational	Load/store, move, compare, Floating-Point Status and Control Register

14.1 Instruction Set Portability

To support embedded real-time applications, the PPC465 implements the defined floating point instruction set of the Book-E Enhanced PowerPC Architecture, with the exception the **fctid**, **fsqrt**, and **fsqrts** instructions.

The Book-E Enhanced PowerPC Architecture defines some instructions that have record forms, often called “dot forms,’ that update the CR1 field of the Condition Register (CR). The record forms of these instructions are not implemented.

14.2 Instruction Formats

For more detailed information about instruction formats, including a summary of instruction field usage and instruction format diagrams for the PPC465, see *Instruction Summary* on page 609.

Instructions are four bytes long. Instruction addresses are always word-aligned.

Instruction bits 0 through 5 always contain the primary opcode. Many instructions also have an extended opcode field. The remaining instruction bits are contained in additional fields. All instruction fields belong to one of the following categories:

- Defined

  These instruction fields 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 specifiers and immediate values, each of which may contain any one of a number of values. The instruction format diagrams specify the field names of 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 specified value, the instruction is illegal and an Illegal Instruction exception type Program interrupt 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 PPC465 executes all invalid instruction forms without causing an Illegal Instruction exception.

14.3 Pseudocode

The pseudocode that appears in the instruction descriptions provides a semi-formal language for describing instruction operations.

The pseudocode uses the following notation:

+	Twos complement addition
%	Remainder of an integer division; (33 % 32) = 1.
$\overset{u}{<}, \overset{u}{>}$	Unsigned comparison relations
(FPR(r))	The contents of FPR r, where $0 \le r \le 31$.
(FRx)	The contents of an FPR, where *x* is A, B, C, S, or T
(GPR(r))	The contents of GPR r, where $0 \le r \le 31$.
(RA|0)	The contents of the register RA or 0, if the RA field is 0.
(Rx)	The contents of a GPR, where *x* is A, B, S, or T
0bn	A binary number
0xn	A hexadecimal number
<, >	Signed comparison relations
=	Assignment
=, ≠	Equal, not equal relations
CEIL(x)	Least integer ≥ x.
CIA	Current instruction address; the 32-bit address of the instruction being described by a sequence of pseudocode. This address is used to set the next instruction address (NIA). Does not correspond to any architected register.
DCR(DCRN)	A Device Control Register (DCR) specified by the DCRF field in an **mfdcr** or **mtdcr** instruction
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.
EXTS(x)	The result of extending *x* on the left with sign bits.
FLD	An instruction or register field
FLD_b	A bit in a named instruction or register field
$FLD_{b,b,\ldots}$	A list of bits, by number or name, in a named instruction or register field
$FLD_{b:b}$	A range of bits in a named instruction or register field

FRx	An FPR, where *x* is A, B, C, S, or T
GPR(r)	General Purpose Register (GPR) r, where $0 \le r \le 31$.
GPRs	RA, RB, . . .
MASK(MB,ME)	Mask having 1s in positions MB through ME (wrapping if MB > ME) and 0s elsewhere.
MS(addr, n)	The number of bytes represented by *n* at the location in main storage represented by *addr*.
NIA	Next instruction address; the 32-bit address of the next instruction to be executed. In pseudocode, a successful branch is indicated by assigning a value to NIA. For instructions that do not branch, the NIA is CIA +4.
PC	Program counter.
Rx	A GPR, where *x* is A, B, S, or T
REG[FLD, FLD . . .]	A list of fields in a named register
REG[FLD:FLD]	A range of fields in a named register
REG[FLD]	A field in a named register
REG_b	A bit in a named register
$REG_{b,b,\ldots}$	A list of bits, by number or name, in a named register
$REG_{b:b}$	A range of bits in a named register
RESERVE	Reserve bit; indicates whether a process has reserved a block of storage.
ROTL((RS),n)	Rotate left; the contents of RS are shifted left the number of bits specified by *n*.
SPR(SPRN)	A Special Purpose Register (SPR) specified by the SPRF field in an **mfspr** or **mtspr** instruction
$c_{0:3}$	A four-bit object used to store condition results in compare instructions.
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.
if...then...else...	Conditional execution; if *condition* then *a* else *b*, where *a* and *b* represent one or more pseudocode statements. Indenting indicates the ranges of *a* and *b*. If *b* is null, the else does not appear.
instruction(EA)	An instruction operating on a data or instruction cache block associated with an EA.
leave	Leave innermost do loop or do loop specified in a leave statement.
n	A decimal number
^{n}b	The bit or bit value *b* is replicated *n* times.
xx	Bit positions which are don’t-cares.
$\|\|$	Concatenation
$\times$	Multiplication
$\div$	Division yielding a quotient
$\oplus$	Exclusive-OR (XOR) logical operator
–	Twos complement subtraction, unary minus
$\neg$	NOT logical operator
$\wedge$	AND logical operator
$\vee$	OR logical operator

14.3.1 Operator Precedence

Table 14-2 lists the pseudocode operators and their associativity in descending order of precedence:

Table 14-2. 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
$=, \neq, <, >, \overset{u}{<}, \overset{u}{>}$	Left to right
$\wedge, \oplus$	Left to right
$\vee$	Left to right
$\leftarrow$	None

14.4 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). Some instructions also change other registers, but the details of the changes are not included in the instruction descriptions. Common examples of these kinds of register changes include the Floating-Point Registers (FPRs) and the Floating-Point Status and Control Register (FPSCR). For discussion of the FPRs, see *Floating-Point Registers (FPR0:31)* on page 86. For discussion of the FPSCR, see *Floating-Point Status and Control Register (FPSCR)* on page 87.

14.5 Floating-Point Instructions

Primary opcode 63 is used for the double-precision arithmetic instructions and miscellaneous instructions (for example, the *Floating-Point Status and Control Register Manipulation* instructions). Primary opcode 59 is used for the single-precision arithmetic instructions.

The single-precision instructions for which there is a corresponding double-precision instruction have the same format and extended opcode as that double-precision instruction.

14.6 Alphabetical Instruction Listing

The following pages list the defined floating point instructions implemented in the PPC465.

fabs FRT, FRB Rc = 0

63	FRT		FRB	263	Rc
0	6	11	16	21	31

(FRT) ← 0 || $(FRB)_{1:63}$

The contents of FRB, with bit 0 set to zero, are placed into FRT.

Registers Altered

- FRT

Exceptions

An attempt to execute **fabs** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fadd FRT, FRA, FRB Rc = 0

63	FRT	FRA	FRB		21	Rc
0	6	11	16	21	26	31

(FRT) ← (FRA) + (FRB)

The floating-point operand in FRA is added to the floating-point operand in FRB.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to double precision under control of the FPSCR[RN] and placed into FRT.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand and the three guard bits (G, R, X) enter into the computation.

If a carry occurs, the significand of the sum is shifted right one bit position and the exponent is increased by one.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI]

Exceptions

An attempt to execute **fadd** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fadds FRT, FRA, FRB Rc = 0

59	FRT	FRA	FRB		21	Rc
0	6	11	16	21	26	31

$(\text{FRT}) \leftarrow (\text{FRA}) +_{sp} (\text{FRB})$

The floating-point operand in FRA is added to the floating-point operand in FRB.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to single precision under control of the FPSCR[RN] and placed into FRT.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand and the three guard bits (G, R, X) enter into the computation.

If a carry occurs, the significand of the sum is shifted right one bit position and the exponent is increased by one.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI]

Exceptions

An attempt to execute **fadds** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fcmpo BF, FRA, FRB

63	BF		FRA	FRB	32	
0	6	9	11	16	21	31

```
if (FRA) is a NaN or
(FRB) is a NaN then        c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else                       c← 0b0010
FPSCR[FPCC] ← c
$CR_{4 \times BF:4 \times BF + 3} \leftarrow c$
if (FRA) is a SNaN or (FRB) is a SNaN then do
    if FPSCR[VE] = 0 then FPSCR[VXVC] ← 1
else if (FRA) is a QNaN or (FRB) is a QNaN then FPSCR[VXVC] ← 1
```

The floating-point operand in FRA is compared to the floating-point operand in FRB. The result of the compare is placed into the CR field BF and FPSCR[FPCC]. The comparison ignores the sign of zero (that is, +0 is considered equal to –0).

If (FRA) or (FRB) is a NaN, either quiet or signaling, the CR field BF and FPSCR[FPCC] are set to reflect unordered.

If (FRA) or (FRB) is a Signaling NaN and Invalid Operation is disabled (FPSCR[VE] = 0), FPSCR[VXVC] = 1. If neither (FRA) nor (FRB) is a Signaling NaN, but at least one operand is a Quiet NaN, FPSCR[VXVC] = 1.

Registers Altered

- CR[BF]
- FPSCR[FPCC, FX, VXSNAN, VXVC]

fcmpu BF, FRA, FRB

63	BF		FRA	FRB	0	
0	6	9	11	16	21	31

```
if (FRA) is a NaN or
(FRB) is a NaN then       c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else                      c← 0b0010
FPSCR[FPCC] ← c
$CR_{4 \times BF:4 \times BF+3}$ ← c
if ((FRA) is a SNaN or (FRB) is a SNaN) then FPSCR[VXSNAN] ← 1
```

The floating-point operand in FRA is compared to the floating-point operand in FRB. The result of the compare is placed into the CR field BF and FPSCR[FPCC]. The comparison ignores the sign of zero (that is, +0 is considered equal to –0).

If (FRA) or (FRB) is a NaN, either quiet or signaling, the CR field BF and FPSCR[FPCC] are set to reflect unordered.

If either (FRA) or (FRB) is a Signaling NaN, FPSCR[VXSNAN] = 1.

Registers Altered

- CR
- FPSCR[FPCC, FX, VXSNAN]

fctiw FRT, FRB Rc = 0

63	FRT		FRB	12	Rc
0	6	11	16	21	31

The floating-point operand in FRB is converted to a 32-bit signed integer, using the rounding mode specified by FPSCR[RN], and placed into $FRT_{32:63}$. $FRT_{0:31}$ are undefined.

If (FRB) is greater than $2^{31}–1$, $FRT_{32:63}$ are set to 0x7FFFFFFF. If (FRB) is less than $–2^{31}$, $FRT_{32:63}$ are set to 0x80000000.

Except for enabled Invalid Operation Exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

Registers Altered

- FRT
- FPSCR[FR, FI, FX, XX, VXSNAN, VXCVI]
- FPSCR[FPRF] undefined

Exceptions

An attempt to execute **fctiw** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fctiwz FRT, FRB Rc = 0

63	FRT		FRB	15	Rc
0	6	11	16	21	31

The floating-point operand in FRB is converted to a 32-bit signed integer, using the *Round toward Zero* rounding mode, and placed into $FRT_{32:63}$. $FRT_{0:31}$ are undefined.

If (FRB) is greater than $2^{31}–1$, $FRT_{32:63}$ are set to 0x7FFF FFFF. If (FRB) is less than $–2^{31}$, $FRT_{32:63}$ are set to 0x8000 0000.

Except for enabled Invalid Operation Exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

Registers Altered

- FRT
- FPSCR[FR, FI, FX, XX, VXSNAN, VXCVI]
- FPSCR[FPRF] undefined

Exceptions

An attempt to execute **fctiwz** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fdiv FRT, FRA, FRB Rc = 0

63	FRT	FRA	FRB		18	Rc
0	6	11	16	21	26	31

(FRT) ← (FRA) ÷ (FRB)

The floating-point operand in FRA is divided by the floating-point operand in FRB.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to double precision under control of FPSCR[RN] and placed into FRT.

The remainder is not supplied as a result.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1 and Zero Divide Exceptions when FPSCR[ZE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNAN, VXIDI, VXZDZ]

Exceptions

An attempt to execute **fdiv** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fdivs FRT, FRA, FRB Rc = 0

59	FRT	FRA	FRB		18	Rc
0	6	11	16	21	26	31

FRT ← (FRA) ÷ (FRB)

The floating-point operand in FRA is divided by the floating-point operand in FRB.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to single precision under control of FPSCR[RN] and placed into (FRT).

The remainder is not supplied as a result.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1 and Zero Divide Exceptions when FPSCR[ZE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNAN, VXIDI, VXZDZ]

Exceptions

An attempt to execute **fdivs** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fmadd FRT, FRA, FRC, FRB Rc = 0

63	**FRT**	**FRA**	**FRB**	**FRC**	**29**	**Rc**
0	6	11	16	21	26	31

$(FRT) \leftarrow [(FRA) \times (FRC)] + (FRB)$

The floating-point operand in FRA is multiplied by the floating-point operand in FRC. The floating-point operand in FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to double precision under control of the FPSCR[RN] and placed into FRT.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ]

Exceptions

An attempt to execute **fmadd** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fmadds FRT, FRA, FRC, FRB Rc = 0

59	FRT	FRA	FRB	FRC	29	Rc
0	6	11	16	21	26	31

$(FRT) \leftarrow [(FRA) \times (FRC)] + (FRB)$

The floating-point operand in FRA is multiplied by the floating-point operand in FRC. The floating-point operand in FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to single precision under control of the FPSCR[RN] and placed into FRT.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ]

Exceptions

An attempt to execute **fmadds** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fmr FRT, FRB Rc = 0

63	FRT		FRB	72	Rc
0	6	11	16	21	31

(FRT) ← FR(FRB)

The contents of FRB are placed into FRT.

Registers Altered

- FRT

Exceptions

An attempt to execute **fmr** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fmsub FRT, FRA, FRC, FRB Rc = 0

63	FRT	FRA	FRB	FRC	28	Rc
0	6	11	16	21	26	31

(FRT) ← [(FRA) × (FRC)] – (FRB)

The floating-point operand in FRA is multiplied by the floating-point operand in FRC. The floating-point operand in FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to double precision under control of FPSCR[RN] and placed into FRT.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ]

Exceptions

An attempt to execute **fmsub** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fmsubs FRT, FRA, FRC, FRB Rc = 0

59	FRT	FRA	FRB	FRC	28	Rc
0	6	11	16	21	26	31

(FRT) ← [(FRA) × (FRC)] – (FRB)

The floating-point operand in FRA is multiplied by the floating-point operand in FRC. The floating-point operand in FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to single precision under control of FPSCR[RN] and placed into FRT.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ]

Exceptions

An attempt to execute **fmsubs** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fmul FRT, FRA, FRC Rc = 0

63	FRT	FRA		FRC	25	Rc
0	6	11	16	21	26	31

(FRT) ← (FRA) × (FRC)

The floating-point operand in FRA is multiplied by the floating-point operand in FRC.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to double precision under control of FPSCR[RN] and placed into FRT.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXIMZ]

Exceptions

An attempt to execute **fmul** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fmuls FRT, FRA, FRC Rc = 0

59	FRT	FRA		FRC	25	Rc
0	6	11	16	21	26	31

(FRT) ← (FRA) × (FRC)

The floating-point operand in FRA is multiplied by the floating-point operand in FRC.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to single precision under control of FPSCR[RN] and placed into FRT.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXIMZ]

Exceptions

An attempt to execute **fmuls** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fnabs FRT, FRB Rc = 0

63	**FRT**	▨	**FRB**	**136**	**Rc**
0	6	11	16	21	31

$(FRT) \leftarrow 1 \parallel (FRB)_{1:63}$

The contents of FRB, with bit 0 set to one, are placed into FRT.

Registers Altered

- FRT

Exceptions

An attempt to execute **fnabs** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fneg FRT, FRB Rc = 0

63	FRT	///	FRB	40	Rc
0	6	11	16	21	31

$(FRT) \leftarrow \neg(FRB)_0 \mid\mid (FRB)_{1:63}$

The contents of FRB, with bit 0 inverted, are placed into FRT.

Registers Altered

- FRT

Exceptions

An attempt to execute **fneg** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fnmadd FRT, FRA, FRC, FRB Rc = 0

63	FRT	FRA	FRB	FRC	31	Rc
0	6	11	16	21	26	31

FRT $\leftarrow -([(\text{FRA}) \times (\text{FRC})] + (\text{FRB}))$

The floating-point operand in FRA is multiplied by the floating-point operand in FRC. The floating-point operand in FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to double precision under control of the FPSCR[RN], then negated and placed into (FRT).

This instruction produces the same result as would be obtained by using the **fmadd** instruction and then negating the result, with the following exceptions.

- Quiet NaNs propagate with no effect on their “sign” (high-order) bit.
- Quiet NaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- Signaling NaNs that are converted to Quiet NaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the Signaling NaN.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ]

Exceptions

An attempt to execute **fnmadd** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fnmadds FRT, FRA, FRC, FRB Rc = 0

59	**FRT**	**FRA**	**FRB**	**FRC**	**31**	**Rc**
0	6	11	16	21	26	31

$(FRT) \leftarrow -([(FRA) \times (FRC)] + (FRB))$

The floating-point operand in FRA is multiplied by the floating-point operand in FRC. The floating-point operand in FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to single precision under control of the FPSCR[RN], then negated and placed into FRT.

This instruction produces the same result as would be obtained by using the **fmadds** instruction and then negating the result, with the following exceptions.

- Quiet NaNs propagate with no effect on their “sign” (high-order) bit.
- Quiet NaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- Signaling NaNs that are converted to Quiet NaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the Signaling NaN.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ]

Exceptions

An attempt to execute **fnmadds** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fnmsub FRT, FRA, FRC, FRB Rc = 0

63	FRT	FRA	FRB	FRC	30	Rc
0	6	11	16	21	26	31

(FRT) ← –([(FRA) × (FRC)] – (FRB))

The floating-point operand in FRA is multiplied by the floating-point operand in FRC. The floating-point operand in FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to double precision under control of the FPSCR[RN], then negated and placed into FRT.

This instruction produces the same result as would be obtained by using the **fmsub** instruction and then negating the result, with the following exceptions.

- Quiet NaNs propagate with no effect on their ”sign” (high-order) bit.
- Quiet NaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- Signaling NaNs that are converted to Quiet NaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the Signaling NaN.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ]

Exceptions

An attempt to execute **fnmsub** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fnmsubs FRT, FRA, FRC, FRB Rc = 0

59	FRT	FRA	FRB	FRC	30	Rc
0	6	11	16	21	26	31

(FRT) ← –([(FRA) × (FRC)] – (FRB))

The floating-point operand in FRA is multiplied by the floating-point operand in FRC. The floating-point operand in FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to single precision under control of the FPSCR[RN], then negated and placed into FRT.

This instruction produces the same result as would be obtained by using the **fmsubs** instruction and then negating the result, with the following exceptions.

- Quiet NaNs propagate with no effect on their ”sign” (high-order) bit.
- Quiet NaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- Signaling NaNs that are converted to Quiet NaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the Signaling NaN.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ]

Exceptions

An attempt to execute **fnmsubs** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fres FRT, FRB (Rc=0)

59	FRT	///	FRB	///	24	Rc
0	6	11	16	21	26	31

(FRT) ← FPReciprocalEstimate(FRB)

A single-precision estimate of the reciprocal of the floating-point operand in FRB is placed into FRT. The estimate is correct to a precision of one part in 2^{13} of the reciprocal of (FRB), that is,

$$\left| \frac{\text{estimate} - \frac{1}{x}}{\frac{1}{x}} \right| \le \frac{1}{2}13$$

where x is the initial value of (FRB).

Table 14-3 summarizes operation with operands having various special values.

Table 14-3. ***fres*** *Operation with Special Operand Values*

Operand	Result	Exception
–∞	–0	—
–0	–∞[1]	ZX
+0	+∞[1]	ZX
+∞	+0	—
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	—
Note: 1. No result if FPSCR[ZE] = 1 2. No result if FPSCR[VE] = 1.		

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1 and Zero Divide Exceptions when FPSCR[ZE] = 1.

Registers Altered

- FRT
- FPSCR[FX, OX, UX, ZX, VXSNAN]
- FPSCR[FPRF, FR, FI] undefined

Exceptions

An attempt to execute **fres** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Architecture Notes

The value placed into FRT may vary between implementations, and between different executions on the same implementation.

frsp FRT, FRB (Rc=0)

63	**FRT**	***///***	**FRB**	**12**		**Rc**
0	6	11	16	21	26	31

The floating-point operand in FRB is rounded to single-precision, using the rounding mode specified by FPSCR[RN], and placed into FRT.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN]

Exceptions

An attempt to execute **frsp** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

frsqrte FRT, FRB (Rc=0)

63	FRT	///	FRB	///	26	Rc
0	6	11	16	21	26	31

(FRT) ← FPReciprocalSquareRootEstimate(FRB)

A double-precision estimate of the reciprocal of the square root of the floating-point operand in FRB is placed into FRT. The estimate placed is correct to a precision of one part in 2^{13} of the reciprocal of the square root of (FRB), that is,

$$\left| \frac{\left(\text{estimate} - \frac{1}{\sqrt{x}}\right)}{\frac{1}{\sqrt{x}}} \right| \le \frac{1}{2}13$$

where *x* is the initial value in FPR(FRB). Note that the value placed into FPR(FRT) may vary between implementations, and between different executions on the same implementation.

Table 14-4 summarizes operation with various special values of the operand.

*Table 14-4. **frsqrte** Operation with Special Operand Values*

Operand	Result	Exception
–∞	QNaN[2]	VXSQRT
< 0	QNaN[2]	VXSQRT
–0	–∞[1]	ZX
+0	+∞[1]	ZX
+∞	+0	—
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	—
Note: 1. No result if FPSCR[ZE] = 1 2. No result if FPSCR[VE] = 1.		

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1 and Zero Divide Exceptions when FPSCR[ZE] = 1.

Registers Altered

- FRT
- FPSCR[FX, ZX, VXSNAN, VXSQRT]
- FPSCR[FPRF, FR, FI] undefined

Exceptions

An attempt to execute **frsqrte** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Architecture Notes

No single-precision version of this instruction is provided. If the floating-point operand in FRB is representable in single format, so is the resulting value of FRT.

Implementation Note

The precision of this estimate in this implementation is higher than that specified in Book-E (2^{-5}). Therefore, code relying on this higher precision may not work on other PowerPC implementations.

fsel FRT, FRA, FRC, FRB (Rc=0)

63	FRT	FRA	FRB	FRC	23	Rc
0	6	11	16	21	26	31

```
if (FRA) ≥ 0.0 then (FRT) ← (FRC)
else (FRT) ← (FRB)
```

The floating-point operand in FRA is compared to zero. If the operand is greater than or equal to 0, FRT is set to the contents of FRC. If the operand is less than 0 or is a NaN, FRT is set to the contents of FRB. The comparison ignores the sign of zero (that is, +0 is considered equal to –0).

Registers Altered

- FRT

Exceptions

An attempt to execute **fsel[.]** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Programming Notes

Care must be taken in using fsel if compatibility with IEEE 754 is required, or if the values being tested can be NaNs or infinities.

Architecture Notes

The *Select* instruction is similar to a *Move* instruction, and therefore does not alter FPSCR.

fsub FRT, FRA, FRB Rc = 0

63	FRT	FRA	FRB		20	Rc
0	6	11	16	21	26	31

(FRT) ← (FRA) – (FRB)

The floating-point operand in FPR(FRB) is subtracted from the floating-point operand in FPR(FRA).

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to double precision under control of FPSCR[RN] and placed into FRT.

The operation of **fsub** is identical to that of **fadd**, except that the contents of FRB participate in the operation with the sign bit (bit 0) inverted.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

- FRT
- FPSRP[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN]

Exceptions

An attempt to execute **fsub** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

fsubs FRT, FRA, FRB Rc = 0

59	FRT	FRA	FRB		20	Rc
0	6	11	16	21	26	31

(FRT) ← (FRA) – (FRB)

The floating-point operand in FPR(FRB) is subtracted from the floating-point operand in FPR(FRA).

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to single precision under control of FPSCR[RN] and placed into FRT.

The operation of **fsubs** is identical to that of **fadds**, except that the contents of FRB participate in the operation with the sign bit (bit 0) inverted.

FPSCR[FPRF] is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR[VE] = 1.

Registers Altered

- FRT
- FPSCR[FPRF, FR, FI FX, OX, UX, XX, VXSNAN]

Exceptions

An attempt to execute **fsubs** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

lfd FRT, D(RA)

50	FRT	RA	D
0	6	11	16 … 31

$EA \leftarrow (RA|0) + EXTS(D)$
$(FRT) \leftarrow MEM(EA,8)$

Let the effective address (EA) be the sum of (RA|0) and the displacement obtained by sign-extending the 16-bit D field to 32 bits.

The doubleword in storage addressed by EA is placed into FRT.

Registers Altered

- FRT

Exceptions

An attempt to execute **lfd** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

The effective address (EA) must be 4-byte aligned to prevent Alignment exception.

lfdu FRT, D(RA)

51	FRT	RA	D	
0	6	11	16	31

```
EA ← (RA|0) + EXTS(D)
(FRT) ← MEM(EA,8)
RA ← EA
```

Let the effective address (EA) be the sum of (RA|0) and the displacement obtained by sign-extending the 16-bit D field to 32 bits.

The doubleword in storage addressed by EA is placed into FRT.

EA is placed into RA.

Registers Altered

- FRT
- RA

Exceptions

An attempt to execute **lfdu** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

The effective address (EA) must be 4-byte aligned to prevent Alignment exception.

Invalid Forms

RA = 0

lfdux FRT, RA, RB

31	FRT	RA	RB	631	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
(FRT) ← MEM(EA,8)
RA ← EA
```

Let the effective address (EA) be the sum (RA|0) and the index specified by (RB).

The doubleword in storage addressed by EA is placed into FRT.

EA is placed into RA.

Registers Altered

- FRT
- RA

Exceptions

An attempt to execute **lfdux** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

The effective address (EA) must be 4-byte aligned to prevent Alignment exception.

Invalid Forms

RA = 0

lfdx FRT, RA, RB

31	FRT	RA	RB	599	
0	6	11	16	21	31

EA ← (RA|0) + (RB)
(FRT) ← MEM(EA,8)

Let the effective address (EA) be the sum (RA|0) and the index specified by (RB).

The doubleword in storage addressed by EA is placed into FRT.

Registers Altered

- FRT

Exceptions

An attempt to execute **lfdx** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Implementation Note

The EA must be a multiple of 4. Otherwise, an Alignment exception occurs.

The effective address (EA) must be 4-byte aligned to prevent Alignment exception.

lfs FRT, D(RA)

48	FRT	RA	D	
0	6	11	16	31

```
EA ← (RA|0) + EXTS(D)
(FRT) ← DOUBLE(MEM(EA,4))
```

Let the effective address (EA) be the sum of (RA|0) and the displacement obtained by sign-extending the 16-bit D field to 32 bits.

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see *Data Handling and Precision* on page 94) and placed into FRT.

Registers Altered

- FRT

Exceptions

An attempt to execute **lfs** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Implementation Note

The EA must be a multiple of 4. Otherwise, an Alignment exception occurs.

lfsu FRT, D(RA)

49	FRT	RA	D	
0	6	11	16	31

```
EA ← (RA|0) + EXTS(D)
(FRT) ← DOUBLE(MEM(EA,4))
RA ← EA
```

Let the effective address (EA) be the sum of (RA|0) and the displacement obtained by sign-extending the 16-bit D field to 32 bits.

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see *Data Handling and Precision* on page 94) and placed into FRT.

EA is placed into RA.

Registers Altered

- FRT
- RA

Exceptions

An attempt to execute **lfsu** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Invalid Forms

RA = 0

Implementation Note

The EA must be a multiple of 4. Otherwise, an Alignment exception occurs.

lfsux FRT, RA, RB

31	FRT	RA	RB	567	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
(FRT) ← DOUBLE(MEM(EA,4))
RA ← EA
```

Let the effective address (EA) be the sum (RA|0) and the index specified by (RB).

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see *Data Handling and Precision* on page 94) and placed into FRT.

EA is placed into RA.

Registers Altered

- FRT
- RA

Exceptions

An attempt to execute **lfsux** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Invalid Forms

RA = 0

Implementation Note

The EA must be a multiple of 4. Otherwise, an Alignment exception occurs.

lfsx FRT, RA, RB

31	FRT	RA	RB	535	
0	6	11	16	21	31

EA ← (RA|0) + (RB)
(FRT) ← DOUBLE(MEM(EA,4))

Let the effective address (EA) be the sum (RA|0) and the index specified by (RB).

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see *Data Handling and Precision* on page 94) and placed into FRT.

Registers Altered

- FRT

Exceptions

An attempt to execute **lfsx** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Implementation Note

The EA must be a multiple of 4. Otherwise, an Alignment exception occurs.

mcrfs BF, BFA

31	BF		BFA		64	
0	6	9	11	14	21	31

$CR_{BF \times 4:BF \times 4+3} \leftarrow FPSCR_{BFA \times 4:BFA \times 4 + 3}$

$FPSCR_{BFA \times 4:BFA \times 4 + 3} \leftarrow 0b0000$

The contents of the FPSCR specified by BFA are copied to the CR field specified by BF. All exception bits that are copied are set to 0 in the FPSCR. If FPSCR[FX] is copied, it is set to 0 in the FPSCR.

Registers Altered

- CR field BF
- FPSCR[FX, OX] if BFA=0
- FPSCR[UX, ZX, XX, VXSNAN] if BFA=1
- FPSCR[VXISI, VXIDI, VXZDZ, VXIMZ] if BFA=2
- FPSCR[VXVC] if BFA=3
- FPSCR[VXSOFT, VXSQRT, VXCVI] if BFA=5

Exceptions

An attempt to execute **mcrfs** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

mffs FRT Rc = 0

63	FRT			583	Rc
0	6	11	16	21	31

(FRT) ← FPSCR

The contents of the FPSCR are placed into $FRT_{32:63}$. $FRT_{0:31}$ are undefined.

Registers Altered

- FRT

Exceptions

An attempt to execute **mffs** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

mtfsb0 BT Rc = 0

63	BT		70	Rc
0	6	11 16	21	31

FPSCR$_{BT}$ $\leftarrow$ 0

Bit BT of the FPSCR is set to 0.

Registers Altered

- FPSCR[BT]

An attempt to execute **mtfsb0** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

mtfsb1 BT Rc = 0

63	**BT**	▨	▨	**70**	**Rc**
0	6	11	16	21	31

FPSCR$_{BT}$ ← 1

Bit BT of the FPSCR is set to 1.

Registers Altered

- FPSCR[BT, FX]

AIf Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

Programming Notes

FPSCR[FEX, VX] cannot be explicitly set.

mtfsf FLM, FRB Rc = 0

63		FLM		FRB	711	Rc
0	6	7	15	16	21	31

```
i ← 0
do while i<8
   if $FLM_i$=1 then $FPSCR_{4 \times i:4 \times i + 3} \leftarrow (FRB)_{4 \times i:4 \times i + 3}$
   i ← i+1
```

The contents of bits 32:63 of FRB are placed into the FPSCR under control of the field mask specified by FLM. The field mask identifies the affected 4-bit fields. Let *i* be an integer in the range 0–7. If $FLM_i = 1$, FPSCR field *i* (FPSCR bits $4 \times i - 4 \times i + 3$) is set to the contents of the corresponding field of the low-order 32 bits of FRB.

FPSCR[FX] is altered only if $FLM_0 = 1$.

Registers Altered

- FPSCR fields selected by mask

Exceptions

An attempt to execute **mtfsf** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

Programming Notes

Updating fewer than all eight fields of the FPSCR may have substantially poorer performance on some implementations than updating all the fields.

When $FPSCR_{0:3}$ is specified, bits 0 and 3 (FPSCR[FX, OX]) are set to the values of FRB_{32} and FRB_{35}. Even if **mtfsf** causes FPSCR[OX] to change from 0 to 1, FPSCR[FX] is set from FRB_{32}, not by the usual rule that FPSCR[FX] is set to 1 when an exception bit changes from 0 to 1. Bits 1 and 2 (FPSCR[FEX, VX]) are set according to the usual rule, given on XREF TBD, and not from $FRB_{33:34}$.

Implementation Note

In general, the source FPR should contain data produced by a double-precision load operation. The value moved to FPSCR is unpredictable if the source FPR contains denormalized single format data (that is, data that cannot be represented as a normalized single-precision number that was produced from a single precision operation.

This is similar to the achitectural rule for **stfiwx**.

mtfsfi BF, U Rc = 0

63	BF		U		134	Rc
0	6	9	16	20	21	31

$FPSCR_{BF \times 4:BF \times 4 + 3} \leftarrow U$

The value of the U field is placed into FPSCR field BF.

$FPSCR_{FX}$ is altered only if BF = 0.

Registers Altered

- $FPSCR_{BF}$

Exceptions

An attempt to execute **mtfsfi** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

If Rc = 1, an Unimplemented Operation exception type Program interrupt occurs.

Programming Notes

When $FPSCR_{0:3}$ is specified, bits 0 and 3 FPSCR[FX, OX] are set to the values of U_0 and U_3. Even if **mtfsfi** causes FPSCR[OX] to change from 0 to 1, FPSCR[FX] is set from U_0 and not by the usual rule that FPSCR[FX] is set to 1 when an exception bit changes from 0 to 1). Bits 1 and 2 (FPSCR[FEX, VX]) are set according to the usual rule, given on XREF TBD, and not from $U_{1:2}$.

stfd FRS, D(RA)

54	FRS	RA	D	
0	6	11	16	31

EA ← (RA|0) + EXTS(D)
MEM(EA,8) ← (FRS)

Let the effective address (EA) be the sum of (RA|0) and the displacement obtained by sign-extending the 16-bit D instruction field to 32 bits.

The contents of FRS are stored into the doubleword in storage addressed by EA.

Registers Altered

- None

Exceptions

An attempt to execute **stfd** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Implementation Note

The effective address (EA) must be 4-byte aligned to prevent Alignment exception.

stfdu FRS, D(RA)

55	FRS	RA	D	
0	6	11	16	31

```
EA ← (RA|0) + EXTS(D)
MEM(EA,8) ← (FRS)
RA ← EA
```

Let the effective address (EA) be the sum of (RA|0) and the displacement obtained by sign-extending the 16-bit D instruction field to 32 bits.

The contents of FRS are stored into the doubleword in storage addressed by EA.

EA is placed into RA.

Registers Altered

- RA

Exceptions

An attempt to execute **stfdu** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Invalid Forms

RA = 0

Implementation Note

The effective address (EA) must be 4-byte aligned to prevent Alignment exception.

stfdux FRS, RA, RB

31	FRS	RA	RB	759	
0	6	11	16	21	31

EA ← (RA|0) + (RB)
MEM(EA,8) ← (FRS)
RA ← EA

Let the effective address (EA) be the sum of (RA|0) and the index specified by (RB).

The contents of FRS are stored into the doubleword in storage addressed by EA.

EA is placed into RA.

Registers Altered

- RA

Exceptions

An attempt to execute **stfdux** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Invalid Forms

RA = 0

Implementation Note

The effective address (EA) must be 4-byte aligned to prevent Alignment exception.

stfdx FRS, RA, RB

31	**FRS**	**RA**	**RB**	**727**	▨
0	6	11	16	21	31

EA ← (RA|0) + (RB)
MEM(EA,8) ← (FRS)

Let the effective address (EA) be the sum of (RA|0) and the index specified by (RB).

The contents of FRS are stored into the doubleword in storage addressed by EA.

Registers Altered

- None

Exceptions

An attempt to execute **stfdx** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Implementation Note

The effective address (EA) must be 4-byte aligned to prevent Alignment exception.

stfiwx FRS, RA, RB

31	FRS	RA	RB	983	
0	6	11	16	21	31

EA ← (RA|0) + (RB)
MEM(EA,4) ← $(FRS)_{32:63}$

Let the effective address (EA) be the sum of (RA|0) and the index specified by (RB).

The contents of bits 32:63 of FRS are stored, without conversion, into the word in storage addressed by EA.

If the contents of FRS were produced, either directly or indirectly, by a *Load Floating-Point Single* instruction, a single-precision *Arithmetic* instruction, or **frsp**, the value stored is undefined. (The contents of FRS are produced directly by such an instruction if FRS is the target register for the instruction. The contents of FRS are produced indirectly by such an instruction if FRS is the final target register of a sequence of one or more *Floating-Point Move* instructions, with the input to the sequence having been produced directly by such an instruction.)

Registers Altered

- None

Exceptions

An attempt to execute **stfiwx** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Implementation Note

The EA must be a multiple of 4. Otherwise, an Alignment exception occurs.

stfs FRS, D(RA)

52	FRS	RA	D	
0	6	11	16	31

```
EA ← (RA|0) + EXTS(D)
MEM(EA,4) ← SINGLE(FRS)
```

Let the effective address (EA) be the sum of (RA|0) and the displacement obtained by sign-extending the 16-bit D instruction field to 32 bits.

The contents of FRS are converted to single-precision format (see *Data Handling and Precision* on page 94) and stored into the word in storage addressed by EA.

Registers Altered

- None

Exceptions

An attempt to execute **stfs** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Implementation Note

The EA must be a multiple of 4. Otherwise, an Alignment exception occurs.

stfsu FRS, D(RA)

53	FRS	RA	D	
0	6	11	16	31

```
EA ← (RA|0) + EXTS(D)
MEM(EA,4) ← SINGLE(FRS)
RA ← EA
```

Let the effective address (EA) be the sum of (RA|0) and the displacement obtained by sign-extending the 16-bit D instruction field to 32 bits.

The contents of FRS are converted to single-precision format (see *Data Handling and Precision* on page 94) and stored into the word in storage addressed by EA.

EA is placed into RA.

Registers Altered

- RA

Exceptions

An attempt to execute **stfsu** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Invalid Forms

RA = 0

Implementation Note

The EA must be a multiple of 4. Otherwise, an Alignment exception occurs.

stfsux FRS, RA, RB

31	FRS	RA	RB	695	
0	6	11	16	21	31

```
EA ← (RA|0) + (RB)
MEM(EA,4) ← SINGLE(FRS)
RA ← EA
```

Let the effective address (EA) be the sum of (RA|0) and the index specified by (RB).

The contents of FRS are converted to single-precision format (see *Data Handling and Precision* on page 94) and stored into the word in storage addressed by EA.

EA is placed into RA.

Registers Altered

- RA

Exceptions

An attempt to execute **stfsux** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Invalid Forms

RA = 0

Implementation Note

The EA must be a multiple of 4. Otherwise, an Alignment exception occurs.

stfsx FRS, RA, RB

31	FRS	RA	RB	663	
0	6	11	16	21	31

EA ← (RA|0) + (RB)
MEM(EA,4) ← SINGLE(FRS)

Let the effective address (EA) be the sum of (RA|0) and the index specified by (RB).

The contents of FRS are converted to single-precision format (see*Data Handling and Precision* on page 94) and stored into the word in storage addressed by EA.

Registers Altered

- None

Exceptions

An attempt to execute **stfsx** while MSR[FP] = 0 causes a Floating-Point Unavailable exception.

Implementation Note

The EA must be a multiple of 4. Otherwise, an Alignment exception occurs.

15. Register Summary

This chapter provides an alphabetical listing and bit definitions for the registers contained in the PPC465.

The registers, of five types, are grouped into several functional categories according to the processor functions with which they are associated. More information about the registers and register categories is provided in *Section 3.2 Registers* on page 48, and in the chapters describing the processor functions with which each register category is associated.

15.1 Register Categories

Table 3-3 on page 51 summarizes the register categories and the registers contained in each category. Italicized register names are implementation-specific. All other registers are defined by the Book-E Enhanced PowerPC Architecture.

Table 15-1, lists the Special Purpose Registers (SPRs) in order by SPR number (SPRN). The table provides mnemonics, names, SPR numbers, model (user or supervisor), and access. All SPR numbers not listed are reserved, and should be neither read nor written.

Note that three registers, DBSR, MCSR, and TSR, are indicated as having the access type of *Read/Clear*. These three registers are *status* registers, and as such behave differently than other SPRs when written. The term Read/Clear does not mean that these registers are automatically cleared upon being read. Clear refers to their behavior when being written. Instead of simply overwriting the SPR with the data in the source GPR, the status SPR is updated by zeroing those bit positions corresponding to 1 values in the source GPR, with those bit positions corresponding to 0 values in the source GPR being left unchanged. In this fashion, it is possible for software to read one of these status SPRs, and then write to it using the same data which was read. Any bits which were read as 1 will then be cleared, and any bits which were not yet set at the time the SPR was read will be left unchanged. If any of these previously clear bits happen to be set between the time the SPR is read and when it is written, then when the SPR is later read again, software will observe any newly set bits. If it were not for this behavior, then software could erroneously clear bits which it had not yet observed as having been set, and overlook the occurrence of certain exceptions.

Table 15-1. Special Purpose Registers Sorted by SPR Number

Mnemonic	Register Name	SPRN	Model	Access
XER	Integer Exception Register	0x001	User	Read/Write
LR	Link Register	0x008	User	Read/Write
CTR	Count Register	0x009	User	Read/Write
DEC	Decrementer	0x016	Supervisor	Read/Write
SRR0	Save/Restore Register 0	0x01A	Supervisor	Read/Write
SRR1	Save/Restore Register 1	0x01B	Supervisor	Read/Write
PID	Process ID	0x030	Supervisor	Read/Write
DECAR	Decrementer Auto-Reload	0x036	Supervisor	Write-only
CSRR0	Critical Save/Restore Register 0	0x03A	Supervisor	Read/Write
CSRR1	Critical Save/Restore Register 1	0x03B	Supervisor	Read/Write
DEAR	Data Exception Address Register	0x03D	Supervisor	Read/Write
ESR	Exception Syndrome Register	0x03E	Supervisor	Read/Write
IVPR	Interrupt Vector Prefix Register	0x03F	Supervisor	Read/Write
USPRG0	User Special Purpose Register General 0	0x100	User	Read/Write

Table 15-1. Special Purpose Registers Sorted by SPR Number (continued)

Mnemonic	Register Name	SPRN	Model	Access
SPRG4	Special Purpose Register General 4	0x104	User	Read-only
SPRG5	Special Purpose Register General 5	0x105	User	Read-only
SPRG6	Special Purpose Register General 6	0x106	User	Read-only
SPRG7	Special Purpose Register General 7	0x107	User	Read-only
TBL	Time Base Lower	0x10C	User	Read-only
TBU	Time Base Upper	0x10D	User	Read-only
SPRG0	Special Purpose Register General 0	0x110	Supervisor	Read/Write
SPRG1	Special Purpose Register General 1	0x111	Supervisor	Read/Write
SPRG2	Special Purpose Register General 2	0x112	Supervisor	Read/Write
SPRG3	Special Purpose Register General 3	0x113	Supervisor	Read/Write
SPRG4	Special Purpose Register General 4	0x114	Supervisor	Write-only
SPRG5	Special Purpose Register General 5	0x115	Supervisor	Write-only
SPRG6	Special Purpose Register General 6	0x116	Supervisor	Write-only
SPRG7	Special Purpose Register General 7	0x117	Supervisor	Write-only
TBL	Time Base Lower	0x11C	Supervisor	Write-only
TBU	Time Base Upper	0x11D	Supervisor	Write-only
PIR	Processor ID Register	0x11E	Supervisor	Read-only
PVR	Processor Version Register	0x11F	Supervisor	Read-only
DBSR	Debug Status Register	0x130	Supervisor	Read/Clear
DBCR0	Debug Control Register 0	0x134	Supervisor	Read/Write
DBCR1	Debug Control Register 1	0x135	Supervisor	Read/Write
DBCR2	Debug Control Register 2	0x136	Supervisor	Read/Write
IAC1	Instruction Address Compare 1	0x138	Supervisor	Read/Write
IAC2	Instruction Address Compare 2	0x139	Supervisor	Read/Write
IAC3	Instruction Address Compare 3	0x13A	Supervisor	Read/Write
IAC4	Instruction Address Compare 4	0x13B	Supervisor	Read/Write
DAC1	Data Address Compare 1	0x13C	Supervisor	Read/Write
DAC2	Data Address Compare 2	0x13D	Supervisor	Read/Write
DVC1	Data Value Compare 1	0x13E	Supervisor	Read/Write
DVC2	Data Value Compare 2	0x13F	Supervisor	Read/Write
TSR	Timer Status Register	0x150	Supervisor	Read/Clear
TCR	Timer Control Register	0x154	Supervisor	Read/Write
IVOR0	Interrupt Vector Offset Register 0	0x190	Supervisor	Read/Write
IVOR1	Interrupt Vector Offset Register 1	0x191	Supervisor	Read/Write
IVOR2	Interrupt Vector Offset Register 2	0x192	Supervisor	Read/Write
IVOR3	Interrupt Vector Offset Register 3	0x193	Supervisor	Read/Write
IVOR4	Interrupt Vector Offset Register 4	0x194	Supervisor	Read/Write
IVOR5	Interrupt Vector Offset Register 5	0x195	Supervisor	Read/Write

Table 15-1. Special Purpose Registers Sorted by SPR Number (continued)

Mnemonic	Register Name	SPRN	Model	Access
IVOR6	Interrupt Vector Offset Register 6	0x196	Supervisor	Read/Write
IVOR7	Interrupt Vector Offset Register 7	0x197	Supervisor	Read/Write
IVOR8	Interrupt Vector Offset Register 8	0x198	Supervisor	Read/Write
IVOR9	Interrupt Vector Offset Register 9	0x199	Supervisor	Read/Write
IVOR10	Interrupt Vector Offset Register 10	0x19A	Supervisor	Read/Write
IVOR11	Interrupt Vector Offset Register 11	0x19B	Supervisor	Read/Write
IVOR12	Interrupt Vector Offset Register 12	0x19C	Supervisor	Read/Write
IVOR13	Interrupt Vector Offset Register 13	0x19D	Supervisor	Read/Write
IVOR14	Interrupt Vector Offset Register 14	0x19E	Supervisor	Read/Write
IVOR15	Interrupt Vector Offset Register 15	0x19F	Supervisor	Read/Write
MCSRR0	Machine Check Save Restore Register 0	0x23A	Supervisor	Read/Write
MCSRR1	Machine Check Save Restore Register 1	0x23B	Supervisor	Read/Write
MCSR	Machine Check Status Register	0x23C	Supervisor	Read/Write
INV0	Instruction Cache Normal Victim 0	0x370	Supervisor	Read/Write
INV1	Instruction Cache Normal Victim 1	0x371	Supervisor	Read/Write
INV2	Instruction Cache Normal Victim 2	0x372	Supervisor	Read/Write
INV3	Instruction Cache Normal Victim 3	0x373	Supervisor	Read/Write
ITV0	Instruction Cache Transient Victim 0	0x374	Supervisor	Read/Write
ITV1	Instruction Cache Transient Victim 1	0x375	Supervisor	Read/Write
ITV2	Instruction Cache Transient Victim 2	0x376	Supervisor	Read/Write
ITV3	Instruction Cache Transient Victim 3	0x377	Supervisor	Read/Write
CCR1	Core Configuration Register 1	0x378	Supervisor	Read/Write
DCRIPR	Device Control Register Immediate Prefix Register	0x37B	Supervisor	Read/Write
DNV0	Data Cache Normal Victim 0	0x390	Supervisor	Read/Write
DNV1	Data Cache Normal Victim 1	0x391	Supervisor	Read/Write
DNV2	Data Cache Normal Victim 2	0x392	Supervisor	Read/Write
DNV3	Data Cache Normal Victim 3	0x393	Supervisor	Read/Write
DTV0	Data Cache Transient Victim 0	0x394	Supervisor	Read/Write
DTV1	Data Cache Transient Victim 1	0x395	Supervisor	Read/Write
DTV2	Data Cache Transient Victim 2	0x396	Supervisor	Read/Write
DTV3	Data Cache Transient Victim 3	0x397	Supervisor	Read/Write
DVLIM	Data Cache Victim Limit	0x398	Supervisor	Read/Write
IVLIM	Instruction Cache Victim Limit	0x399	Supervisor	Read/Write
RSTCFG	Reset Configuration	0x39B	Supervisor	Read-only
DCDBTRL	Data Cache Debug Tag Register Low	0x39C	Supervisor	Read-only
DCDBTRH	Data Cache Debug Tag Register High	0x39D	Supervisor	Read-only
ICDBTRL	Instruction Cache Debug Tag Register Low	0x39E	Supervisor	Read-only
ICDBTRH	Instruction Cache Debug Tag Register High	0x39F	Supervisor	Read-only

Table 15-1. Special Purpose Registers Sorted by SPR Number (continued)

Mnemonic	Register Name	SPRN	Model	Access
MMUCR	Memory Management Unit Control Register	0x3B2	Supervisor	Read/Write
CCR0	Core Configuration Register 0	0x3B3	Supervisor	Read/Write
ICDBDR	Instruction Cache Debug Data Register	0x3D3	Supervisor	Read-only
DBDR	Debug Data Register	0x3F3	Supervisor	Read/Write

15.2 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.

15.3 Alphabetical Listing of Processor Core Registers

The following table lists the processor core registers available in the PPC465. For each register, the following information is supplied:

- Register mnemonic.

  Note: Note that these registers, unlike those associated with functional units outside the processor core, have no prefix.
- Register type.
- Register description.
- Register number or address.
- Register programming model (User or Supervisor).
- Access (Read/Clear, Read-Only, Read/Write, Write-Only).
- Cross reference to detailed register information.

Table 15-2. Alphabetical Listing of Processor Core Registers

Register	Type	Description	Address	Model	Access	See page
CCR0	SPR	Core Configuration Register 0	0x3B3	Supervisor	R/W	73
CCR1	SPR	Core Configuration Register 1	0x378	Supervisor	R/W	75
CR	CR	Condition Register	NA	User	R/W	66
CSRR0	SPR	Critical Save/Restore Register 0	0x03A	Supervisor	R/W	255
CSRR1	SPR	Critical Save/Restore Register 1	0x03B	Supervisor	R/W	255
CTR	SPR	Count Register	0x009	User	R/W	66
DAC1	SPR	Data Address Compare 1	0x13C	Supervisor	R/W	340
DAC2	SPR	Data Address Compare 2	0x13D	Supervisor	R/W	340

Table 15-2. Alphabetical Listing of Processor Core Registers (continued)

Register	Type	Description	Address	Model	Access	See page
DBCR0	SPR	Debug Control Register 0	0x134	Supervisor	R/W	335
DBCR1	SPR	Debug Control Register 1	0x135	Supervisor	R/W	336
DBCR2	SPR	Debug Control Register 2	0x136	Supervisor	R/W	338
DBDR	SPR	Debug Data Register	0x3F3	Supervisor	R/W	341
DBSR	SPR	Debug Status Register	0x130	Supervisor	Read/Clear	339
DCDBTRH	SPR	Data Cache Debug Tag Register High	0x39D	Supervisor	Read-only	151
DCDBTRL	SPR	Data Cache Debug Tag Register Low	0x39C	Supervisor	Read-only	151
DCRIPR	SPR	Device Control Register Immediate Prefix Register	0x37B	Supervisor	R/W	66
DEAR	SPR	Data Exception Address Register	0x03D	Supervisor	R/W	256
DEC	SPR	Decrementer	0x016	Supervisor	R/W	309
DECAR	SPR	Decrementer Auto-Reload	0x036	Supervisor	Write-only	309
DNV0	SPR	Data Cache Normal Victim 0	0x390	Supervisor	R/W	126
DNV1	SPR	Data Cache Normal Victim 1	0x391	Supervisor	R/W	126
DNV2	SPR	Data Cache Normal Victim 2	0x392	Supervisor	R/W	126
DNV3	SPR	Data Cache Normal Victim 3	0x393	Supervisor	R/W	126
DTV0	SPR	Data Cache Transient Victim 0	0x394	Supervisor	R/W	126
DTV1	SPR	Data Cache Transient Victim 1	0x395	Supervisor	R/W	126
DTV2	SPR	Data Cache Transient Victim 2	0x396	Supervisor	R/W	126
DTV3	SPR	Data Cache Transient Victim 3	0x397	Supervisor	R/W	126
DVC1	SPR	Data Value Compare 1	0x13E	Supervisor	R/W	340
DVC2	SPR	Data Value Compare 2	0x13F	Supervisor	R/W	340
DVLIM	SPR	Data Cache Victim Limit	0x398	Supervisor	R/W	127
ESR	SPR	Exception Syndrome Register	0x03E	Supervisor	R/W	258
GPR (R0:R31)	GPR	General Purpose Registers	NA	User	R/W	69
IAC1	SPR	Instruction Address Compare 1	0x138	Supervisor	R/W	340
IAC2	SPR	Instruction Address Compare 2	0x139	Supervisor	R/W	340
IAC3	SPR	Instruction Address Compare 3	0x13A	Supervisor	R/W	340
IAC4	SPR	Instruction Address Compare 4	0x13B	Supervisor	R/W	340
ICDBDR	SPR	Instruction Cache Debug Data Register	0x3D3	Supervisor	Read-only	136
ICDBTRH	SPR	Instruction Cache Debug Tag Register High	0x39F	Supervisor	Read-only	136
ICDBTRL	SPR	Instruction Cache Debug Tag Register Low	0x39E	Supervisor	Read-only	136
INV0	SPR	Instruction Cache Normal Victim 0	0x370	Supervisor	R/W	126
INV1	SPR	Instruction Cache Normal Victim 1	0x371	Supervisor	R/W	126
INV2	SPR	Instruction Cache Normal Victim 2	0x372	Supervisor	R/W	126
INV3	SPR	Instruction Cache Normal Victim 3	0x373	Supervisor	R/W	126
ITV0	SPR	Instruction Cache Transient Victim 0	0x374	Supervisor	R/W	126
ITV1	SPR	Instruction Cache Transient Victim 1	0x375	Supervisor	R/W	126
ITV2	SPR	Instruction Cache Transient Victim 2	0x376	Supervisor	R/W	126

Table 15-2. Alphabetical Listing of Processor Core Registers (continued)

Register	Type	Description	Address	Model	Access	See page
ITV3	SPR	Instruction Cache Transient Victim 3	0x377	Supervisor	R/W	126
IVLIM	SPR	Instruction Cache Victim Limit	0x399	Supervisor	R/W	127
IVOR0	SPR	Interrupt Vector Offset Register 0	0x190	Supervisor	R/W	257
IVOR1	SPR	Interrupt Vector Offset Register 1	0x191	Supervisor	R/W	257
IVOR2	SPR	Interrupt Vector Offset Register 2	0x192	Supervisor	R/W	257
IVOR3	SPR	Interrupt Vector Offset Register 3	0x193	Supervisor	R/W	257
IVOR4	SPR	Interrupt Vector Offset Register 4	0x194	Supervisor	R/W	257
IVOR5	SPR	Interrupt Vector Offset Register 5	0x195	Supervisor	R/W	257
IVOR6	SPR	Interrupt Vector Offset Register 6	0x196	Supervisor	R/W	257
IVOR7	SPR	Interrupt Vector Offset Register 7	0x197	Supervisor	R/W	257
IVOR8	SPR	Interrupt Vector Offset Register 8	0x198	Supervisor	R/W	257
IVOR9	SPR	Interrupt Vector Offset Register 9	0x199	Supervisor	R/W	257
IVOR10	SPR	Interrupt Vector Offset Register 10	0x19A	Supervisor	R/W	257
IVOR11	SPR	Interrupt Vector Offset Register 11	0x19B	Supervisor	R/W	257
IVOR12	SPR	Interrupt Vector Offset Register 12	0x19C	Supervisor	R/W	257
IVOR13	SPR	Interrupt Vector Offset Register 13	0x19D	Supervisor	R/W	257
IVOR14	SPR	Interrupt Vector Offset Register 14	0x19E	Supervisor	R/W	257
IVOR15	SPR	Interrupt Vector Offset Register 15	0x19F	Supervisor	R/W	257
IVPR	SPR	Interrupt Vector Prefix Register	0x03F	Supervisor	R/W	258
LR	SPR	Link Register	0x008	User	R/W	66
MCSR	SPR	Machine Check Status Register	0x23C	Supervisor	R/W	260
MCSRR0	SPR	Machine Check Save Restore Register 0	0x23A	Supervisor	R/W	255
MCSRR1	SPR	Machine Check Save Restore Register 1	0x23B	Supervisor	R/W	256
MMUCR	SPR	Memory Management Unit Control Register	0x3B2	Supervisor	R/W	236
MSR	MSR	Machine State Register	NA	Supervisor	R/W	253
PID	SPR	Process ID	0x030	Supervisor	R/W	239
PIR	SPR	Processor ID Register	0x11E	Supervisor	Read-only	73
PVR	SPR	Processor Version Register	0x11F	Supervisor	Read-only	73
RSTCFG	SPR	Reset Configuration	0x39B	Supervisor	Read-only	76
SPRG0	SPR	Special Purpose Register General 0	0x110	Supervisor	R/W	72
SPRG1	SPR	Special Purpose Register General 1	0x111	Supervisor	R/W	72
SPRG2	SPR	Special Purpose Register General 2	0x112	Supervisor	R/W	72
SPRG3	SPR	Special Purpose Register General 3	0x113	Supervisor	R/W	72
SPRG4	SPR	Special Purpose Register General 4	0x104	User	Read-only	72
SPRG4	SPR	Special Purpose Register General 4	0x114	Supervisor	Write-only	72
SPRG5	SPR	Special Purpose Register General 5	0x105	User	Read-only	72
SPRG5	SPR	Special Purpose Register General 5	0x115	Supervisor	Write-only	72
SPRG6	SPR	Special Purpose Register General 6	0x106	User	Read-only	72

Table 15-2. Alphabetical Listing of Processor Core Registers (continued)

Register	Type	Description	Address	Model	Access	See page
SPRG6	SPR	Special Purpose Register General 6	0x116	Supervisor	Write-only	72
SPRG7	SPR	Special Purpose Register General 7	0x107	User	Read-only	72
SPRG7	SPR	Special Purpose Register General 7	0x117	Supervisor	Write-only	72
SRR0	SPR	Save/Restore Register 0	0x01A	Supervisor	R/W	254
SRR1	SPR	Save/Restore Register 1	0x01B	Supervisor	R/W	254
TBL	SPR	Time Base Lower	0x10C	User	Read-only	308
TBL	SPR	Time Base Lower	0x11C	Supervisor	Write-only	308
TBU	SPR	Time Base Upper	0x10D	User	Read-only	308
TBU	SPR	Time Base Upper	0x11D	Supervisor	Write-only	308
TCR	SPR	Timer Control Register	0x154	Supervisor	R/W	312
TSR	SPR	Timer Status Register	0x150	Supervisor	Read/Clear	313
USPRG0	SPR	User Special Purpose Register General 0	0x100	User	R/W	72
XER	SPR	Integer Exception Register	0x001	User	R/W	70

15.3.1 L2 Cache DCR Registers

There are two types of DCR's in the L2C core, architected and indirect. The architected DCR's are accessed directly on the DCR bus using their DCR number.

The architected DCRs are:
- L2DCDCRAI (L2 D-Cache DCR Address Indirect)
- L2DCDCRDI (L2 D-Cache DCR Data Indirect)

Table 15-3. L2DCDCRAI and L2DCDCRDI *Register Summary*

Mnemonic	Register	DCR Number	Access	Privileged	Page
L2DCDCRAI	L2 D-Cache DCR Address Indirect	0x0000	RW	Yes	182
L2DCDCRDI	L2 D-Cache DCR Data Indirect	0x0001	RW	Yes	182

The indirect DCRs for the L2 Cache are read and written using indirect addressing. In order to access an L2C indirect DCR, software must perform the following sequence:
- Write the L2DCDCRAI (using mtdcr) with the L2 Internal DCR number that software wishes to read or write.
- Issue a mtdcr or mfdcr using the L2DCDCRDI DCR number. This will cause the L2 to read or write the DCR currently pointed to by L2DCDCRAI.

Each indirect DCR has the following associated information:
- DCR Address: DCR Index number
- Valid Bits: valid data bits for the DCR
- Privileged: whether or not this DCR is privileged
- Access: the type of access permitted - R=read, W=write, S=set, C=clear
- Reset value: DCR value after reset
- Internal update: whether or not the L2C can update the DCR directly

15.3.1.1 L2C Indirect DCR Summary

The L2 Cache device control registers are written and read in an indirect fashion using DCR ports L2DCDCRAI (for address pointer) and L2DCDCRDI (for the data). *Table 15-4* only lists the lower offset address assignment for the L2 DCR registers.

Table 15-4 summarizes the L2 cache device control registers sorted by DCR number.

Table 15-4. L2 Cache Device Control Register Summary

Mnemonic	Register	DCR Indirect Number	Access	Privileged	Page
L2CR0	L2 Configuration Register 0	0x00	RW	Yes	182
L2CR1	L2 Configuration Register 1	0x01	RW	Yes	184
L2CR2	L2 Configuration Register 2	0x02	RW	Yes	185
L2CR3	L2 Configuration Register 3	0x03	RW	Yes	186
L2ERAPR	L2 Extended Real Address Prefix Register	0x08	RW	Yes	188
L2SLVERAPR	PLB Slave Extended Real Address Prefix Register	0x09	RW	Yes	188
L2FAMAR	Fixed Address Mode Address Register	0x0A	RW	Yes	189
L2REVID	L2 Revision ID Register	0x0B	RO	Yes	190
L2COPR	L2 Cache Op Register	0x0C	RW	Yes	187
L2THRR	L2 Throttle Register	0x0D	RW	Yes	190
L2FER0	Force Error Register 0	0x0E	RW	Yes	194
L2FER1	Force Error Register 1	0x0F	RW	Yes	195
L2MCSR	Machine Check Status Register	0x10	RW	Yes	197
L2MCRER	Machine Check Reporting Enable Register	0x11	RW	Yes	196
L2ECCIDX	ECC Index Register	0x12	RO	Yes	189
L2SLVEAR0	PLB Slave Port Error Address Register 0	0x14	RO	Yes	217
L2SLVEAR1	PLB Slave Port Error Address Register 1	0x15	RO	Yes	217
L2SLVMIDR	PLB Slave Port Error Master ID Register	0x16	RO	Yes	218
L2DBACMR	L2 Debug Address Compare Mask Register	0x17	RW	Yes	199
L2DBSR	L2 Debug Status Register	0x18	RC	Yes	200
L2DBCR	L2 Debug Control Register	0x19	RW	Yes	200
L2DBACSR	L2 Debug Address Compare Status Register	0x1A	RC	Yes	203
L2DBACCR	L2 Debug Address Compare Control Register	0x1B	RW	Yes	203
L2SLVAC0	Slave Address Compare 0 Register	0x1C	RW	Yes	202
L2DBACR	L2 Debug Address Compare Register	0x1D	RW	Yes	203
L2SNPAC0	Snoop Address Compare 0 Register	0x1E	RW	Yes	202
L2SNPAC1	Snoop Address Compare 1 Register	0x1F	RW	Yes	202
L2DBDR0	L2 Debug Data Register 0	0x20	RO	Yes	191

Table 15-4. L2 Cache Device Control Register Summary (continued)

Mnemonic	Register	DCR Indirect Number	Access	Privileged	Page
L2DBDR1	L2 Debug Data Register 1	0x21	RO	Yes	191
L2DBDR2	L2 Debug Data Register 2	0x22	RO	Yes	192
L2DBDR3	L2 Debug Data Register 3	0x23	RO	Yes	192
L2DBDR4	L2 Debug Data Register 4	0x24	RO	Yes	192
L2DBTR0	L2 Debug Tag Register 0	0x25	RO	Yes	193
L2DBTR1	L2 Debug Tag Register 1	0x26	RO	Yes	193
L2DBTR2	L2 Debug Tag Register 2	0x27	RO	Yes	194
L2MCSRS	L2 Machine-Check Status Register (set)	0x30	S	Yes	n/a
L2DBSRS	L2 Debug Status Register (set)	0x38	S	Yes	n/a
L2DBACSRS	L2 Debug Address Compare Status Register (set)	0x3A	S	Yes	n/a

Appendix A. Instruction Summary

This appendix describes the various instruction formats, and lists all of the PPC465 instructions summarized alphabetically and by opcode.

Appendix A.1 on page 609 illustrates the PPC465 instruction forms (allowed arrangements of fields within instructions).

Appendix A.2 on page 614 lists all PPC465 mnemonics, including extended mnemonics. A short functional description is included for each mnemonic.

Appendix A.3 on page 643 identifies those opcodes which are allocated by PowerPC Book-E for implementation-dependent usage, including auxiliary processors.

Appendix A.4 on page 643 identifies those opcodes which are identified by PowerPC Book-E as “preserved” for compatibility with previous versions of the architecture.

Appendix A.5 on page 644 indentifies those opcodes which are “reserved” for use by future versions of the architecture.

Appendix A.6 on page 644, lists all instructions implemented within the PPC465, sorted by primary and secondary opcodes. Extended mnemonics are not included in the opcode list, but allocated, preserved, and reserved-nop opcodes are included.

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 PPC465 executes all invalid instruction forms without causing an illegal instruction exception.

A.1.1 Instruction Fields

PPC465 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 “Branch Instruction BO Field” on page 64.
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 two’s-complement integer displacement for load/store instructions.
DCRF (11:20)	Specifies a device control register (DCR). This field represents the DCR Number (DCRN) with the upper and lower five bits reversed (that is, DCRF = DCRN[5:9] || DCRN[0:4]).
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.
MO (6:10)	Memory Ordering. Provides a storage ordering function for storage accesses executing prior to an **mbar** instruction. MO is ignored and treated as 0 in the PPC465 CPU core.
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 66 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). This field represents the SPR Number (SPRN) with the upper and lower five bits reversed (that is, SPRF = SPRN[5:9] || SPRN[0:4]).
TO (6:10)	Specifies the conditions on which to trap, as described under **tw** and **twi** instructions.
WS (16:20)	Specifies the portion of a TLB entry to be read/written by **tlbre**/**tlbwe**.
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-5* on page 645 indicates which "form" is utilized by each PPC465 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	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

OPCD	RT	RA	RB	XO	Rc
OPCD	RT	RA	RB	XO	/
OPCD	RT	RA	NB	XO	/
OPCD	RT	RA	WS	XO	/
OPCD	RT	///	RB	XO	/
OPCD	RT	///	///	XO	/
OPCD	RS	RA	RB	XO	Rc
OPCD	RS	RA	RB	XO	1
OPCD	RS	RA	RB	XO	/
OPCD	RS	RA	NB	XO	/
OPCD	RS	RA	WS	XO	/
OPCD	RS	RA	SH	XO	Rc
OPCD	RS	RA	///	XO	Rc
OPCD	RS	///	RB	XO	/
OPCD	RS	///	///	XO	/
OPCD	BF / L	RA	RB	XO	/
OPCD	BF //	BFA //	///	XO	Rc
OPCD	BF //	///	///	XO	/
OPCD	BF //	///	U	XO	Rc
OPCD	BF //	///	///	XO	/
OPCD	TO	RA	RB	XO	/
OPCD	BT	///	///	XO	Rc
OPCD	MO	///	///	XO	/
OPCD	///	RA	RB	XO	/
OPCD	///	///	///	XO	/
OPCD	///	///	E //	XO	/
0	6	11	16	21	31

A.1.2.6 XL-Form

Figure A-6. XL Instruction Format

OPCD	BT		BA		BB	XO	/
OPCD	BC		BI		///	XO	LK
OPCD	BF	//	BFA	//	///	XO	/
OPCD	///		///		///	XO	/
0	6		11		16	21	31

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 XO-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 Alphabetical Summary of Implemented Instructions

Table A-1 summarizes the PPC465 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 Chapter 13, “Instruction Set,” which is also alphabetized under the root form. *Section 13* also describes the instruction operands and notation.

Programming Note: Bit 4 of the BO instruction field provides a hint about the most likely outcome of a conditional branch. (See “Branch Prediction” on page 65 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 Table A-1, BO_4 = 0 has always been assumed. The assembler must enable the programmer to specify branch prediction. To do this, the assembler supports suffixes for the conditional branch mnemonics:

+ Predict branch to be taken.

– Predict branch not to be taken.

For example, **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 65 for more information.

Table A-1. PPC465 Instruction Syntax Summary

Mnemonic	Operands	Function	Other Registers Changed	Page
add	RT, RA, RB	Add (RA) to (RB). Place result in RT.		348
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].		349
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].		350
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.		351
addic	RT, RA, IM	Add EXTS(IM) to (RA|0). Place result in RT. Place carry-out in XER[CA].		352
addic.	RT, RA, IM	Add EXTS(IM) to (RA|0). Place result in RT. Place carry-out in XER[CA].	CR[CR0]	353
addis	RT, RA, IM	Add (IM || ^{16}0) to (RA|0). Place result in RT.		354
addme	RT, RA	Add XER[CA], (RA), (-1). Place result in RT. Place carry-out in XER[CA].		355
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].		356
addze.			CR[CR0]	
addzeo			XER[SO, OV]	
addzeo.			CR[CR0] XER[SO, OV]	

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
and	RA, RS, RB	AND (RS) with (RB). Place result in RA.		357
and.	RA, RS, RB	AND (RS) with (RB). Place result in RA.	CR[CR0]	357
andc	RA, RS, RB	AND (RS) with ¬(RB). Place result in RA.		358
andc.	RA, RS, RB	AND (RS) with ¬(RB). Place result in RA.	CR[CR0]	358
andi.	RA, RS, IM	AND (RS) with ($^{16}0 \parallel$ IM). Place result in RA.	CR[CR0]	359
andis.	RA, RS, IM	AND (RS) with (IM $\parallel$ $^{16}0$). Place result in RA.	CR[CR0]	360
b	target	Branch unconditional relative. LI $\leftarrow$ (target – CIA)$_{6:29}$ NIA $\leftarrow$ CIA + EXTS(LI $\parallel$ $^{2}0$)		361
ba	target	Branch unconditional absolute. LI $\leftarrow$ target$_{6:29}$ NIA $\leftarrow$ EXTS(LI $\parallel$ $^{2}0$)		361
bl	target	Branch unconditional relative. LI $\leftarrow$ (target – CIA)$_{6:29}$ NIA $\leftarrow$ CIA + EXTS(LI $\parallel$ $^{2}0$)	(LR) $\leftarrow$ CIA + 4	361
bla	target	Branch unconditional absolute. LI $\leftarrow$ target$_{6:29}$ NIA $\leftarrow$ EXTS(LI $\parallel$ $^{2}0$)	(LR) $\leftarrow$ CIA + 4	361
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	362
bca	BO, BI, target	Branch conditional absolute. BD $\leftarrow$ target$_{16:29}$ NIA $\leftarrow$ EXTS(BD $\parallel$ $^{2}0$)	CTR if BO$_2$ = 0	362
bcl	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 (LR) $\leftarrow$ CIA + 4	362
bcla	BO, BI, target	Branch conditional absolute. BD $\leftarrow$ target$_{16:29}$ NIA $\leftarrow$ EXTS(BD $\parallel$ $^{2}0$)	CTR if BO$_2$ = 0 (LR) $\leftarrow$ CIA + 4	362
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	367
bcctrl	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 (LR) $\leftarrow$ CIA + 4	367
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	370
bclrl	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 (LR) $\leftarrow$ CIA + 4	370
bctr		Branch unconditionally to address in CTR. *Extended mnemonic for* **bcctr 20,0**		367
bctrl		*Extended mnemonic for* **bcctrl 20,0**	(LR) $\leftarrow$ CIA + 4	367
bdnz	target	Decrement CTR. Branch if CTR $\neq$ 0 *Extended mnemonic for* **bc 16,0,target**		362
bdnza	target	*Extended mnemonic for* **bca 16,0,target**		362
bdnzl	target	*Extended mnemonic for* **bcl 16,0,target**	(LR) $\leftarrow$ CIA + 4	362
bdnzla	target	*Extended mnemonic for* **bcla 16,0,target**	(LR) $\leftarrow$ CIA + 4	362

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bdnzlr		Decrement CTR. Branch if CTR ≠ 0 to address in LR. *Extended mnemonic for* **bclr 16,0**		370
bdnzlrl		*Extended mnemonic for* **bclrl 16,0**	(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**		362
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**		370
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**		362
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**		370
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**		362
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**		370
bdzlrl		*Extended mnemonic for* **bclrl 18,0**	(LR) ← CIA + 4	

Table A-1. PPC465 Instruction Syntax Summary (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**		362
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**		370
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**		362
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**		370
bdztlrl		*Extended mnemonic for* **bclrl 10,cr_bit**	(LR) ← CIA + 4	
beq	[cr_field], target	Branch if equal. UseCR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 12,4*cr_field+2,target**		362
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. UseCR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4*cr_field+2**		367
beqctrl		*Extended mnemonic for* **bcctrl 12,4*cr_field+2**	(LR) ← CIA + 4	
beqlr	[cr_field]	Branch if equal to address in LR. UseCR[CR0] if cr_field is omitted. *Extended mnemonic for* **bclr 12,4*cr_field+2**		370
beqlrl		*Extended mnemonic for* **bclrl 12,4*cr_field+2**	(LR) ← CIA + 4	

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bf	cr_bit, target	Branch if CR_{cr_bit} = 0. *Extended mnemonic for* **bc 4,cr_bit,target**		362
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**		367
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**		370
bflrl		*Extended mnemonic for* **bclrl 4,cr_bit**	(LR) ← CIA + 4	
bge	[cr_field], target	Branch if greater than or equal. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+0,target**		362
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4∗cr_field+0**		367
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+0**		370
bgelrl		*Extended mnemonic for* **bclrl 4,4∗cr_field+0**	(LR) ← CIA + 4	
bgt	[cr_field], target	Branch if greater than. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 12,4∗cr_field+1,target**		362
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	

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bgtctr	[cr_field]	Branch if greater than to address in CTR. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4∗cr_field+1**		367
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bclr 12,4∗cr_field+1**		370
bgtlrl		*Extended mnemonic for* **bclrl 12,4∗cr_field+1**	(LR) ← CIA + 4	
ble	[cr_field], target	Branch if less than or equal. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 4,4∗cr_field+1,target**		362
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4*cr_field+1**		367
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bclr 4,4∗cr_field+1**		370
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**		370
blrl		*Extended mnemonic for* **bclrl 20,0**	(LR) ← CIA + 4	
blt	[cr_field], target	Branch if less than. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 12,4*cr_field+0,target**		362
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4*cr_field+0**		367
bltctrl		*Extended mnemonic for* **bcctrl 12,4*cr_field+0**	(LR) ← CIA + 4	

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bltlr	[cr_field]	Branch if less than to address in LR. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bclr 12,4*cr_field+0**		370
bltlrl		*Extended mnemonic for* **bclrl 12,4*cr_field+0**	(LR) ← CIA + 4	
bne	[cr_field], target	Branch if not equal. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 4,4*cr_field+2,target**		362
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4*cr_field+2**		367
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bclr 4,4*cr_field+2**		370
bnelrl		*Extended mnemonic for* **bclrl 4,4*cr_field+2**	(LR) ← CIA + 4	
bng	[cr_field], target	Branch if not greater than. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 4,4*cr_field+1,target**		362
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4*cr_field+1**		367
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bclr 4,4*cr_field+1**		370
bnglrl		*Extended mnemonic for* **bclrl 4,4*cr_field+1**	(LR) ← CIA + 4	

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bnl	[cr_field], target	Branch if not less than. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 4,4*cr_field+0,target**		362
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4*cr_field+0**		367
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bclr 4,4*cr_field+0**		370
bnllrl		*Extended mnemonic for* **bclrl 4,4*cr_field+0**	(LR) ← CIA + 4	
bns	[cr_field], target	Branch if not summary overflow. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 4,4*cr_field+3,target**		362
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4*cr_field+3**		367
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bclr 4,4*cr_field+3**		370
bnslrl		*Extended mnemonic for* **bclrl 4,4*cr_field+3**	(LR) ← CIA + 4	
bnu	[cr_field], target	Branch if not unordered. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 4,4*cr_field+3,target**		362
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	

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bnuctr	[cr_field]	Branch if not unordered to address in CTR. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 4,4*cr_field+3**		367
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bclr 4,4*cr_field+3**		370
bnulrl		*Extended mnemonic for* **bclrl 4,4*cr_field+3**	(LR) ← CIA + 4	
bso	[cr_field], target	Branch if summary overflow. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 12,4*cr_field+3,target**		362
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. UseCR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4*cr_field+3**		367
bsoctrl		*Extended mnemonic for* **bcctrl 12,4*cr_field+3**	(LR) ← CIA + 4	
bsolr	[cr_field]	Branch if summary overflow to address in LR. UseCR[CR0] if cr_field is omitted. *Extended mnemonic* for **bclr 12,4*cr_field+3**		370
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**		362
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**		367
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**		370
btlrl		*Extended mnemonic for* **bclrl 12,cr_bit**	(LR) ← CIA + 4	

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
bun	[cr_field], target	Branch if unordered. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bc 12,4*cr_field+3,target**		362
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 CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bcctr 12,4*cr_field+3**		367
bunctrl		*Extended mnemonic for* **bcctrl 12,4*cr_field+3**	(LR) ← CIA + 4	
bunlr	[cr_field]	Branch if unordered, to address in LR. Use CR[CR0] if cr_field is omitted. *Extended mnemonic for* **bclr 12,4*cr_field+3**		370
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**		495
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**		495
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**		495
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.		374
cmpi	BF, 0, RA, IM	Compare (RA) to EXTS(IM), signed. Results in CR[CRn], where n = BF.		375
cmpl	BF, 0, RA, RB	Compare (RA) to (RB), unsigned. Results in CR[CRn], where n = BF.		376
cmpli	BF, 0, RA, IM	Compare (RA) to $({}^{16}0 \parallel IM)$, unsigned. Results in CR[CRn], where n = BF.		377
cmplw	[BF,] RA, RB	Compare Logical Word. UseCR[CR0] if BF is omitted. *Extended mnemonic for* **cmpl BF,0,RA,RB**		376

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
cmplwi	[BF,] RA, IM	Compare Logical Word Immediate. UseCR[CR0] if BF is omitted. *Extended mnemonic for* **cmpli BF,0,RA,IM**		377
cmpw	[BF,] RA, RB	Compare Word. UseCR[CR0] if BF is omitted. *Extended mnemonic for* **cmp BF,0,RA,RB**		374
cmpwi	[BF,] RA, IM	Compare Word Immediate. UseCR[CR0] if BF is omitted. *Extended mnemonic for* **cmpi BF,0,RA,IM**		375
cntlzw	RA, RS	Count leading zeros in RS. Place result in RA.		378
cntlzw.			CR[CR0]	
crand	BT, BA, BB	AND bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		379
crandc	BT, BA, BB	AND bit (CR_{BA}) with $\neg$(CR_{BB}). Place result in CR_{BT}.		380
crclr	bx	Condition register clear. *Extended mnemonic for* **crxor bx,bx,bx**		386
creqv	BT, BA, BB	Equivalence of bit CR_{BA} with CR_{BB}. $CR_{BT} \leftarrow \neg(CR_{BA} \oplus CR_{BB})$		381
crmove	bx, by	Condition register move. *Extended mnemonic for* **cror bx,by,by**		384
crnand	BT, BA, BB	NAND bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		382
crnor	BT, BA, BB	NOR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		383
crnot	bx, by	Condition register not. *Extended mnemonic for* **crnor bx,by,by**		383
cror	BT, BA, BB	OR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		384
crorc	BT, BA, BB	OR bit (CR_{BA}) with $\neg$(CR_{BB}). Place result in CR_{BT}.		385
crset	bx	Condition register set. *Extended mnemonic for* **creqv bx,bx,bx**		381
crxor	BT, BA, BB	XOR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT}.		386
dcba	RA, RB	Treated as a no-op.		387
dcbf	RA, RB	Flush (store, then invalidate) the data cache block which contains the effective address (RA|0) + (RB).		388
dcbi	RA, RB	Invalidate the data cache block which contains the effective address (RA|0) + (RB).		389
dcbst	RA, RB	Store the data cache block which contains the effective address (RA|0) + (RB).		390
dcbt	RA, RB	Load the data cache block which contains the effective address (RA|0) + (RB).		391
dcbtst	RA,RB	Load the data cache block which contains the effective address (RA|0) + (RB).		392
dcbz	RA, RB	Zero the data cache block which contains the effective address (RA|0) + (RB).		393

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
dccci	RA, RB	Invalidate the data cache array.		394
dcread	RT, RA, RB	Read tag and data information from the data cache line selected using effective address bits 17:26. The effective address is calculated by (RA|0) + (RB). Place the data word selected by effective address bits 27:29 in GPR RT; place the tag information in DCDBTRH and DCDBTRL.		395
divw	RT, RA, RB	Divide (RA) by (RB), signed. Place result in RT.		397
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.		398
divwu.			CR[CR0]	
divwuo			XER[SO, OV]	
divwuo.			CR[CR0] XER[SO, OV]	

dlmzb	RA, RS, RB	d ← (RS) || (RB)
		i, x, y ← 0
		do while (x < 8) ∧ (y = 0)
		    x ← x + 1
		    if $d_{i:i+7}$ = 0 then
		        y ← 1
		    else
		        i ← i + 8
		(RA) ← x
		XER[TBC] ← x
		if Rc = 1 then
		    $CR[CR0]_3$ ←XER[SO]
		    if y = 1 then
		        if x < 5 then
		        $CR[CR0]_{0:2}$ ← 0b010
		        else
		        $CR[CR0]_{0:2}$ ← 0b100
		    else
		        $CR[CR0]_{0:2}$ ← 0b001
			XER[TBC], RA	399

dlmzb.			XER[TBC], RA, CR[CR0]	
eqv	RA, RS, RB	Equivalence of (RS) with (RB). (RA) ← ¬((RS) ⊕ (RB))		400
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**		495
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**		495
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.		401
extsb.			CR[CR0]	
extsh	RA, RS	Extend the sign of halfword $(RS)_{16:31}$. Place the result in RA.		402
extsh.			CR[CR0]	
icbi	RA, RB	Invalidate the instruction cache block which contains the effective address (RA|0) + (RB).		403

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
icbt	RA, RB	Load the instruction cache block which contains the effective address (RA|0) + (RB).		401
iccci	RA, RB	Invalidate the instruction cache array.		406
icread	RA, RB	Read tag and data information from the instruction cache line selected using effective address bits 17:26. The effective address is calculated by (RA|0) + (RB). Place the instruction selected by effective address bits 27:29 in ICDBDR; place the tag information in ICDBTRH and ICDBTRL.		407
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**		494
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**		494
insrwi.		*Extended mnemonic for* **rlwimi. RA,RS,32−b−n,b,b+n−1**	CR[CR0]	
isel	RT, RA, RB, CRb	RT ← (RA|0) if CRb = 1, else RT ← (RB)		409
isync		Synchronize execution context by flushing the prefetch queue.		410
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**		351
lbz	RT, D(RA)	Load byte from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) ← $^{24}0$ || MS(EA,1).		411
lbzu	RT, D(RA)	Load byte from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) ← $^{24}0$ || MS(EA,1). Update the base address, (RA) ← EA.		412
lbzux	RT, RA, RB	Load byte from EA = (RA|0) + (RB) and pad left with zeroes, (RT) ← $^{24}0$ || MS(EA,1). Update the base address, (RA) ← EA.		413
lbzx	RT, RA, RB	Load byte from EA = (RA|0) + (RB) and pad left with zeroes, (RT) ← $^{24}0$ || MS(EA,1).		414
lha	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and sign extend, (RT) ← EXTS(MS(EA,2)).		415
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.		416
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.		417
lhax	RT, RA, RB	Load halfword from EA = (RA|0) + (RB) and sign extend, (RT) ← EXTS(MS(EA,2)).		418

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
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).		419
lhz	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) $\leftarrow$ $^{16}0$ || MS(EA,2).		420
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.		421
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.		422
lhzx	RT, RA, RB	Load halfword from EA = (RA|0) + (RB) and pad left with zeroes, (RT) $\leftarrow$ $^{16}0$ || MS(EA,2).		423
li	RT, IM	Load immediate. (RT) $\leftarrow$ EXTS(IM) *Extended mnemonic for* **addi RT,0,value**		351
lis	RT, IM	Load immediate shifted. (RT) $\leftarrow$ (IM || $^{16}0$) *Extended mnemonic for* **addis RT,0,value**		354
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).		424
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}.		425
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.		427
lwarx	RT, RA, RB	Load word from EA = (RA|0) + (RB) and place in RT, (RT) $\leftarrow$ MS(EA,4). Set the Reservation bit.		429
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).		431
lwz	RT, D(RA)	Load word from EA = (RA|0) + EXTS(D) and place in RT, (RT) $\leftarrow$ MS(EA,4).		432

Table A-1. PPC465 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) ← MS(EA,4). Update the base address, (RA) ← EA.		433
lwzux	RT, RA, RB	Load word from EA = (RA|0) + (RB) and place in RT, (RT) ← MS(EA,4). Update the base address, (RA) ← EA.		434
lwzx	RT, RA, RB	Load word from EA = (RA|0) + (RB) and place in RT, (RT) ← MS(EA,4).		435
macchw	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		436
macchw.			CR[CR0]	
macchwo			XER[SO, OV]	
macchwo.			CR[CR0] XER[SO, OV]	
macchwu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		439
macchwu.			CR[CR0]	
macchwuo			XER[SO, OV]	
macchwuo.			CR[CR0] XER[SO, OV]	
macchws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \vee {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		437
macchws.			CR[CR0]	
macchwso			XER[SO, OV]	
macchwso.			CR[CR0] XER[SO, OV]	
macchwsu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow (temp_{1:32} \vee {}^{31}temp_0)$		438
macchwsu.			CR[CR0]	
macchwsuo			XER[SO, OV]	
macchwsuo.			CR[CR0] XER[SO, OV]	
machhw	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		440
machhw.			CR[CR0]	
machhwo			XER[SO, OV]	
machhwo.			CR[CR0] XER[SO, OV]	
machhwu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		443
machhwu.			CR[CR0]	
machhwuo			XER[SO, OV]	
machhwuo.			CR[CR0] XER[SO, OV]	
machhws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \vee {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		441
machhws.			CR[CR0]	
machhwso			XER[SO, OV]	
machhwso.			CR[CR0] XER[SO, OV]	
machhwsu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow (temp_{1:32} \vee {}^{31}temp_0)$		442
machhwsu.			CR[CR0]	
machhwsuo			XER[SO, OV]	
machhwsuo.			CR[CR0] XER[SO, OV]	

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
maclhw	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		444
maclhw.			CR[CR0]	
maclhwo			XER[SO, OV]	
maclhwo.			CR[CR0] XER[SO, OV]	
maclhwu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		447
maclhwu.			CR[CR0]	
maclhwuo			XER[SO, OV]	
maclhwuo.			CR[CR0] XER[SO, OV]	
maclhws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \vee {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		445
maclhws.			CR[CR0]	
maclhwso			XER[SO, OV]	
maclhwso.			CR[CR0] XER[SO, OV]	
maclhwsu	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow prod_{0:31} + (RT)$ $(RT) \leftarrow (temp_{1:32} \vee {}^{31}temp_0)$		446
maclhwsu.			CR[CR0]	
maclhwsuo			XER[SO, OV]	
maclhwsuo.			CR[CR0] XER[SO, OV]	
mbar		Storage synchronization. All loads and stores that precede the **mbar** instruction complete before any loads and stores that follow the instruction access main storage.		448
mcrf	BF, BFA	Move CR field, (CR[CRn]) ← (CR[CRm]) where m ← BFA and n ← BF		449
mcrxr	BF	Move XER[0:3] into field CRn, where n←BF. CR[CRn] ← (XER[SO, OV, CA]) (XER[SO, OV, CA]) ← $^{3}0$		450
mfcr	RT	Move from CR to RT, (RT) ← (CR).		451
mfdcr	RT, DCRN	Move from DCR to RT, (RT) ← (DCR(DCRN)).		452
mfmsr	RT	Move from MSR to RT, (RT) ← (MSR).		455

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
mfccr0 **mfccr1** **mfcsrr0** **mfcsrr1** **mfctr** **mfdac1** **mfdac2** **mfdbcr0** **mfdbcr1** **mfdbcr2** **mfdbdr** **mfdbsr** **mfdcdbtrh** **mfdcdbtrl** **mfdear** **mfdec** **mfdnv0** **mfdnv1** **mfdnv2** **mfdnv3** **mfdtv0** **mfdtv1** **mfdtv2** **mfdtv3** **mfdvc1** **mfdvc2** **mfdvlim** **mfesr** **mfiac1** **mfiac2** **mfiac3** **mfiac4** **mficdbdr** **mficdbtrh** **mficdbtrl** **mfinv0** **mfinv1** **mfinv2** **mfinv3** **mfitv0** **mfitv1** **mfitv2** **mfitv3** **mfivlim** **mfivor0** **mfivor1** **mfivor2** **mfivor3** **mfivor4** **mfivor5** **mfivor6** **mfivor7** **mfivor8** **mfivor9** **mfivor10** **mfivor11** **mfivor12** **mfivor13** **mfivor14** **mfivor15** **mfivpr** **mflr** **mfmcsr** **mfmcsrr0** **mfmcsrr1** **mfmmucr**	RT	Move from special purpose register (SPR) SPRN. *Extended mnemonic for* **mfspr RT,SPRN** See *Table 15-1 Special Purpose Registers Sorted by SPR Number* on page 599 for listing of valid SPRN values.		456

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
mfpid **mfpir** **mfpvr** **mfsprg0** **mfsprg1** **mfsprg2** **mfsprg3** **mfsprg4** **mfsprg5** **mfsprg6** **mfsprg7** **mfsrr0** **mfsrr1** **mftbl** **mftbu** **mftcr** **mftsr** **mfusprg0** **mfxer**		Move from special purpose register (SPR) SPRN. *Extended mnemonic for* **mfspr RT,SPRN** See *Table 15-1 Special Purpose Registers Sorted by SPR Number* on page 599 for listing of valid SPRN values.		
mfspr	RT, SPRN	Move from SPR to RT, $(RT) \leftarrow (SPR(SPRN))$.		456
mr	RT, RS	Move register. $(RT) \leftarrow (RS)$ *Extended mnemonic for* **or RT,RS,RS**		487
mr.		*Extended mnemonic for* **or. RT,RS,RS**	CR[CR0]	
msync		Synchronization. All instructions that precede **msync** complete before any instructions that follow **msync** begin. When **msync** completes, all storage accesses initiated prior to **msync** will have completed.		459
mtcr	RS	Move to Condition Register. *Extended mnemonic for* **mtcrf 0xFF,RS**		460
mtcrf	FXM, RS	Move some or all of the contents of RS into CR as specified by FXM field, mask $\leftarrow {}^{4}(FXM_0) \parallel {}^{4}(FXM_1) \parallel \ldots \parallel {}^{4}(FXM_6) \parallel {}^{4}(FXM_7)$. $(CR) \leftarrow ((RS) \wedge mask) \vee (CR) \wedge \neg mask)$.		460
mtdcr	DCRN, RS	Move to DCR from RS, $(DCR(DCRN)) \leftarrow (RS)$.		461
mtmsr	RS	Move to MSR from RS, $(MSR) \leftarrow (RS)$.		464

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
mtccr0 **mtccr1** **mtcsrr0** **mtcsrr1** **mtctr** **mtdac1** **mtdac2** **mtdbcr0** **mtdbcr1** **mtdbcr2** **mtdbdr** **mtdbsr** **mtdear** **mtdec** **mtdecar** **mtdnv0** **mtdnv1** **mtdnv2** **mtdnv3** **mtdtv0** **mtdtv1** **mtdtv2** **mtdtv3** **mtdvc1** **mtdvc2** **mtdvlim** **mtesr** **mtiac1** **mtiac2** **mtiac3** **mtiac4** **mtinv0** **mtinv1** **mtinv2** **mtinv3** **mtitv0** **mtitv1** **mtitv2** **mtitv3** **mtivlim** **mtivor0** **mtivor1** **mtivor2** **mtivor3** **mtivor4** **mtivor5** **mtivor6** **mtivor7** **mtivor8** **mtivor9** **mtivor10** **mtivor11** **mtivor12** **mtivor13** **mtivor14** **mtivor15** **mtivpr** **mtlr** **mtmcsr** **mtmcsrr0** **mtmcsrr1** **mtmmucr** **mtpid**	RS	Move to SPR SPRN. *Extended mnemonic for* **mtspr SPRN,RS** See *Table 15-1 Special Purpose Registers Sorted by SPR Number* on page 599 for listing of valid SPRN values.		465

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
mtsprg0 **mtsprg1** **mtsprg2** **mtsprg3** **mtsprg4** **mtsprg5** **mtsprg6** **mtsprg7** **mtsrr0** **mtsrr1** **mttbl** **mttbu** **mttcr** **mttsr** **mtusprg0** **mtxer**				
mtspr	SPRN, RS	Move to SPR from RS, $(SPR(SPRN)) \leftarrow (RS)$.		465
mulchw	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ (signed)		468
mulchw.			CR[CR0]	
mulchwu	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ (unsigned)		469
mulchwu.			CR[CR0]	
mulhhw	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{0:15} \times (RB_{0:15}$ (signed)		470
mulhhw.			CR[CR0]	
mulhhwu	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ (unsigned)		471
mulhhwu.			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). $(RT) \leftarrow prod_{0:31}$.		472
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). $(RT) \leftarrow prod_{0:31}$.		473
mulhwu.			CR[CR0]	
mullhw	RT, RA, RB	$(RT)_{0:31} \leftarrow (RA)_{16:31} \times (RB_{16:31}$ (signed)		474
mullhw.			CR[CR0]	
mullhwu	RT, RA, RB	$(RT)_{16:31} \leftarrow (RA)_{0:15} \times (RB)_{16:31}$ (unsigned)		475
mullhwu.			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) $(RT) \leftarrow prod_{16:47}$		476
mullw	RT, RA, RB	Multiply (RA) and (RB), signed. Place low-order result in RT. $prod_{0:63} \leftarrow (RA) \times (RB)$ (signed). $(RT) \leftarrow prod_{32:63}$.		477
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.		478
nand.			CR[CR0]	
neg	RT, RA	Negative (two's complement) of RA. $(RT) \leftarrow \neg(RA) + 1$		479
neg.			CR[CR0]	
nego			XER[SO, OV]	
nego.			CR[CR0] XER[SO, OV]	

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
nmacchw	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow -prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		480
nmacchw.			CR[CR0]	
nmacchwo			XER[SO, OV]	
nmacchwo.			CR[CR0] XER[SO, OV]	
nmacchws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow -prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \vee {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		481
nmacchws.			CR[CR0]	
nmacchwso			XER[SO, OV]	
nmacchwso.			CR[CR0] XER[SO, OV]	
nmachhw	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow -prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		482
nmachhw.			CR[CR0]	
nmachhwo			XER[SO, OV]	
nmachhwo.			CR[CR0] XER[SO, OV]	
nmachhws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}$ $temp_{0:32} \leftarrow -prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \vee {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		483
nmachhws.			CR[CR0]	
nmachhwso			XER[SO, OV]	
nmachhwso.			CR[CR0] XER[SO, OV]	
nmaclhw	RT, RA, RB	$prod_{0:31} \leftarrow (RA_{16:31} \times (RB)_{16:31}$ $temp_{0:32} \leftarrow -prod_{0:31} + (RT)$ $(RT) \leftarrow temp_{1:32}$		484
nmaclhw.			CR[CR0]	
nmaclhwo			XER[SO, OV]	
nmaclhwo.			CR[CR0] XER[SO, OV]	
nmaclhws	RT, RA, RB	$prod_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ $temp_{0:32} \leftarrow -prod_{0:31} + (RT)$ if $((prod_0 = RT_0) \wedge (RT_0 \neq temp_1))$ then $(RT) \leftarrow (RT_0 \vee {}^{31}(\neg RT_0))$ else $(RT) \leftarrow temp_{1:32}$		485
nmaclhws.			CR[CR0]	
nmaclhwso			XER[SO, OV]	
nmaclhwso.			CR[CR0] XER[SO, OV]	
nop		Preferred no-op, triggers optimizations based on no-ops. *Extended mnemonic for* **ori 0,0,0**		489
nor	RA, RS, RB	NOR (RS) with (RB). Place result in RA.		486
nor.			CR[CR0]	
not	RA, RS	Complement register. $(RA) \leftarrow \neg(RS)$ *Extended mnemonic for* **nor RA,RS,RS**		486
not.		*Extended mnemonic for* **nor. RA,RS,RS**	CR[CR0]	
or	RA, RS, RB	OR (RS) with (RB). Place result in RA.		487
or.			CR[CR0]	
orc	RA, RS, RB	OR (RS) with $\neg$(RB). Place result in RA.		488
orc.			CR[CR0]	
ori	RA, RS, IM	OR (RS) with $({}^{16}0 \parallel IM)$. Place result in RA.		489
oris	RA, RS, IM	OR (RS) with $(IM \parallel {}^{16}0)$. Place result in RA.		490

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
rfci		Return from critical interrupt (PC) ← (CSRR0). (MSR) ← (CSRR1).		491
rfi		Return from interrupt. (PC) ← (SRR0). (MSR) ← (SRR1).		492
rfmci		Return from machine check interrupt (PC) ← (MCSRR0). (MSR) ← (MCSRR1).		493

rlwimi	RA, RS, SH, MB, ME	Rotate left word immediate, then insert according to mask.
		r ← ROTL((RS), SH)
		m ← MASK(MB, ME)
		(RA) ← (r ∧ m) ∨ ((RA) ∧ ¬m)
				494

rlwimi.			CR[CR0]	

rlwinm	RA, RS, SH, MB, ME	Rotate left word immediate, then AND with mask.
		r ← ROTL((RS), SH)
		m ← MASK(MB, ME)
		(RA) ← (r ∧ m)
				495

rlwinm.			CR[CR0]	

rlwnm	RA, RS, RB, MB, ME	Rotate left word, then AND with mask.
		r ← ROTL((RS), (RB)$_{27:31}$)
		m ← MASK(MB, ME)
		(RA) ← (r ∧ m)
				497

rlwnm.			CR[CR0]	
rotlw	RA, RS, RB	Rotate left. (RA) ← ROTL((RS), (RB)$_{27:31}$) *Extended mnemonic for* **rlwnm RA,RS,RB,0,31**		497
rotlw.		*Extended mnemonic for* **rlwnm. RA,RS,RB,0,31**	CR[CR0]	
rotlwi	RA, RS, n	Rotate left immediate. (RA) ← ROTL((RS), n) *Extended mnemonic for* **rlwinm RA,RS,n,0,31**		495
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**		495
rotrwi.		*Extended mnemonic for* **rlwinm. RA,RS,32−n,0,31**	CR[CR0]	
sc		System call exception is generated. (SRR1) ← (MSR) (SRR0) ← (PC) PC ← EVPR$_{0:15}$ || 0x0C00 (MSR[WE, PR, EE, PE, DR, IR]) ← 0		498

slw	RA, RS, RB	Shift left (RS) by (RB)$_{27:31}$.
		n ← (RB)$_{27:31}$.
		r ← ROTL((RS), n).
		if (RB)$_{26}$ = 0 then m ← MASK(0, 31 – n)
		else m ← $^{32}0$
		(RA) ← r ∧ m.
				499

slw.			CR[CR0]	
slwi	RA, RS, n	Shift left immediate. (n < 32) (RA)$_{0:31-n}$ ← (RS)$_{n:31}$ (RA)$_{32-n:31}$ ← $^{n}0$ *Extended mnemonic for* **rlwinm RA,RS,n,0,31−n**		495
slwi.		*Extended mnemonic for* **rlwinm. RA,RS,n,0,31−n**	CR[CR0]	

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
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] $\leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		500
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] $\leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		501
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 $m \leftarrow {}^{32}0$ $(RA) \leftarrow r \wedge m$.		502
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**		495
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|0) + EXTS(D).		503
stbu	RS, D(RA)	Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + EXTS(D). Update the base address, (RA) $\leftarrow$ EA.		504
stbux	RS, RA, RB	Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + (RB). Update the base address, (RA) $\leftarrow$ EA.		505
stbx	RS, RA, RB	Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + (RB).		506
sth	RS, D(RA)	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + EXTS(D).		507
sthbrx	RS, RA, RB	Store halfword $(RS)_{16:31}$ byte-reversed in memory at EA = (RA|0) + (RB). MS(EA, 2) $\leftarrow (RS)_{24:31} \parallel (RS)_{16:23}$		508
sthu	RS, D(RA)	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + EXTS(D). Update the base address, (RA) $\leftarrow$ EA.		509
sthux	RS, RA, RB	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + (RB). Update the base address, (RA) $\leftarrow$ EA.		510
sthx	RS, RA, RB	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + (RB).		511
stmw	RS, D(RA)	Store consecutive words from RS through GPR(31) in memory starting at EA = (RA|0) + EXTS(D).		512

Table A-1. PPC465 Instruction Syntax Summary (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).		512
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).		514
stw	RS, D(RA)	Store word (RS) in memory at EA = (RA|0) + EXTS(D).		515
stwbrx	RS, RA, RB	Store word (RS) byte-reversed in memory at EA = (RA|0) + (RB). MS(EA, 4) $\leftarrow$ $(RS)_{24:31} \parallel (RS)_{16:23} \parallel$ $(RS)_{8:15} \parallel (RS)_{0:7}$		516

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]) $\leftarrow$ ${}^{2}0 \parallel 0 \parallel XER_{SO}$.
				517

stwu	RS, D(RA)	Store word (RS) in memory at EA = (RA|0) + EXTS(D). Update the base address, (RA) $\leftarrow$ EA.		519
stwux	RS, RA, RB	Store word (RS) in memory at EA = (RA|0) + (RB). Update the base address, (RA) $\leftarrow$ EA.		520
stwx	RS, RA, RB	Store word (RS) in memory at EA = (RA|0) + (RB).		521
sub	RT, RA, RB	Subtract (RB) from (RA). (RT) $\leftarrow$ $\neg$(RB) + (RA) + 1. *Extended mnemonic for* **subf RT,RB,RA**		522
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) $\leftarrow$ $\neg$(RB) + (RA) + 1. Place carry-out in XER[CA]. *Extended mnemonic for* **subfc RT,RB,RA**		523
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]	

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
subf	RT, RA, RB	Subtract (RA) from (RB). (RT) ← ¬(RA) + (RB) + 1.		522
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].		523
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].		524
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].		525
subfme	RT, RA, RB	Subtract (RA) from (–1) with carry-in. (RT) ← ¬(RA) + (–1) + XER[CA]. Place carry-out in XER[CA].		526
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].		527
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**		351
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**		352
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]	353
subis	RT, RA, IM	Subtract (IM || $^{16}0$) from (RA|0). Place result in RT. *Extended mnemonic for* **addis RT,RA,–IM**		354

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
tlbre	RT, RA,WS	tlbentry ← TLB[$(RA)_{26:31}$] if WS = 0 $(RT)_{0:27}$ ← tlbentry[EPN,V,TS,SIZE] $(RT)_{28:31}$ ← $^{4}0$ MMUCR[STID] ← tlbentry[TID] else if WS = 1 $(RT)_{0:21}$ ← tlbentry[RPN] $(RT)_{22:27}$ ← $^{6}0$ $(RT)_{28:31}$ ← tlbentry[ERPN] else if WS = 2 $(RT)_{0:15}$ ← $^{16}0$ $(RT)_{16:24}$ ← tlbentry[U0,U1,U2,U3,W,I,M,G,E] $(RT)_{25}$ ← 0 $(RT)_{26:31}$ ← tlbentry[UX,UW,UR,SX,SW,SR] else (RT), MMUCR[STID] ← undefined		528

tlbsx	RT,RA,RB	Search the TLB for a valid entry that translates the EA.
		EA = (RA|0) + (RB)
		if Rc = 1
		    CR[CR0]$_0$ ← 0
		    CR[CR0]$_1$ ← 0
		    CR[CR0]$_3$ ← XER[SO}
		if Valid TLB entry matching EA and MMUCR[STID,STS] is in the TLB then
		    (RT) ← Index of matching TLB Entry
		    if Rc = 1
		        CR[CR0]$_2$ ← 1
		else
		    (RT) ← Undefined
		    if Rc = 1
		        CR[CR0]$_2$ ← 0
				530

tlbsx.			CR[CR0]	
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 PPC465, **tlbsync** is a no-op.		531

tlbwe	RS, RA,WS	tlbentry ← TLB[$(RA)_{26:31}$]
		if WS = 0
		    tlbentry[EPN,V,TS,SIZE] ← $(RS)_{0:27}$
		    tlbentry[TID] ← MMUCR[STID]
		else if WS = 1
		    tlbentry[RPN] ← $(RS)_{0:21}$
		    tlbentry[ERPN] ← $(RS)_{28:31}$
		else if WS = 2
		    tlbentry[U0,U1,U2,U3,W,I,M,G,E] ← $(RS)_{16:24}$
		    tlbentry[UX,UW,UR,SX,SW,SR] ← $(RS)_{26:31}$
		else tlbentry ← undefined
				532

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
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**		533
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.		533

Table A-1. PPC465 Instruction Syntax Summary (continued)

Mnemonic	Operands	Function	Other Registers Changed	Page
tweqi	RA, IM	Trap if (RA) equal to EXTS(IM). *Extended mnemonic for* **wi 4,RA,IM**		535
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.		535
wrtee	RS	Write value of RS$_{16}$ to MSR[EE].		537
wrteei	E	Write value of E to MSR[EE].		538
xor	RA, RS, RB	XOR (RS) with (RB). Place result in RA.		539
xor.			CR[CR0]	
xori	RA, RS, IM	XOR (RS) with ($^{16}0$ ‖ IM). Place result in RA.		540
xoris	RA, RS, IM	XOR (RS) with (IM ‖ $^{16}0$). Place result in RA.		541

A.3 Allocated Instruction Opcodes

Allocated instructions are provided for purposes that are outside the scope of PowerPC Book-E architecture, and are for implementation-dependent and application-specific use, including use within auxiliary processors.

Table A-2 lists the blocks of opcodes which have been allocated by PowerPC Book-E for these purposes. In the table, the character “u” designates a secondary opcode bit which can be set to any value. In some cases, the decimal value of a secondary opcode is shown in parentheses after the binary value.

Table A-2. Allocated Opcodes

Primary Opcode	Extended Opcodes	PPC465 Usage
0	All instruction encodings (bits 6:31) except 0x0000 0000 (the instruction encoding of 0x0000 0000 is and always will be reserved-illegal)	None
4	All instruction encodings (bits 6:31)	Various (see *Table A-5* on page 645)
19	Secondary opcodes (bits 21:30) = 0buuuuu 0u11u	None
31	Secondary opcodes (bits 21:30) = 0buuuuu 0011u Secondary opcodes (bits 21:30) = 0buuuuu 0u110 Secondary opcode (bits 21:30) = 0b01010 10110 (342) Secondary opcode (bits 21:30) = 0b01011 10110 (374) Secondary opcode (bits 21:30) = 0b11001 10110 (822)	Various (see *Table A-5* on page 645)
59	Secondary opcodes (bits 21:30) = 0buuuuu 0u10u	None
63	Secondary opcodes (bits 21:30) = 0buuuuu 0u10u (except secondary opcode decimal 12, which is the **fsrp** defined instruction)	None

All of the allocated opcodes listed in the table above are available for use by auxiliary processors attached to the PPC465, except for those which have already been implemented within the PPC465 for certain implementation-specific purposes. As indicated in the table above, this is the case for certain secondary opcodes within primary opcodes 4 and 31. These opcodes are identified in *Table A-5* on page 645, along with all of the defined, preserved, and reserved-nop class opcodes which are implemented within the PPC465.

A.4 Preserved Instruction Opcodes

The preserved instruction class is provided to support backward compatibility with the PowerPC Architecture, and/or earlier versions of the PowerPC Book-E architecture. This instruction class includes opcodes which were defined for these previous architectures, but which are no longer defined for PowerPC Book-E.

Table A-3 lists the reserved opcodes designated by PowerPC Book-E. The decimal value of the secondary opcode is shown in parentheses after the binary value.

Table A-3. Preserved Opcodes

Primary Opcode	Extended Opcode	Preserved Mnemonic	PPC465 Usage
31	0b00110 10010 (210)	**mtsr**	
31	0b00111 10010 (242)	**mtsrin**	
31	0b01011 10010 (370)	**tlbia**	
31	0b01001 10010 (306)	**tlbie**	
31	0b01011 10011 (371)	**mftb**	Yes
31	0b10010 10011 (595)	**mfsr**	
31	0b10100 10011 (659)	**mfsrin**	
31	0b01001 10110 (310)	**eciwx**	
31	0b01101 10110 (438)	**ecowx**	

As indicated in the table above, the only preserved opcode which is implemented within the PPC465 is the **mftb** instruction. See “Preserved Instruction Class” on page 55 for more information on PPC465 support for this instruction. All other preserved instructions are treated as reserved by PPC465 and will cause Illegal Instruction exception type Program interrupts if their execution is attempted.

The preserved opcode for **mftb** is included in *Table A-5* on page 645, along with all of the defined, allocated, and reserved-nop class opcodes which are implemented within the PPC465.

A.5 Reserved Instruction Opcodes

This class of instructions consists of all instruction primary opcodes (and associated extended opcodes, if applicable) which do not belong to either the defined, allocated, or preserved instruction classes.

Reserved instructions are available for future versions of PowerPC Book-E architecture. That is, future versions of PowerPC Book-E may define any of these instructions to perform new functions or make them available for implementation-dependent use as allocated instructions. There are two types of reserved instructions: reserved-illegal and reserved-nop.

Table A-4 lists the reserved-nop opcodes designated by PowerPC Book-E. In the table, the character “u” designates a secondary opcode bit which can be set to any value. All other reserved opcodes are in the reserved-illegal class.

Table A-4. Reserved-nop Opcodes

Primary Opcode	Extended Opcode
31	0b10uuu10010

As shown in the table, there are a total of eight (8) secondary opcodes in the reserved-nop class. The PPC465 implements all of the reserved-nop instruction opcodes as true no-ops. These opcodes are included in *Table A-5* on page 645, along with all of the defined, allocated, and preserved class opcodes which are implemented within the PPC465.

A.6 Implemented Instructions Sorted by Opcode

Table A-5 on page 645 lists all of the instructions which have been implemented within the PPC465, sorted by primary and secondary opcode. These include defined, allocated, preserved, and reserved-nop class instructions (see “Instruction Classes” on page 53 for a more detailed description of each of these instruction classes). Opcodes which are *not* implemented in the PPC465 are *not* shown in the table, and consist of the following:

- Defined instructions

  These include the floating-point operations (which may be implemented in an auxiliary processor and executed via the AP interface), as well as the 64-bit operations and the **tlbiva** and **mfapidi** instructions, all of which are handled as reserved-illegal instructions by the PPC465.

- Allocated instructions

  These include all of the allocated opcodes identified in *Table A-2* on page 643 which are not already implemented within the PPC465. If not implemented within an attached auxiliary processor, these instructions will be handled as reserved-illegal by the PPC465.

- Preserved instructions

These include all of the preserved opcodes identified in *Table A-3* on page 643 except for the **mftb** opcode (which *is* implemented and thus included in Table A-5). These instructions will be handled as reserved-illegal by the PPC465.

- Reserved instructions

  These include all of the reserved opcodes as defined by *Appendix A.5* on page 644, except for the reserved-nop opcodes identified in *Table A-4* on page 644. These instructions by definition are all in the reserved-illegal class and will be handled as such by the PPC465.

All PowerPC Book-E 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 612) in bits 0:5. Some instructions also have a secondary opcode field (shown as field XO in Figure A-1 through Figure A-9).

The “Form” indicated in the table refers to the arrangement of valid field combinations within the four-byte instruction. See *Appendix A.1* on page 609, 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 table below.

Table A-5. PPC465 Instructions by Opcode

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
3		D	**twi**	TO, RA, IM	535
4	8	X	**mulhhwu**	RT, RA, RB	471
			mulhhwu.		
4	12 (524)	XO	**machhwu**	RT, RA, RB	443
			machhwu.		
			machhwuo		
			machhwuo.		
4	40	X	**mulhhw**	RT, RA, RB	470
			mulhhw.		
4	44 (556)	XO	**machhw**	RT, RA, RB	440
			machhw.		
			machhwo		
			machhwo.		
4	46 (558)	XO	**nmachhw**	RT, RA, RB	482
			nmachhw.		
			nmachhwo		
			nmachhwo.		
4	76 (588)	XO	**machhwsu**	RT, RA, RB	442
			machhwsu.		
			machhwsuo		
			machhwsuo.		
4	108 (620)	XO	**machhws**	RT, RA, RB	441
			machhws.		
			machhwso		
			machhwso.		
4	110 (622)	XO	**nmachhws**	RT, RA, RB	483
			nmachhws.		
			nmachhwso		
			nmachhwso.		

Table A-5. PPC465 Instructions by Opcode (continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
4	136	X	**mulchwu**	RT, RA, RB	469
			mulchwu.		
4	140 (652)	XO	**macchwu**	RT, RA, RB	439
			macchwu.		
			macchwuo		
			machhwuo.		
4	168	X	**mulchw**	RT, RA, RB	468
			mulchw.		
4	172 (684)	XO	**macchw**	RT, RA, RB	436
			macchw.		
			macchwo		
			macchwo.		
4	174 (686)	XO	**nmacchw**	RT, RA, RB	480
			nmacchw.		
			nmacchwo		
			nmacchwo.		
4	204 (716)	XO	**macchwsu**	RT, RA, RB	438
			macchwsu.		
			macchwsuo		
			macchwsuo.		
4	236 (748)	XO	**macchws**	RT, RA, RB	437
			macchws.		
			macchwso		
			macchwso.		
4	238 (750)	XO	**nmacchws**	RT, RA, RB	481
			nmacchws.		
			nmacchwso		
			nmacchwso.		
4	392	X	**mullhwu**	RT, RA, RB	475
			mullhwu.		
4	396 (908)	XO	**maclhwu**	RT, RA, RB	447
			maclhwu.		
			maclhwuo		
			maclhwuo.		
4	424	X	**mullhw**	RT, RA, RB	474
			mullhw.		
4	428 (940)	XO	**maclhw**	RT, RA, RB	444
			maclhw.		
			maclhwo		
			maclhwo.		
4	430 (942)	XO	**nmaclhw**	RT, RA, RB	484
			nmaclhw.		
			nmaclhwo		
			nmaclhwo.		
4	460 (972)	XO	**maclhwsu**	RT, RA, RB	446
			maclhwsu.		
			maclhwsuo		
			maclhwsuo.		

Table A-5. PPC465 Instructions by Opcode (continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
4	492 (1004)	XO	**maclhws**	RT, RA, RB	445
			maclhws.		
			maclhwso		
			maclhwso.		
4	494 (1006)	XO	**nmaclhws**	RT, RA, RB	485
			nmaclhws.		
			nmaclhwso		
			nmaclhwso.		
7		D	**mulli**	RT, RA, IM	476
8		D	**subfic**	RT, RA, IM	525
10		D	**cmpli**	BF, 0, RA, IM	377
11		D	**cmpi**	BF, 0, RA, IM	375
12		D	**addic**	RT, RA, IM	352
13		D	**addic.**	RT, RA, IM	353
14		D	**addi**	RT, RA, IM	351
15		D	**addis**	RT, RA, IM	354
16		B	**bc**	BO, BI, target	362
			bca		
			bcl		
			bcla		
17		SC	**sc**		498
18		I	**b**	target	361
			ba		
			bl		
			bla		
19	0	XL	**mcrf**	BF, BFA	449
19	16	XL	**bclr**	BO, BI	370
			bclrl		
19	33	XL	**crnor**	BT, BA, BB	383
19	38	XL	**rfmci**		493
19	50	XL	**rfi**		492
19	51	XL	**rfci**		491
19	129	XL	**crandc**	BT, BA, BB	380
19	150	XL	**isync**		410
19	193	XL	**crxor**	BT, BA, BB	386
19	225	XL	**crnand**	BT, BA, BB	382
19	257	XL	**crand**	BT, BA, BB	379
19	289	XL	**creqv**	BT, BA, BB	381
19	417	XL	**crorc**	BT, BA, BB	385
19	449	XL	**cror**	BT, BA, BB	384
19	528	XL	**bcctr**	BO, BI	367
			bcctrl		
20		M	**rlwimi**	RA, RS, SH, MB, ME	494
			rlwimi.		
21		M	**rlwinm**	RA, RS, SH, MB, ME	495
			rlwinm.		
23		M	**rlwnm**	RA, RS, RB, MB, ME	497
			rlwnm.		

Table A-5. PPC465 Instructions by Opcode (continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
24		D	**ori**	RA, RS, IM	489
25		D	**oris**	RA, RS, IM	490
26		D	**xori**	RA, RS, IM	540
27		D	**xoris**	RA, RS, IM	541
28		D	**andi.**	RA, RS, IM	359
29		D	**andis.**	RA, RS, IM	360
31	0	X	**cmp**	BF, 0, RA, RB	374
31	4	X	**tw**	TO, RA, RB	533
31	8 (520)	XO	**subfc** **subfc.** **subfco** **subfco.**	RT, RA, RB	523
31	10 (522)	XO	**addc** **addc.** **addco** **addco.**	RT, RA, RB	349
31	11 (523)	XO	**mulhwu** **mulhwu.**	RT, RA, RB	473
31	15	XO	**isel**	RT, RA, RB	409
31	19	X	**mfcr**	RT	451
31	20	X	**lwarx**	RT, RA, RB	429
31	22	X	**icbt**	RA, RB	404
31	23	X	**lwzx**	RT, RA, RB	435
31	24	X	**slw** **slw.**	RA, RS, RB	499
31	26	X	**cntlzw** **cntlzw.**	RA, RS	378
31	28	X	**and** **and.**	RA, RS, RB	357
31	32	X	**cmpl**	BF, 0, RA, RB	376
31	40 (552)	XO	**subf** **subf.** **subfo** **subfo.**	RT, RA, RB	522
31	54	X	**dcbst**	RA, RB	392
31	55	X	**lwzux**	RT, RA, RB	434
31	60	X	**andc** **andc.**	RA, RS, RB	358
31	75 (587)	XO	**mulhw** **mulhw.**	RT, RA, RB	472
31	78	X	**dlmzb** **dlmzb.**	RA, RS, RB	399
31	83	X	**mfmsr**	RT	455
31	86	X	**dcbf**	RA, RB	388
31	87	X	**lbzx**	RT, RA, RB	414

Table A-5. PPC465 Instructions by Opcode (continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	104 (616)	XO	**neg**	RT, RA	479
			neg.		
			nego		
			nego.		
31	119	X	**lbzux**	RT, RA, RB	413
31	124	X	**nor**	RA, RS, RB	486
			nor.		
31	131	X	**wrtee**	RS	537
31	136 (648)	XO	**subfe**	RT, RA, RB	524
			subfe.		
			subfeo		
			subfeo.		
31	138 (650)	XO	**adde**	RT, RA, RB	350
			adde.		
			addeo		
			addeo.		
31	144	XFX	**mtcrf**	FXM, RS	460
31	146	X	**mtmsr**	RS	464
31	150	X	**stwcx.**	RS, RA, RB	517
31	151	X	**stwx**	RS, RA, RB	521
31	163	X	**wrteei**	E	538
31	183	X	**stwux**	RS, RA, RB	520
31	200 (712)	XO	**subfze**	RT, RA, RB	527
			subfze.		
			subfzeo		
			subfzeo.		
31	202 (714)	XO	**addze**	RT, RA	356
			addze.		
			addzeo		
			addzeo.		
31	215	X	**stbx**	RS, RA, RB	506
31	232 (744)	XO	**subfme**	RT, RA, RB	526
			subfme.		
			subfmeo		
			subfmeo.		
31	234 (746)	XO	**addme**	RT, RA	355
			addme.		
			addmeo		
			addmeo.		
31	235 (747)	XO	**mullw**	RT, RA, RB	477
			mullw.		
			mullwo		
			mullwo.		
31	246	X	**dcbtst**	RA,RB	392
31	247	X	**stbux**	RS, RA, RB	504
31	262	X	**icbt**	RA, RB	404

Table A-5. PPC465 Instructions by Opcode (continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	266 (778)	XO	**add**	RT, RA, RB	348
			add.		
			addo		
			addo.		
31	278	X	**dcbt**	RA, RB	390
31	279	X	**lhzx**	RT, RA, RB	423
31	284	X	**eqv**	RA, RS, RB	400
			eqv.		
31	311	X	**lhzux**	RT, RA, RB	422
31	316	X	**xor**	RA, RS, RB	539
			xor.		
31	323	XFX	**mfdcr**	RT, DCRN	452
31	339	XFX	**mfspr**	RT, SPRN	456
31	343	X	**lhax**	RT, RA, RB	418
31	371	XFX	**mftb**	RT, SPRN	643
31	375	X	**lhaux**	RT, RA, RB	417
31	407	X	**sthx**	RS, RA, RB	511
31	412	X	**orc**	RA, RS, RB	488
			orc.		
31	439	X	**sthux**	RS, RA, RB	510
31	444	X	**or**	RA, RS, RB	487
			or.		
31	451	XFX	**mtdcr**	DCRN, RS	461
31	454	X	**dccci**	RA, RB	394
31	459 (971)	XO	**divwu**	RT, RA, RB	398
			divwu.		
			divwuo		
			divwuo.		
31	467	XFX	**mtspr**	SPRN, RS	465
31	470	X	**dcbi**	RA, RB	389
31	476	X	**nand**	RA, RS, RB	478
			nand.		
31	486	X	**dcread**	RT, RA, RB	395
31	491 (1003)	XO	**divw**	RT, RA, RB	397
			divw.		
			divwo		
			divwo.		
31	512	X	**mcrxr**	BF	450
31	530		**Reserved-nop**		644
31	533	X	**lswx**	RT, RA, RB	427
31	534	X	**lwbrx**	RT, RA, RB	431
31	536	X	**srw**	RA, RS, RB	502
			srw.		
31	562		**Reserved-nop**		644
31	566	X	**tlbsync**		531
31	594		**Reserved-nop**		644
31	597	X	**lswi**	RT, RA, NB	425
31	598	X	**msync**		459

Table A-5. PPC465 Instructions by Opcode (continued)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	626		**Reserved-nop**		644
31	658		**Reserved-nop**		644
31	661	X	**stswx**	RS, RA, RB	514
31	662	X	**stwbrx**	RS, RA, RB	516
31	690		**Reserved-nop**		644
31	722		**Reserved-nop**		644
31	725	X	**stswi**	RS, RA, NB	512
31	754		**Reserved-nop**		644
31	758	X	**dcba**	RA, RB	387
31	790	X	**lhbrx**	RT, RA, RB	419
31	792	X	**sraw**	RA, RS, RB	500
			sraw.		
31	824	X	**srawi**	RA, RS, SH	501
			srawi.		
31	854	X	**mbar**	MO	448
31	914	X	**tlbsx**	RT,RA,RB	530
			tlbsx.		
31	918	X	**sthbrx**	RS, RA, RB	508
31	922	X	**extsh**	RA, RS	402
			extsh.		
31	946	X	**tlbre**	RT, RA,WS	528
31	954	X	**extsb**	RA, RS	401
			extsb.		
31	966	X	**iccci**	RA, RB	406
31	978	X	**tlbwe**	RS, RA,WS	532
31	982	X	**icbi**	RA, RB	403
31	998	X	**icread**	RA, RB	407
31	1014	X	**dcbz**	RA, RB	393
32		D	**lwz**	RT, D(RA)	432
33		D	**lwzu**	RT, D(RA)	433
34		D	**lbz**	RT, D(RA)	411
35		D	**lbzu**	RT, D(RA)	412
36		D	**stw**	RS, D(RA)	515
37		D	**stwu**	RS, D(RA)	519
38		D	**stb**	RS, D(RA)	503
39		D	**stbu**	RS, D(RA)	504
40		D	**lhz**	RT, D(RA)	420
41		D	**lhzu**	RT, D(RA)	421
42		D	**lha**	RT, D(RA)	415
43		D	**lhau**	RT, D(RA)	416
44		D	**sth**	RS, D(RA)	507
45		D	**sthu**	RS, D(RA)	509
46		D	**lmw**	RT, D(RA)	424
47		D	**stmw**	RS, D(RA)	512

Appendix B. Instruction Execution Performance and Code Optimizations

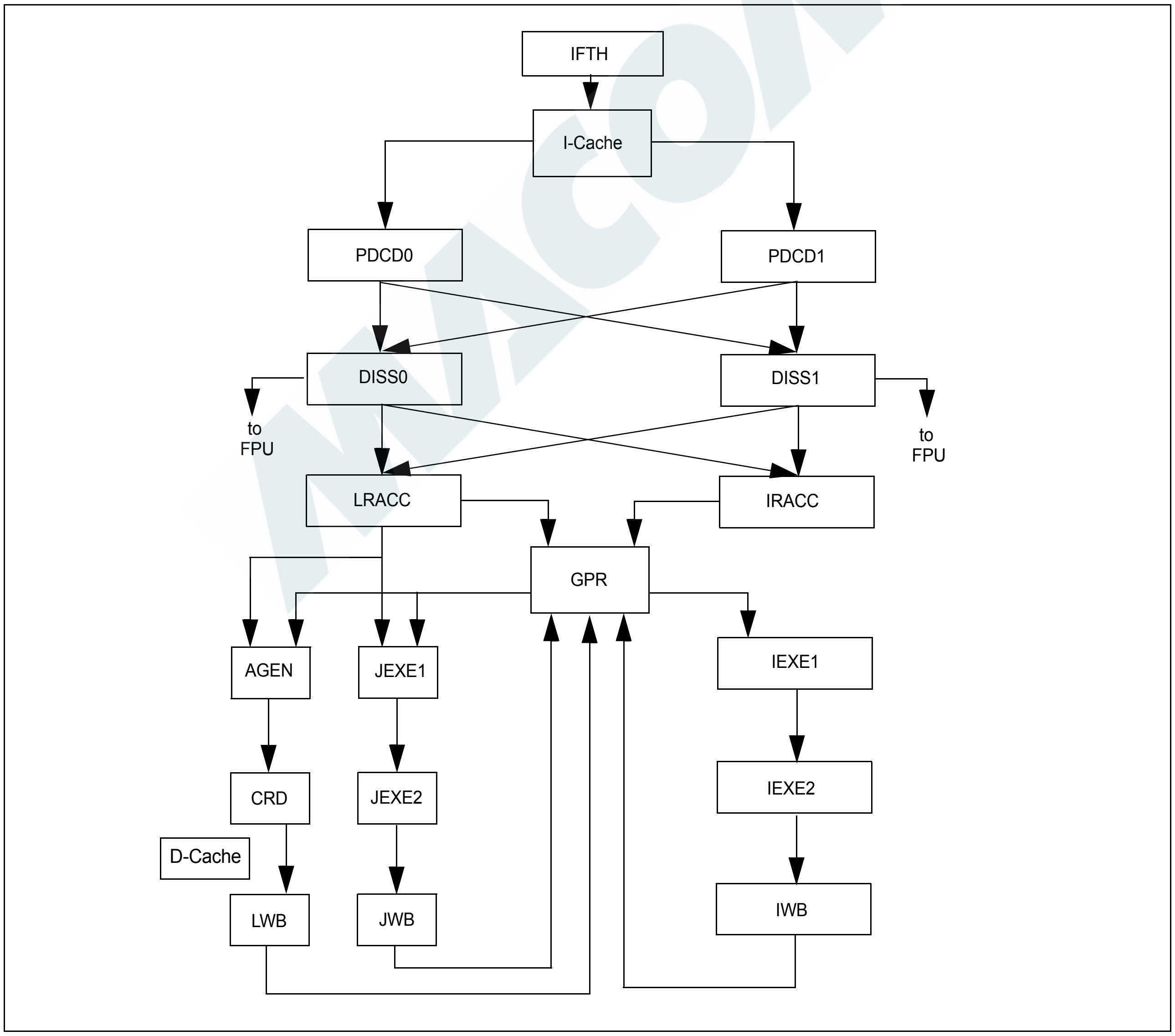
The instruction timing information and code optimization guidelines provided in this appendix can help compiler developers and application programmers produce high-performance code and accurately analyze instruction execution performance. While this appendix does not comprehensively identify every micro-architectural characteristic that could have a potential impact on instruction execution time within the PPC465 core, it does provide a high-level overview of basic instruction operation and pipeline performance. The information provided is sufficient to analyze the performance of code sequences to a high degree of accuracy.

B.1 PPC465 Pipeline Overview

As described in “Overview” on page 27, the PPC465 is a superscalar processor core capable of issuing two instructions per cycle to any two of the core’s three execution pipelines (complex integer, simple integer, and load/store). *Figure B-1* provides an illustration of the 7-stage pipeline of the PPC465 core.

.

Figure B-1. PPC465 Pipeline Structure



As illustrated in *Figure B-1*, the first three pipeline stages are common and provide the fetch (IFTH), decode (PDCD0/PDCD1), and issue (DISS0/DISS1) of up to two instructions per cycle. The next pipeline stage (LRACC/IRACC) provides register access and dispatch of up to two instructions per cycle, with the IRACC stage being dedicated to the complex integer (I-pipe) pipeline, and the LRACC stage being shared between the simple integer (J-pipe) and load/store (L-pipe) pipelines. The final three pipeline stages are replicated for each pipeline, and provide for the execution of the instructions.

In the IFTH pipeline stage, the next two instructions are fetched from the I-cache, unless the instruction fetch address has reached the last instruction in the cache line (32 bytes), in which case only one instruction can be fetched and the first instruction of the next cache line can be fetched in the next cycle. The Branch Target Address Cache (BTAC) and the Branch History Table (BHT) are also accessed in the IFTH stage (See "Branch Instruction Handling" on page 667.for more details on the BTAC, the BHT, and how branch instructions are handled by the pipeline).

In the PDCD pipeline stage, the two instructions which were just fetched are decoded and sent on to the DISS stage.

The DISS pipeline stage is actually a queue with four positions (DISS0 - DISS3), although only the oldest two positions (DISS0 and DISS1) are eligible for issue to the RACC stage, and thus only these positions are illustrated

in the pipeline figure. During the DISS stage, a determination is made as to which pipeline is required by each of the two instructions in DISS0 and DISS1, and the instructions then *issue* to the RACC stage accordingly (the term "issue" refers specifically to the action of moving from either DISS0 or DISS1 to either the LRACC or the IRACC stage). Floating-point and auxiliary processor (AP) instructions which do not require the use of the PPC465 core's execution pipelines issue from DISS0 or DISS1 and enter the appropriate RACC stage within the floating-point or auxiliary processor unit. An instruction in DISS1 (which by design is guaranteed to be *newer* than an instruction in DISS0) may issue *out-of-order* with respect to an instruction in DISS0, in the event that the instruction in DISS0 requires a RACC stage which is not currently available while the RACC stage required by the instruction in DISS1 is available. See "Instruction Issue Operation" on page 657. for more information on how instructions issue to the RACC stage.

The LRACC and IRACC pipeline stages are generally where instructions hold waiting for their source operands to become ready. This is the case for all source operands except the data operand for store instructions (for which the store instruction will hold in the AGEN stage) and the accumulate operand for integer multiplyaccumulate instructions (for which the integer multiply-accumulate instruction will hold in the IEXE1 stage). Once its source operands become ready, the instruction will *dispatch* to the first execution stage of the corresponding pipeline (the term "dispatch" refers specifically to the action of moving from one of the RACC stages to the first execution stage of a pipeline -- either IEXE1, JEXE1, or AGEN). Instructions in the RACC stages may dispatch out-of-order with respect to each other, in the event that the pipeline required by the newer instruction becomes available prior to the pipeline required by the older instruction. Due to the out-of-order issue capability from the DISS stage, there is no guaranteed relative order between the two instructions in the RACC stage. However, the sequence of instructions contained within a given execution pipeline are always guaranteed to be in-order with respect to each other, since they have to pass through the same RACC stage in order to enter the execution pipeline.

The last three stages of the pipeline are unique for each of the three pipelines. In the L-pipe, the AGEN stage is where addresses are generated, and also where store instructions hold waiting for the store data operand to become ready. The CRD stage is where the D-cache is accessed to determine whether the target location exists in the cache, and to obtain load data. The LWB stage is where load hit data is written back to the GPR file, and where store hit data is written back to the D-cache.

In the I-pipe, the IEXE1 stage is the first cycle of instruction execution, and for most operations, the result is available to be forwarded from the end of this stage to subsequent instructions requiring the result as a source operand. Such operations which calculate their result completely in IEXE1 simply proceed down through IEXE2 and into IWB, from which they write their result back to the GPR file. IEXE1 is also where integer multiply-accumulate instructions hold waiting for the accumulate source operand to become ready. Some operations (such as multiply and divide instructions) must continue to execute in IEXE2 and IWB in order to fully calculate their results. Divide instructions actually reside in IWB for 33 cycles as they iteratively calculate their result, at which point they write the result back to the GPR file.

In the J-pipe, all instructions which can utilize the J-pipe are also capable of calculating their final result in the first execution stage JEXE1. These instructions then simply proceed down through JEXE2 and into JWB, from which they write their result back to the GPR file.

The subsequent sections of this appendix provide additional details regarding the performance of various instruction sequences, including the simultaneous issue capabilities and latencies of various instruction pairs.

B.2 Instruction Execution Latency and Penalty

The term *latency* refers to the number of cycles of execution required for a given instruction to produce its “result”, typically the value to be written to the target general-purpose register (GPR) specified as part of the instruction. Most integer instructions (such as the standard arithmetic and logical instructions) have *1-cycle latency*, meaning that their results are “ready” at the end of the first execution stage of the pipeline, and thus available to be *forwarded* (delivered) to any subsequent instruction which may require that result as one of the subsequent instruction’s source operands. One significant exception to this is the *load* instruction category. These instructions have *3-cycle* latency (assuming the target memory location is found in the data cache), with their results becoming available at the end of the third execution stage of the pipeline. Various other special cases exist, and are described in more detail in “Instruction Pair Execution Performance Rules” on page 659.

The term *penalty* refers to the number of processor cycles for which a given instruction cannot proceed down the processor pipeline, due to a *dependency* between itself and an immediately preceding instruction. In other words, if a source operand for a given instruction is the same as the target operand for the preceding instruction, then the given instruction may have to “hold” in the operand access pipeline stage for some number of cycles waiting for its source operand to become “ready”, depending on the latency of the preceding instruction. For example, if a source operand for a given instruction is the same as the target operand of an immediately preceding load instruction, which has three-cycle latency, then there would be a two-cycle *penalty* associated with the given instruction, as it “waits” at the operand access stage of the pipeline for two extra cycles, in order for the load instruction to reach the third execution stage and forward its result to the given instruction. In contrast, if the earlier instruction has one-cycle latency, there would be a zero-cycle penalty (no penalty) associated with the dependent instruction, as the dependent instruction would be able to proceed down the pipeline immediately after the earlier instruction, which would forward its result from the first execution stage of the pipeline.

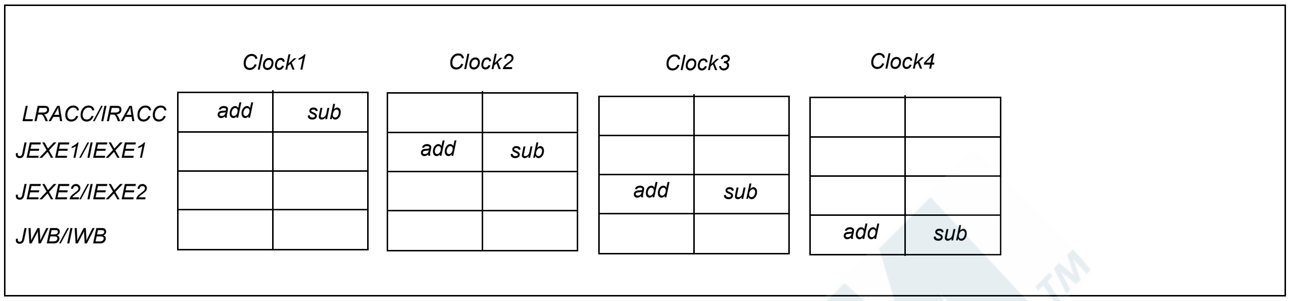
Generally speaking, if a given instruction immediately follows the instruction on which it is dependent, then the number of penalty cycles will be one less than the number of cycles of latency associated with the earlier instruction. There are exceptions to this rule, however, and these are summarized in more detail later in this appendix.

Note that the concept of penalty associated with the instructions in a given code sequence is generally defined to be relative to the performance of a code sequence consisting entirely of non-dependent and/or 1- cycle latency instructions, and executing on a *single-issue* processor micro-architecture. In other words, the “base” performance level is one in which an instruction sequence executes with one instruction completing every cycle. If there are no dependencies between instructions, or every instruction has only 1-cycle latency, then a single-issue micro-architecture will be able to complete one instruction every cycle.

Since the PPC465 is a dual-issue micro-architecture, it is possible for certain instruction sequences to execute at a rate of more than one instruction per cycle, up to a maximum of two instructions per cycle. Thus, the penalty associated with such an instruction stream could be viewed as being “less than zero”, relative to the single-issue micro-architecture. To illustrate with a couple of examples, consider (Figure B-2) a sequence of two instructions, an add followed by a subtract, which have no dependency between them (that is, neither of the source operands of the subtract are the same as the target operand of the add). These two instructions could issue and dispatch at the same time to two different pipelines, giving a performance rate of two instructions per cycle, whereby the “penalty” associated with the subtract instruction could be considered to be “negative one cycle” in that it was able to issue at the same time as the add instruction, rather than one cycle later.

.

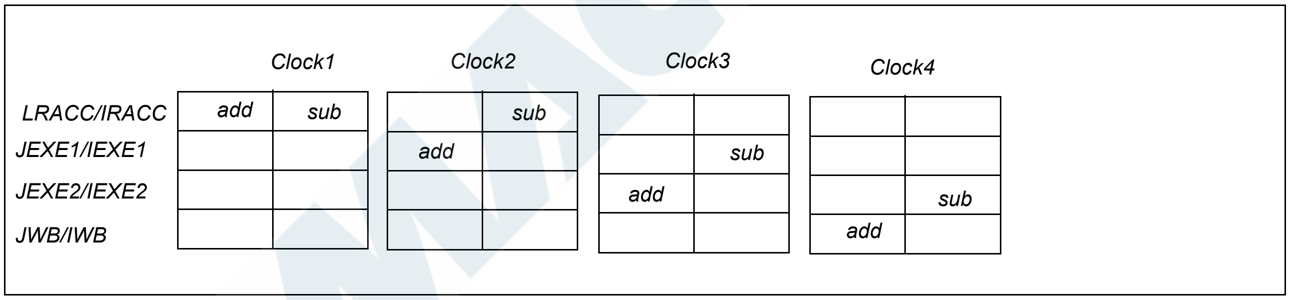
Figure B-2. Add Followed by Subtract with no Dependency between Them



Now consider the same example, but with the subtract instruction dependent on the add (Figure B-3). Now although the two instructions could still issue at the same time to the two RACC stages, they can no longer dispatch at the same time, since the subtract instruction will have to wait in the RACC stage for one extra cycle, while the add instruction dispatches to the first execute cycle and makes its result available as a source operand for the subtract instruction. Since the two instructions must dispatch in two separate but consecutive cycles, the performance is equivalent to a single-issue micro-architecture, and thus exhibits a relative penalty of zero cycles.

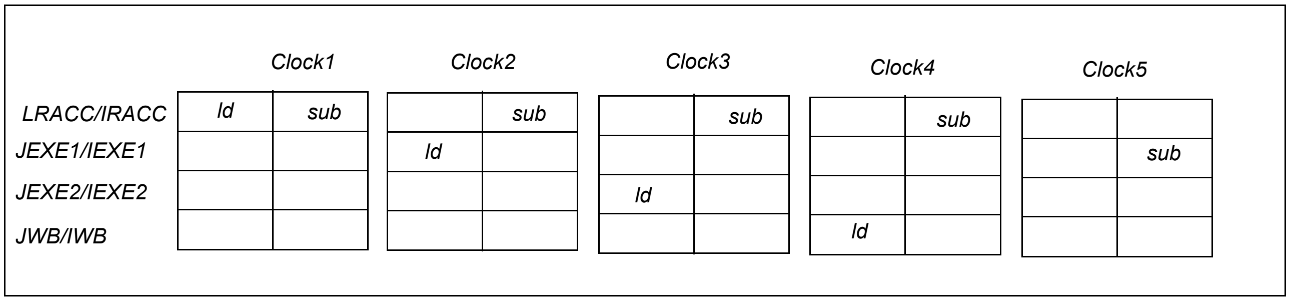
.

Figure B-3. Add Followed by Subtract Which is Dependent on the Add



Lastly, consider a modified sequence where the add instruction is replaced by a load, with the subtract instruction still dependent on the load (Figure B-4). In this case, again the instructions can still issue to the two different RACC stages in the same cycle (with the load necessarily going to LRACC while the subtract goes to IRACC), and the subtract instruction must hold for one cycle in IRACC while the load instruction dispatches to AGEN. But in this case however, the subtract instruction must also hold in IRACC for two additional cycles (the penalty cycles) before it dispatches, three cycles after the load instruction dispatched, in order for the load instruction to have reached the LWB stage and made its result available (assuming a D-cache hit) as a source operand to the subtract instruction. This gives a total of two instructions dispatched in four cycles, or two cycles worse than the baseline non-dependent single-issue performance, and thus a penalty of two cycles.

Figure B-4. Load Followed by Subtract Which is Dependent on the Load



B.3 Instruction Issue Operation

The PPC465 can generally issue two instructions in any given cycle to the RACC stage of the pipeline. The two oldest instructions in the issue queue stage of the pipeline (DISS0 and DISS1) are examined to determine which RACC stage they require (LRACC for the L-pipe or the J-pipe, and IRACC for the I-pipe). If they require (or can use) different RACC stages, then they may issue together. Conversely, if both instructions require the same RACC stage then they must issue in separate cycles, with the older instruction issuing first. Certain instruction types must use LRACC and the L-pipe (such as storage access instructions), certain instructions must use IRACC and the I-pipe (such as branch and multiply instructions), and the rest of the instructions can use either LRACC/J-pipe or IRACC/I-pipe (certain floating-point and AP instruction types do not need either of the PPC465 core's RACC stages as they do not get executed by the core's pipelines). This section summarizes the pipelines which may or must be used by each of the instruction categories, and the rules regarding the simultaneous issuing of instructions.

B.3.1 L-Pipe Instructions

The following categories of instructions must utilize the LRACC dispatch stage and be executed by the Lpipe:

- Storage access, including integer, floating-point, and AP load/store instructions

  Note that "update" forms of load/store instructions, which update the base address register used for the load/store operation, are executed simultaneously by both the L-pipe and the J-pipe. For such instructions, the storage access operation utilizes the L-pipe, while the base address update operation uses the J-pipe.
- Cache management
- Storage synchronization
- Allocated cache management
- Allocated cache debug

B.3.2 I-Pipe Instructions

The following categories of instructions must utilize the IRACC dispatch stage and be executed by the I-pipe:

- • Integer multiply
- • Integer divide
- • Allocated arithmetic (includes multiply-accumulate, negative multiply-accumulate, and multiply halfword)
- • Integer trap
- • Integer count leading zeros

- Allocated logical (**dlmzb**)
- TLB management
- • Branch
- Processor control (includes processor synchronization, register management, system linkage, and condition register logical instructions)
- All instructions which use and/or update the CR (including integer and floating-point instructions)

  Note: the **stwcx.** instruction is both a storage access and a CR-updating instruction, and hence it is executed by *both* the L-pipe and the I-pipe. It simultaneously issues from DISS0 to both LRACC and IRACC.
- All instructions which use and/or update the XER

  Note: The load/store string indexed (**lswx** and **stswx**) instructions are exceptions to this rule, in that they use the XER[TBC] field but are executed by the L-pipe.
- All AP instructions which use and/or update the GPR file (other than AP load/store instructions, which use the GPR file indirectly for address generation)

B.3.3 I-Pipe/J-Pipe Instructions

All other integer instructions may utilize either the IRACC or the LRACC dispatch stage and be executed by either the I-pipe or the J-pipe, respectively. These include:

- • Integer arithmetic instructions which do *not* utilize the CR or the XER, except for multiply and divide instructions (which must be executed by the I-pipe)
- • Integer logical instructions which do *not* update the CR, except for count leading zeros instructions (which must be executed by the I-pipe)
- • Integer rotate instructions which do *not* update the CR
- • Integer shift instructions which do *not* update the CR or the XER

B.3.4 Other Floating-Point and AP Instructions

All floating-point and AP instructions which do not access storage, do not update the CR, and do not use or update the GPR file are executed without using the PPC465 core's execution pipelines. Instead, these instructions issue from DISS0 or DISS1 into the appropriate auxiliary processor pipeline, at which point they no longer exist in the PPC465 core.

B.3.5 Instruction Issue Rules

When determining to which RACC stage(s) the two instructions in DISS0 or DISS1 should be issued, the following rules apply:

- If both instructions require the same RACC stage, then only the DISS0 instruction will issue, with the DISS1 instruction moving to DISS0 and the next instruction moving into DISS1
- If one of the two DISS instructions requires a particular RACC stage, and the other instruction can use either RACC stage (or in the case of a floating-point or AP instruction does not need either RACC stage), then the instruction requiring the particular RACC stage will issue to it, and the other instruction will issue to the other RACC stage (or the floating-point or AP RACC stage, as appropriate).
- If neither instruction requires a particular RACC stage, then DISS0 will by default issue to LRACC (or to the floating-point or AP RACC stage, as appropriate) and DISS1 will by default issue to IRACC (or to the floating-point or AP RACC stage, as appropriate).

- The **stwcx.** instruction requires both the LRACC and the IRACC dispatch stages, as it both accesses storage and updates the CR. Therefore, this instruction cannot issue from DISS1, but must wait until it enters DISS0 and then can simultaneously issue to both LRACC and IRACC.
- Due to various architectural requirements regarding context synchronization and interrupt ordering, the following instructions are treated uniquely with regards to issuing: **isync**, **mtmsr**, **rfi**, **rfci**, **rfmci**, **sc**, **wrtee**, and **wrteei**. These instructions all must wait until they occupy DISS0 before issuing to the IRACC stage. Furthermore, these special instructions each block any subsequent instructions from issuing until the special instruction has completed execution, at which time all subsequent instructions are flushed from the pipelines and re-fetched, at the appropriate address according to the functional definition of the particular instruction. This behavior effectively increases the latency associated with these instructions, and the penalty associated with this extra latency is described in "Processor Control Instruction Operation" on page 665.

B.4 Instruction Pair Execution Performance Rules

In general, most sequences of two non-dependent instructions which do not require the same RACC stage can be simultaneously issued, dispatched, executed, and completed, resulting in a net execution performance of two instructions in one cycle (corresponding to a *penalty* of "negative one" cycle, relative to the single-issue micro-architecture model of two instructions in two cycles; see the definition of "penalty" in ("Instruction Execution Latency and Penalty" on page 655). There are, however, many instruction sequence scenarios where such parallel instruction processing is not possible, due to various factors such as dependencies between the instructions, contention for the same RACC stage and/or execution pipeline, and so on.

This section summarizes the exception cases, identifying those instruction sequences for which simultaneous instruction processing of one form or another is not possible, thereby leading to an increase in the number of cycles required to process the instructions (a decrease in the instructions per cycle metric). These penalty cycles are generally due to one or the other of the two instructions having to hold in a given pipeline stage for more than a cycle, while a dependency or pipeline resource conflict is resolved. For any given sequence of two instructions, if the sequence is not covered by one of the rules listed in this section, then it may be assumed that the two instructions can be processed simultaneously at the rate of two instructions per cycle.

It is important to note that calculating the total number of cycles to execute a sequence of greater than two instructions is not simply a matter of adding up the number of cycles identified in these rules for each of the consecutive instruction pairs in the sequence. Rather, the out-of-order issue, dispatch, execution, and completion capabilities of the PPC465 make it possible in most cases for the cycles associated with any given instruction pair to be overlapped to varying degrees with the cycles associated with another instruction pair. To illustrate with a simple example, consider a sequence of two non-dependent load instructions followed by two non-dependent branch instructions, for which each pair of instructions is subject to exception rule #1 in the preceding list, in that both instructions in each pair require the same execution pipeline, and thus each pair effectively requires two cycles to complete. However, since the first branch instruction can be issued together with the second load instruction, the net throughput for these four instructions would be three cycles, not four. Similarly, the first load instruction *might* issue together with the immediately preceding instruction, and the second branch instruction *might* issue together with the immediately following instruction, potentially maintaining the theoretical maximum execution rate of two instructions per cycle, when considered in the context of the overall instruction sequence.

Also note that when considering the penalty associated with the execution of any given pair of instructions, where the second instruction has some form of dependency on the immediately preceding instruction, this penalty can generally be reduced or avoided altogether by inserting other, non-dependent instructions between the pair. For example, consider the previously mentioned case of a load instruction followed immediately by an instruction dependent on the load result. These two instructions take four cycles to execute, for a penalty of two cycles. As described above, if the load instruction is able to issue together with the instruction preceding it, and the second (dependent) instruction is able to issue together the instruction following it, then the net execution time is four cycles for the *four* instructions, or a zero cycle penalty (net of two cycles per two instructions, the same as the single-issue micro-architecture default). However, if the software sequence were able to be changed such that four more non-dependent instructions were to be inserted between the load and the dependent instruction, then the net execution performance could be increased back to eight instructions for the four cycles (the load and its preceding instruction, the four instructions after the load, the dependent instruction and its successor). This corresponds to the default “negative one” cycle penalty for the two-issue micro-architecture model, or one cycle per two instructions. Generally speaking, compilers should whenever possible attempt to eliminate the penalties associated with the instruction pairings described in the following sections by inserting non-dependent but still useful instructions between the penalty-inducing pair.

Note: Some of the execution performance rules described in the following subsections are related to CR dependencies. When considering these dependencies, there is a special case that should be noted. Specifically, condition register logical instructions specify two bits of the CR as source operands, and specify a third bit of the CR as the target operand. However, since the PPC465 updates the CR on a *field* (as opposed to a *bit*) basis, the field containing the *target* bit operand is actually considered a *source* operand for the sake of any of the CR-related dependency rules described in the following subsections. This is necessary in order to source the “old” value of the three bits of the target field which are *not* being updated by the condition register logical instruction.

Note: Some of the execution performance rules described in the following subsections are related to XER dependencies. Specifically, various “o” form instructions update XER[SO,OV], and various other instructions read XER[SO] and/or XER[OV] as a source operand. These instructions that use XER[SO] and/or XER[OV] as a source operand are: **mfspr** (with the XER specified as the source SPR), **mcrxr**, compare instructions (which copy XER[SO] into CR[CR0]$_3$), and all “record” form instructions (which copy XER[SO] into CR[CR0]$_3$).

B.4.1 Contention for the Same RACC Stage

If the two instructions require the same RACC stage, then they must issue in separate cycles and thus their effective throughput is two cycles for the two instructions. This corresponds to a “penalty” of zero cycles, one cycle worse than the “negative one” cycle penalty for the default, non-contention case where the two instructions can issue together.

B.4.2 General GPR Operand Dependency

If the second instruction has a GPR source operand that is the same as one of the first instruction’s GPR target operands (that is, a GPR read-after-write (RaW) hazard), then in general the second instruction must execute at least one cycle after the first instruction, thereby requiring at least two cycles to execute the two instructions (zero cycles of penalty). Depending on the instruction type of the first instruction (and the pipeline stage at which it finishes calculating its result), the second instruction might have to wait more than one cycle for its source operand to become ready, thereby increasing the penalty for the two instruction sequence even further. The circumstances that result in such additional delay are described in rules listed later in this section.

Two exceptions to this general rule exist. These exceptions are for integer store instructions and the allocated integer multiply-accumulate (MAC) instructions. Specifically, when the second instruction of the sequence is either a store instruction or a MAC instruction, and the source GPR operand of the second instruction which matches the target GPR operand of the first instruction is specifically the store *data* operand (that is, the RS operand shown in the store instruction description) or the MAC *accumulate* operand (that is, the RT operand shown in the MAC instruction description, which is both a source and a target for MAC instructions), respectively, then the two instructions will still generally be able to execute and complete in parallel, such that the effective throughput will still generally be one cycle for the two instructions, which is equivalent to the nondependent case.

This parallel execution is generally possible because the store data operand and the MAC accumulate operand are each accessed one cycle later in the execution pipeline than are other GPR source operands (See “PPC465 Pipeline Overview” on page 653.). As is the case for the general GPR operand dependency rule described in the preceding paragraph, however, and depending on the instruction type of the first instruction, there may be additional delay in the calculation of the first instruction’s result and hence additional penalty in the execution of the two-instruction sequence in such cases, even for the special case of the second instruction being one of these two types. Again, the circumstances under which this additional penalty applies are described in rules listed later in this section. In general though, the GPR dependency-related penalty for the special case of the dependency being for the store data or MAC accumulate operand will be one cycle less than the standard GPR dependency-related penalty.

B.4.3 General CR Operand Dependency

There is no need for a separate general rule for execution penalty associated with CR operand dependencies, corresponding to the general rule for GPR dependencies. This is because all instructions which utilize the CR (either as a source or as a target operand) must issue to the IRACC pipeline stage and be executed by the I-pipe. Therefore, the RACC contention rule described in “Contention for the Same RACC Stage” on page 660 applies to all instruction sequences involving such CR dependencies, leading to a “default” base execution rate of two cycles for the two instructions with such a dependency. For example, the sequence of a compare instruction (which writes a field of the CR as a target) followed by a conditional branch (which reads a bit of the CR as a source) takes two cycles to execute. This is true whether or not the branch instruction is actually conditional upon a CR bit which was updated by the compare instruction (or indeed, whether or not the branch is even conditional). Of course for branch instructions there are other considerations, related to the predicted outcome of the branch and the latency with which the instructions *subsequent* to the branch may be executed, but these considerations are described in more detail in “Branch Instruction Handling” on page 667. Also, as is the case with GPR dependencies, there are other special cases involving instructions which do not calculate their CR results in the first cycle of execution (IEXE1 pipeline stage), and hence which introduce additional cycles of penalty when the subsequent instruction is dependent on those CR results. These special cases are covered in the rules listed later in this section.

B.4.4 Multiply Dependency

Multiply instructions (including the PowerPC-architected 32-bit x 32-bit multiply instructions and the allocated 16-bit x 16-bit multiply-halfword instructions) calculate their results in the IWB pipeline stage (including the GPR result, and the CR result for “record” forms of multiply which update the CR, and the XER result for “o” forms of multiply which update XER[SO,OV]). Therefore, instruction sequences consisting of a multiply followed immediately by an instruction which uses the multiply result (either the GPR, CR, or XER result) as an input operand will take four cycles to complete, which corresponds to a two-cycle penalty, or two cycles *more* than the penalty for the general GPR dependency rule described in “General GPR Operand Dependency” on page 660. Also note that as described in that section, if the dependency involved in the sequence is specifically a store *data* GPR operand, then the penalty is one cycle less, or a total execution time of three cycles, not four. The same is true if the first instruction is a PowerPC-architected 32-bit x 32-bit multiply instruction and the second instruction is a MAC instruction, with the only dependency between the two being the MAC accumulate GPR operand (the total execution time is three cycles).

However, unlike what is described in “General GPR Operand Dependency” on page 660, if the first instruction in the sequence is specifically a multiply-halfword instruction and the second instruction is a MAC instruction using the GPR result of the multiply-halfword instruction as the accumulate operand, then the penalty associated with the sequence is zero cycles, or a total execution time of two cycles for the two instructions, not four.

B.4.5 Multiply-Accumulate (MAC) Dependency

MAC instructions calculate their results in the IWB pipeline stage (including the GPR result, and the CR result for “record” forms of MAC which update the CR, and the XER result for “o” forms of MAC which update XER[SO,OV]). Therefore, instruction sequences consisting of a MAC instruction followed immediately by an instruction which uses the MAC result (either the GPR, CR, or XER result) as an input operand will generally take four cycles to complete, which corresponds to a two-cycle penalty, or two cycles *more* than the penalty for the general GPR dependency rule described in “General GPR Operand Dependency” on page 660. Also note that as described in that section, if the dependency involved in the sequence is specifically a store *data* GPR operand, then the penalty is one cycle less, or a total execution time of three cycles, not four.

However, unlike what is described in “General GPR Operand Dependency” on page 660, if the second instruction in the sequence is another MAC instruction using the same GPR accumulate operand (and there is no XER[SO] dependency between the instructions), then the penalty associated with the sequence is zero cycles, or a total execution time of two cycles for the two instructions, not four. In other words, MAC instructions with the only dependency between them being the GPR accumulate operand can be executed with single-cycle throughput, due to a special forwarding path within the execution pipeline.

Lastly, due to a *write-after-read (WaR)* hazard, instruction sequences consisting of a MAC instruction *preceded* immediately by an instruction which updates the same GPR as the MAC instruction updates will generally take two cycles to complete, which corresponds to a zero-cycle penalty.

B.4.6 Divide Dependency and Pipeline Stall

Divide instructions iteratively calculate their results (including the GPR result, and the CR result for “record” forms of divide which update the CR, and the XER result for “o” forms of divide which update XER[SO,OV]) while remaining in the IWB stage for a total of 33 cycles. Therefore, instruction sequences consisting of a divide followed immediately by an instruction which uses the divide result (either the GPR, CR, or XER result) as an input operand will take 36 cycles to complete, which corresponds to a 34-cycle penalty, or 34 cycles *more* than the penalty for the general GPR dependency rule described in See “General GPR Operand Dependency” on page 660.. Also note that as described in that section, if the dependency involved in the sequence is specifically a store *data* or MAC *accumulate* GPR operand (and there is no XER[SO] dependency between the instructions), then the penalty is one cycle less, or a total execution time of 35 cycles, not 36.

Furthermore, since divide instructions occupy the IWB pipeline stage for a total of 33 cycles (instead of the standard one cycle), they impose an additional 32-cycle penalty on *any* immediately succeeding instruction that also uses the I-pipe, regardless of any dependency that may exist. That is, any instruction sequence involving a divide instruction followed immediately by another instruction that uses the I-pipe will take a minimum of 34 cycles to execute, or a total penalty of 32 cycles. This includes the zero-cycle penalty associated with contention for the same RACC stage (the rule described in See “Contention for the Same RACC Stage” on page 660.), plus the 32 additional cycles of penalty caused by the divide occupying the IWB stage for 33 cycles instead of just one cycle.

On the other hand, instructions subsequent to the divide which utilize the L-pipe or J-pipe and are not dependent on the result of the divide can be executed and completed while the divide is iterating in the IWB pipeline stage.

B.4.7 Move To Condition Register Fields (mtcrf) Instruction Dependency

Due to the nature of the **mtcrf** instruction, which can update any combination of the eight, 4-bit CR fields at once, subsequent instructions which utilize *any* bit or field of the CR as a source must wait for the preceding **mtcrf** instruction to complete before dispatching from the IRACC stage. Therefore, the total execution time for a **mtcrf** instruction followed by an instruction using the CR as a source operand is five cycles, or a penalty of three cycles. Note that this penalty applies whether or not the **mtcrf** instruction is actually updating any of the CR bits or fields being used as source operands by the subsequent instruction.

The following instructions use the CR as a source operand and hence are subject to this three-cycle penalty when they immediately follow a **mtcrf** instruction:
- **bc**, **bclr**, **bcctr** (with BO[0]=0)
- mfcr
- **mcrf**
- Condition register logical instructions (**crand**, **cror**, **crnand**, **crnor**, **crandc**, **crorc**, **crxor**, **creqv**)

B.4.8 Store Word Conditional Indexed (stwcx.) Instruction Dependency

Due to the nature of the **stwcx.** instruction, which conditionally performs a storage access in addition to updating CR[CR0], subsequent instructions which utilize any bit of CR[CR0] as a source operand must wait for the preceding **stwcx.** instruction to complete before dispatching from the IRACC stage. Therefore, the total execution time for a **stwcx.** instruction followed by an instruction using any bit of CR[CR0] as a source operand is five cycles, or a penalty of three cycles.

The following instructions potentially use CR[CR0] (either the whole field or a single bit of the field) as a source operand and if so are subject to this three-cycle penalty when they immediately follow a **stwcx.** instruction:
- **bc**, **bclr**, **bcctr** (with BO[0]=0)
- mfcr
- **mcrf**
- Condition register logical instructions (**crand**, **cror**, **crnand**, **crnor**, **crandc**, **crorc**, **crxor**, **creqv**)

B.4.9 Move From Condition Register (mfcr) Instruction Dependency

Since the **mfcr** instruction reads all eight CR fields at once, and since there can be multiple CR-updating instructions in execution at one time, the **mfcr** instruction must wait until all preceding CR updates have completed before beginning execution. Therefore, any two-instruction sequence involving a CR-updating instruction followed immediately by a **mfcr** instruction will take four cycles to execute, or a penalty of two cycles.

See “CR Updating Instructions” on page 67 for a list of instructions that update the CR. Note that although the **mtcrf** instruction is included in this table, the actual penalty for the sequence of **mtcrf** followed immediately by **mfcr** is three cycles not two, as described in “Move To Condition Register Fields (mtcrf) Instruction Dependency” on page 662 above. Similarly, although the **stwcx.** instruction is included in the table as well, the actual penalty for the sequence of **stwcx.** followed immediately by **mfcr** is three cycles not two, as described in “Store Word Conditional Indexed (stwcx.) Instruction Dependency” on page 663 above.

B.4.10 Move From Special Purpose Register (mfspr) Dependency

The **mfspr** instruction provides its result in the IEXE2 pipeline stage. Therefore, instruction sequences consisting of a **mfspr** followed immediately by an instruction which uses the target GPR of the **mfspr** instruction as an input operand will generally take three cycles to complete, which corresponds to a one-cycle penalty, or one cycle *more* than the penalty for the general GPR dependency rule described in “General GPR Operand Dependency” on page 660. Also note that as described in that section, if the dependency involved in the sequence is specifically a store *data* or MAC *accumulate* operand, then the penalty is one cycle less, or a total execution time of two cycles, not three.

However, this rule applies only to SPRs other than the LR, CTR, or XER. For these three SPRs, the result of **mfspr** is available in the IEXE1 stage and therefore the general GPR dependency rule of “General GPR Operand Dependency” on page 660 applies.

B.4.11 Move From Machine State Register (mfmsr) Dependency

The **mfmsr** instruction provides its result in the IEXE2 pipeline stage. Therefore, the same rule described in “Move From Special Purpose Register (mfspr) Dependency” on page 663 above for **mfspr** applies to the **mfmsr** instruction as well.

B.4.12 Move To Special Purpose Register (mtspr) Dependency and Pipeline Stall

Mtspr instructions occupy the IWB stage for a total of three cycles, and do not perform the write of the target SPR until this third cycle, in order to enforce various architectural rules regarding instruction ordering. Therefore, instruction sequences consisting of a **mtspr** followed immediately by a **mfspr** that references the *same* SPR will take six cycles to complete, which corresponds to a four-cycle penalty. However, this penalty does not apply to the LR, CTR, or XER registers. Special handling within the execution pipeline allows a **mtspr**/**mfspr** sequence that involves one of these three registers to operate in two cycles, thereby incurring only the zero-cycle penalty due to both instructions requiring the I-pipe.

Similarly, when a **mtspr** instruction that specifically targets the MMUCR is followed immediately by a **tlbsx** instruction (which uses some fields of the MMUCR as input operands), the sequence will also take six cycles to complete.

Furthermore, since **mtspr** instructions occupy the IWB pipeline stage for a total of three cycles (instead of the standard one cycle), they impose an additional two-cycle penalty on *any* immediately succeeding instruction that also uses the I-pipe, regardless of any dependency that may exist. That is, any instruction sequence involving a **mtspr** instruction followed immediately by another instruction that uses the I-pipe will take a minimum of four cycles to execute, or a total penalty of two cycles. However, this penalty again does not apply to the LR, CTR, or XER, nor does it apply to the SPRG registers (SPRG0 - SPRG7 and USPRG0). Special handling within the execution pipeline allows a **mtspr** instruction which targets one of these registers to move through the pipeline in the normal fashion, occupying the IWB stage for only one cycle.

Also, instructions subsequent to the **mtspr** which utilize the L-pipe or J-pipe can be executed and completed while the **mtspr** is continuing to occupy the IWB pipeline stage.

B.4.13 TLB Management Instruction Dependencies

In addition to the dependency between a **mtspr** that targets the MMUCR and a subsequent **tlbsx** instruction, for which the penalty is described in "Move To Special Purpose Register (mtspr) Dependency and Pipeline Stall" on page 664, there are four other special case dependencies involving TLB management instructions that lead to execution penalties.

First, the **tlbwe** instruction occupies the IWB pipeline stage for a total of three cycles (similar to the **mtspr** instruction). Therefore, any instruction sequence involving a **tlbwe** instruction followed immediately by another instruction that uses the I-pipe will take a minimum of four cycles to execute, or a total penalty of two cycles. However, instructions subsequent to the **tlbwe** which utilize the L-pipe or J-pipe can be executed and completed while the **tlbwe** is continuing to occupy the IWB pipeline stage.

Second, instruction sequences involving a **tlbre** or **tlbsx** instruction followed immediately by a **mfspr** instruction (that targets any SPR *except* the LR, CTR, or XER) take four cycles to complete, corresponding to a penalty of two cycles. This penalty is due to conflicting usage of pipeline resources between the two instructions.

Third, instruction sequences involving a **tlbwe** instruction followed immediately by a **tlbre** or **tlbsx** instruction also take four cycles to complete, corresponding to a penalty of two cycles. This penalty is due to conflicting usage of the TLB array between the two instructions.

Fourth, instruction sequences involving a **tlbre** or **tlbsx** instruction followed immediately by a load, store, cache management (except **dcba**, which performs no operation on the PPC465), cache debug, or storage synchronization instruction, take four cycles to complete, corresponding to a penalty of two cycles. Similarly, if the first instruction is instead a **tlbwe** then the two-instruction sequence takes six cycles to complete, since the **tlbwe** instruction holds in the IWB pipeline stage for two extra cycles. Conversely, if the order of the two instructions is reversed, with the TLB management instruction coming immediately *after* a load, store, cache management, cache debug, or storage synchronization instruction, the two-instruction sequence takes either three or eight cycles to complete (it takes eight cycles if the first instruction is **icbi**, **icbt**, **iccci**, or **icread**, and three cycles otherwise). These penalties are all due to the potential for conflicting usage of the TLB array or other pipeline resources between the two instructions.

B.4.14 DCR Register Management Instruction Dependency and Pipeline Stall

Since the DCR register management instructions (**mtdcr, mtdcrx, mtdcrux, mfdcr, mfdcrx** and **mfdcrux**) must interact with the asynchronous DCR interface of the PPC465 core, they stall temporarily within the Ipipe. Specifically, these instructions hold in the IEXE1 pipeline stage as they participate in the asynchronous handshake protocol of the DCR interface. The number of cycles for which these instructions remain in the IEXE1 pipeline stage depends upon the speed with which the DCR device responds to the transaction. In general, a DCR register management instruction will occupy the IEXE1 pipeline stage for two cycles, plus the number of CPU clock cycles associated with the DCR interface clock synchronization and the transaction itself. The number of these extra cycles, beyond the base two cycles, depends on the relative clock frequencies of the CPU clock and the DCR interface clock, and on the number of cycles of the DCR transaction itself. Assume a CPU:DCR clock ratio of *R* = *C*/*D*, where *C* and *D* are both positive integers, with *C* > *D*. Also assume a number of DCR interface transaction cycles (of the DCR clock) *M*, where *M* includes all cycles beginning with the one in which the PPC465 core first drives the DCR operation signal active, and ending with the one in which the DCR acknowledge signal is asserted into the PPC465 core by the DCR device. Given these assumptions, the average total number of CPU clock cycles *N* associated with the clock synchronization and the DCR transaction is given by the equation:

$$N = Floor\left(\sum_{(i=1)/(CD)+MR+1}^{C} i \right)$$

Note that as specified above, *N* is an *average* number of cycles; the actual number can vary slightly according to the specific timing of the DCR request relative to the CPU and DCR clock domains. Accordingly, the DCR register management instructions will occupy the IEXE1 pipeline stage for *N*+2 cycles, thereby leading to a penalty of *N*+1 cycles for any immediately subsequent instruction which must utilize the I-pipe. On the other hand, instructions subsequent to a DCR register management instruction which utilize the Lpipe or J-pipe can be executed and completed while the DCR register management instruction is continuing to occupy the IEXE1 pipeline stage.

Furthermore, the **mfdcr** instruction can not forward its GPR result to a subsequent instruction until the IEXE2 pipeline stage. Therefore, instruction sequences consisting of a **mfdcr** followed immediately by an instruction which uses the **mfdcr, mfdcrx or mfdcrux** target register as an input operand will generally take *N*+4 cycles to complete, which corresponds to an *N*+2-cycle penalty. Also note that as described in “General GPR Operand Dependency” on page 660, if the dependency involved in the sequence is specifically a store *data* or MAC *accumulate* operand, then the penalty is one cycle less, or a total execution time of *N*+3 cycles, not *N*+4.

B.4.15 Processor Control Instruction Operation

Various processor control instructions require special handling within the PPC465 core due to the context synchronization requirements of the PowerPC Book-E architecture. These instructions include:

- sc
- mtmsr
- wrtee
- wrteei
- isync
- rfi
- rfci
- rfmci

Each of these instructions requires that the instruction stream be flushed and re-fetched immediately after the instruction's execution, either at the next sequential address (for **mtmsr**, **wrtee**, **wrteei**, and **isync**), or at the System Call interrupt vector location (for **sc**), or at the interrupt return address (for **rfi**, **rfci**, and **rfmci**). Due to the instruction re-fetching requirement and other instruction processing requirements, the minimum execution time for a two-instruction sequence involving one of these instructions as the *first* instruction is as follows:

- eight cycles (for **sc**, **wrteei**, **rfi**, **rfci**, and **rfmci**)
- eleven cycles (for **mtmsr**, **wrtee**, and **isync**)

Furthermore, none of these instructions can be issued together with *any* preceding instruction, which means that the minimum execution time is two cycles (zero-cycle penalty) for any two-instruction sequence in which the *second* instruction is one of these instructions.

B.4.16 Load Instruction Dependency

Load instructions that obtain their data from the data cache generally provide their result in the LWB pipeline stage. Therefore, instruction sequences consisting of a load instruction followed immediately by an instruction which uses the target GPR of the load instruction as an input operand will generally take four cycles to complete, which corresponds to a two-cycle penalty, or one cycle *more* than the penalty for the general GPR dependency rule described in “General GPR Operand Dependency” on page 660. Also note that as described in that section, if the dependency involved in the sequence is specifically a store *data* or MAC *accumulate* operand, then the penalty is one cycle less, or a total execution time of three cycles, not four. The dependency described by this section applies only to the target *data* operand of a load instruction, and not to the target *address* operand of a load *with update* instruction, for which the result is available from the JEXE1 pipeline stage and hence only the general GRP dependency rule applies.

Note that there are many other factors that affect the performance of load and other storage access instructions (such as whether or not their target location is in the data cache). These factors are described in more detail in “Load/Store and Data Cache Effects” on page 672.

B.4.17 Load/Store Ops

A load which depends on the result of a previous store must obtain the store's data as required by the Sequential Execution Model (SEM). To handle this Read-After-Write (RaW) hazard, a read instruction must hold in RACC until all of its operands are available (i.e., the results of all previous writes to the read's operands are known). Note that it isn't necessary that all of these earlier writes have actually been performed in the GPR file, nor that these writes have even been committed by the CS. Rather, it is only required that the results be known and available such that the read operation can proceed into execution with the correct values.

Similarly, a Write-After-Read (WaR) hazard must be handled such that the results of later writes are not erroneously forwarded to earlier reads. Unlike the RaW hazard described above, a write may leave a given RACC stage even if the read is holding in the other RACC stage. Note that this behavior is identical to the LRACC WaR hazard against a MAC in IRACC (described in “Multiply-Accumulate (MAC) Dependency” on page 661) which has a zero-cycle penalty.

The stalls described above are required to handle RaW and WaR hazards based on operand dependency between reads and writes. These stalls do not, however, handle various resource dependencies which may exist lower in the pipe. One special case will be described here, however, more examples can be found in “Load/Store and Data Cache Effects” on page 672. First note that load/store hits can generally flow through the pipe with an overall throughput of one cycle per instruction, assuming a load hit does not immediately follow a store hit to the same address (Note, a typical compiler will never emit such a scenario). In the case that a load hit immediately follows a store hit to the same address, the load will incur a two-cycle penalty (one cycle for the store to write to the RAM array and the other for the load to reaccess the cache). In the special case of a load hit following two store hits where the load matches both stores, the load will incur a three-cycle penalty (two cycles of RAM writes plus one cycle for the load to reaccess the cache).

B.4.18 String/Multiple Ops

To allow for simplifications in the hazard logic implementation for string/multiple operations, all load string multiples are assumed to update all registers (this simplification is necessary because it is not known which registers the string/multiple will access until the final piece is in AGEN). Thus, a load string or multiple which is in LRACC must hold until all older register reads are either past IRACC or are leaving IRACC this cycle (except for MAC, which must be leaving IEXE1). Conversely, a newer register write in IRACC or LRACC must wait until a store string or multiple is finished replicating and the last piece is leaving AGEN before it can proceed. A further simplification was made in the implementation which causes newer load string/multiple operations to hold in LRACC for all older writes to be completely gone from all pipelines. Thus, the associated penalty for any load or store string/multiple depends on the operations which exist in the pipeline at the time the load or store is in LRACC. The pipeline stall scenarios for strings/multiples are summarized below:

- Any GPR Read or Write older than a load string/multiple in DISS0/LRACC/AGEN must hold in IRACC/LRACC
- Any GPR Write older than a store string/multiple in DISS0/LRACC/AGEN must hold in IRACC/LRACC
- A newer load string/multiple in LRACC waits until all older RA/RB reads and all GPR writes are leaving IRACC and all older RS reads are leaving IEXE1
- A newer store string/multiple in LRACC waits until all older GPR writes are gone from the pipeline (DISS0, IRACC, IEXE1, IEXE2, IWB, JEXE1, JEXE2, JWB, AGEN, CRD, LWB, LMQ0-3)

B.4.19 Lwarx/Stwcx. Ops

In general, lwarx/stwcx. operations incur the same penalties as normal loads/stores based on dependencies which exist in the pipeline. Lwarx/stwcx. operations do, however, have a special hold condition which may cause extra penalty cycles. In order to simplify CR updates due to lwarx/stwcx. operations in the I-Pipe, two different lwarx/stwcx. instructions are not allowed to commit simultaneously. To prevent this scenario, all lwarx/stwcx. instructions hold in LRACC if another uncommitted lwarx/stwcx. is in AGEN, CRD, or LWB. Thus, a lwarx/stwcx. instruction may incur extra penalty cycles equal to the number of cycles in which another uncommitted lwarx/stwcx. instruction exists in AGEN, CRD, or LWB while the newer lwarx/stwcx. is in LRACC.

A similar hold condition for stwcx. instructions exists in the I-Pipe since a stwcx. must update the CR. All CR read instructions will hold in IRACC until all CR updates have left IWB and updated the CR. Thus, CR reads may incur extra penalty cycles in IRACC equal to the number of cycles it takes a stwcx. to leave the I-Pipe.

B.4.20 Storage Synchronization Ops

An **msync** instruction travels down the L-Pipe and waits in LWB for all load/store resources to be empty before confirming. The **msync** may then be committed as soon as the next cycle after which it will leave LWB. Thus, any L-Pipe instruction which immediately follows an **msync** will incur a total penalty of two plus the number of cycles it takes for all load/store resources to become empty. Note that in the 465-S implementation, **mbar** is treated exactly as an **msync** instruction. See “Load/Store and Data Cache Effects” on page 672. for more details on the various load/store resources and the potential conflicts/dependencies between them.

B.5 Branch Instruction Handling

The 465-S branch prediction mechanism includes a 4K x 2-bit Branch History Table (BHT) and a 16-entry 8- way associative Branch Target Address Cache (BTAC). The sections below describe the implementation details of the BHT and BTAC along with any branch-related timing dependencies.

B.5.1 BHT Operation

The 4 K x 2-bit BHT is used to maintain dynamic predictions of branches. It is direct-mapped/shared, but indexed using a combination of the branch address and a hash within the 4-bit Global History Register (GHR). The “Gshare” method of indexing is used to reduce the aliasing of branches by keeping a history of branch activity. The history consists of a stream of taken/not taken bits based on any branch in the instruction stream. A predetermined length of history is XORed with the higher order bits of the address index. The address index consists of enough lower order bits of the fetch address to fully index into the chosen BHT size. Note that the same branch could map to several different entries in the BHT, each with its own prediction value.

The most significant bit of each of the 4K 2-bit counters in the BHT is used to decide whether the prediction should agree (1) or disagree (0) with the static prediction as per the PowerPC architecture. Once the branch is determined taken or not taken, the counter counts up if the branch determination agrees with static prediction or down if it disagrees. Once the counter saturates with zero or three, it will only count back away from saturation. When the BHT is read during the IFTH stage (note that it must be read in every IFTH cycle and has an overall throughput of 1 branch per cycle), an entire line of predictions is delivered due to the organization of the RAM. The prediction value is selected and passed on to the PDCD stage. Because the GRAM is slow, the BHT lookup result is unavailable in IFTH. Instead, it is latched and sent to PDCD to be used in the branch prediction logic in that cycle. The BHT write occurs in the cycle after the branch is determined in IEXE1. If the BHT was used in PDCD and the new counter value to be written to the BHT differs from the original counter value, then the write occurs. Note that only branches which are dependent on the CR only are allowed to update the BHT prediction (also note that branches dependent on both the CR and CTR are not allowed in either the BTAC or BHT).

B.5.2 BTAC Operation

The 16-entry 8-way associative BTAC supports the ability to predict branches and their targets before the ICache can deliver the branch instruction data. It looks up the address of the current fetch in the BTAC while the I-Cache is fetching the address. If the BTAC contains the address of the branch, then the information stored in the BTAC arrives sooner than the I-Cache data does. Since two instructions are fetched from the ICache every cycle, two BTAC accesses must be made each cycle as well. The BTAC is split into evan and odd sides such that only even addresses will be stored in the even BTAC and only odd addresses in the odd BTAC.

Since BHT information is not available in the same cycle the BTAC is read, it is impossible to get an incorrectly predicted branch out of the BTAC once it has been written. Thus, BHT-based branches are not allowed into the BTAC. As a result, **bdnz** instructions that are not dependent on the CR and unconditional branches are the only ones allowed into the BTAC (a single bit in the BTAC is used to differentiate between the two).

The BTAC design includes both a read and write port to eliminate the need to arbitrate between read/write operations. The replacement policy of the BTAC is least recently used (LRU). A read and write will never happen simultaneously since a write can only occur if accompanied by an IFTH abort. Since aborted lookups can’t cause LRU updates, the read will not affect the LRU. Note that a read hit only causes an update to the replacement logic if the branch is actually predicted taken.

BTAC information must always be correct so that it is not necessary to verify the results later in the pipe. Because of this constraint, the BTAC must be invalidated on reset and context synchronizing events which could contain code modification or address re-mappings.

B.5.3 Branch Information Queue (BIQ)

The BIQ is a 6-entry FIFO which allows for simultaneous branches to exist in IEXE1, IRACC, and DISS0-3. As a branch instruction submits from PDCD to DISS, the address of the first instruction down the correction path loads into the BIQ along with additional information about the branch (including: BLR, BCTR, BHT index, BHT prediction, predict taken, branch sequential, and a bit indicating that the BHT was used). When a branch reaches IEXE1 it is determined. If the branch is mispredicted the fetcher will go after the correction path supplied by the BIQ during the first cycle the branch is in IEXE1. Regardless of misprediction, the oldest BIQ entry is invalidated during the first cycle a branch is in IEXE1 (By design, the oldest BIQ entry corresponds to the oldest valid branch in the pipeline, up to and including the IEXE1 stage). Also, in the determination cycle, the BIQ send an updated LR value to the E-Unit if the branch links. The decision to update the BHT (and the value of the update) is based on information in the BIQ and branch determination from the E-Unit. If there is any flush the BIQ is cleared.

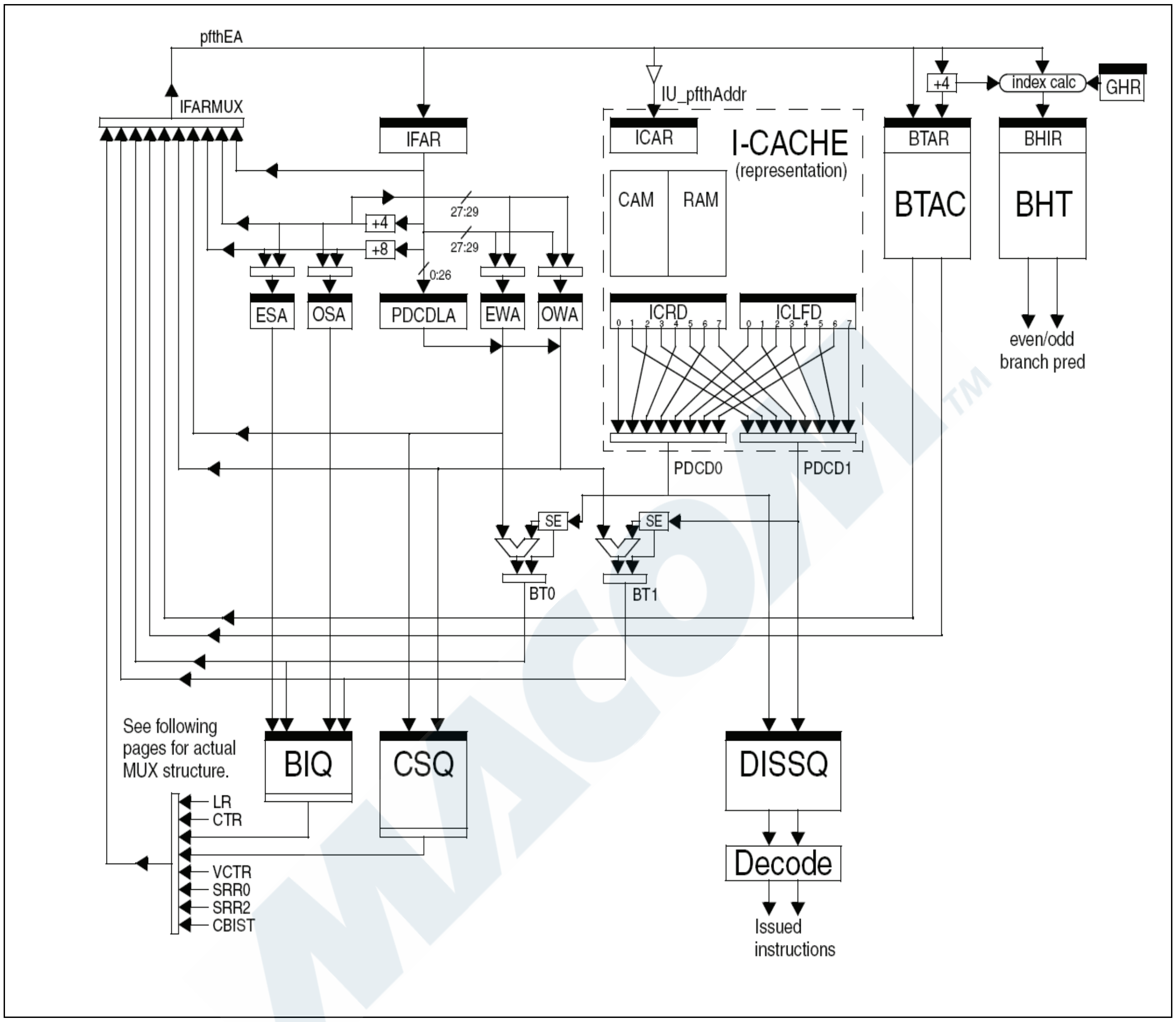
B.6 Fetch/Decode/Issue Effects

Prior to the first ‘true’ pipe stage (IFTH), a pseudo-stage (pre-IFTH) represents the cycle the effective address is being passed to the IFAR/ICAR. In this cycle, a pre-fetch request is made to the I-Cache along with the effective address of the requested instruction. The I-Cache access takes place in the IFTH stage at which point it supplies the PDCD0/1 data to the fetcher. The fetcher implementation is described in detail in the section below followed by a discussion of the decode/issue rules.

B.6.1 IFTH

The Instruction Fetch Address Register (IFAR) represents the start of the IFTH stage of the pipeline. The address in the IFAR is used to read an entire cache line (32 bytes) into the I-Cache Read Data (ICRD) register. The ICRD latches the line at the end of the IFTH cycle. During the IFTH cycle, the same address in the IFAR is used to access the BTAC and BHT. The information from the BTAC and BHT, along with information on the current PDCD instructions, branch corrections from AGEN/IEXE1, exceptions, and context synchronizations, are used during the IFTH cycle to determine what the next IFAR value will be. In the event of an I-Cache miss, the current IFAR value will still flow to the PDCD address register, and the IFAR will be allowed to update to the next value (sequential, or even a branch target if there was a BTAC hit). Likewise, the ICAR will update to the same address as the IFAR. Thus, it is possible that a particular IFTH could miss in the I-Cache, but the IFTH of the subsequent instructions could hit in the I-Cache. In such acase, the address of the first IFTH will be sitting in the PDCD AREG awaiting instructions from the line fill, while the address of the second IFTH will be stuck in the IFAR/ICAR, but the instructions from that second IFTH could already be sitting in the ICRD. Alternatively, the second IFTH could be a miss as well, in which case the C-Unit could pass line fill requests for both lines on to the PLB. Eventually, the instructions for the first IFTH will arrive in the line fill buffer and be provided as the PDCD instructions, clearing the way for the second IFTH address to finally flow from the IFAR into the PDCD AREG. *Figure B-5* provides a block diagram of the Fetcher and Branch Processing Unit.

Figure B-5. Fetcher and Branch Processing Unit



The implementation rules for the selection of the next instruction fetch address are listed below:

- The cycle after reset, the fetcher will go after 0xfffffffc.
- If there is a vector/refetch/return, then fetch the associated address.
- If there is a mispredicted branch, fetch the correction address.
- If there is a pending blr/bctr in **diss** or **iracc** and the **lr**/**ctr** is ready, fetch the **lr**/**ctr**.
- If the older pdcd is being trashed and not submitting, refetch it’s address.
- If the older pdcd is a pending blr/bctr and the lr/ctr is ready, fetch the lr/ctr.
- If the older pdcd is a predicted taken branch, fetch it’s target.
- If the newer pdcd is being trashed and not submitting, refetch it’s address.
- If the newer pdcd is a pending blr/bctr and the lr/ctr is ready, fetch the lr/ctr.
- If the newer pdcd is a predicted taken branch, fetch it’s target.

- If the I-cache is not fetching, refetch ifar.
- If the current fetch (IFAR) matches the ICLFD which hasn't been reload dumped and is getting overwritten now, refetch the IFAR.
- • If the older ifth is a btac hit, predicted taken branch, fetch it's target. (BTAC target or lr/ctr)
- • If only the older ifth word can move to pdcd, refetch the newer ifth word's address.
- • If both ifth words can move to pdcd and the newer ifth word is a btac hit, predicted taken branch, fetch it's target. (BTAC target or lr/ctr)
- • If both ifth words can move to pdcd, fetch the newer ifth word's sequential address.

B.6.2 PDCD

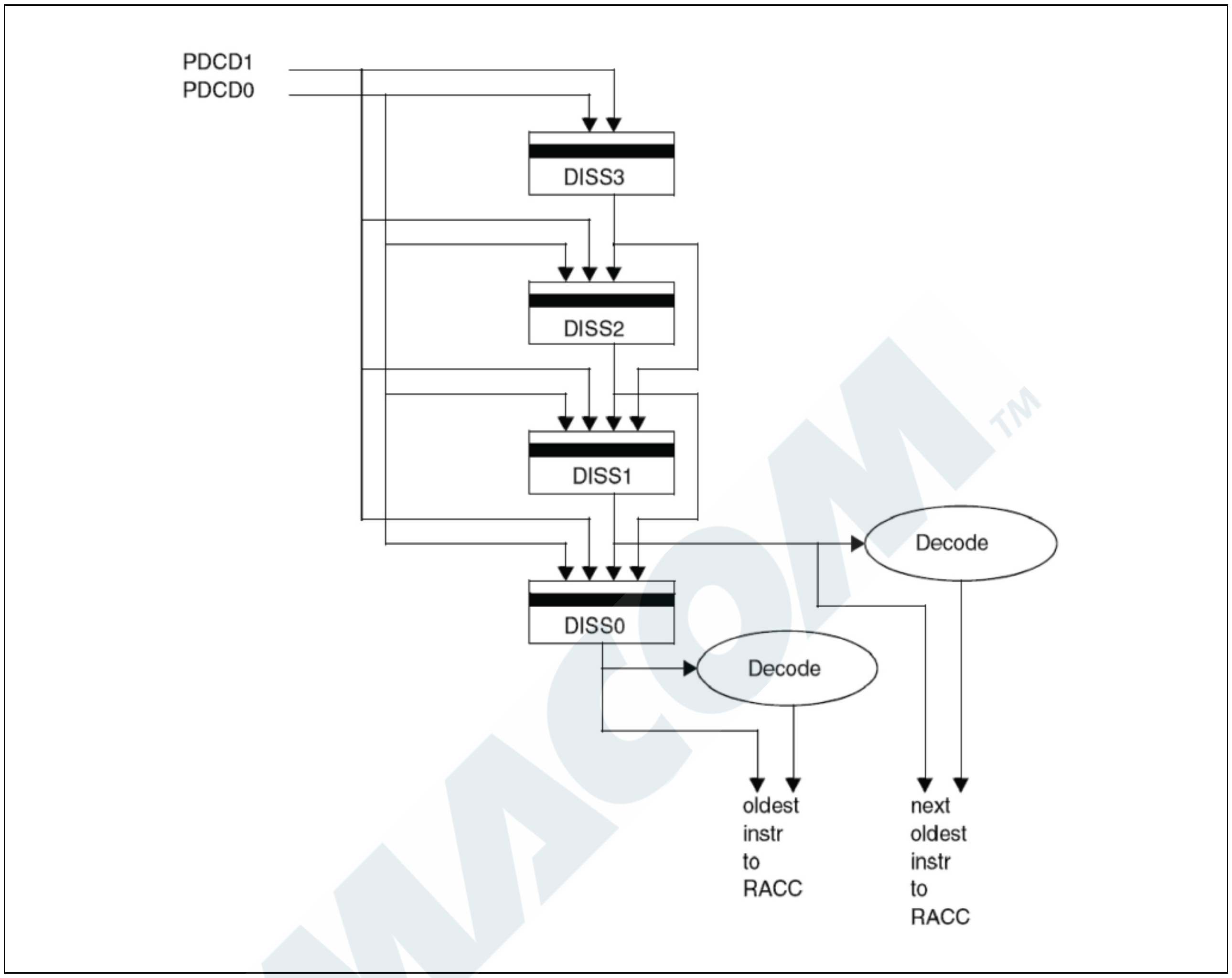
The ICRD register represents the beginning of the PDCD stage. The next two instructions are selected from the line captured in ICRD and delivered as PDCD0/1 instructions. In the case of only one available DISSQ position, the older of the two instructions will submit to DISS. The following rules govern the PDCD behavior:

- Instructions from only one cache line can occupy the PDCD (line buffer or line fill buffer) in any given cycle.
- The fetcher will only fetch adjacent entries of a cache line.
- A branch and its target cannot be submitted simultaneously in PDCD, even if they are adjacent. However, a re-read of the cache is avoided in the case of a branch and its target in the same line by using the same line indicator (SLI) in the BTAC. If this bit is set, then the fetcher knows the branch target is already in the line buffer or coming into the line fill buffer, so a re-read is unnecessary.

B.6.3 DISS

The DISSQ (shown in *Figure B-6*) is a 4 position instruction decode and issue queue numbered 0-3, where 0 is the location of the oldest instruction in the queue. The queue accepts 0, 1, or 2 (submitting) instructions per cycle based on the lowest number of available queue positions (0, 1, 2+) in the core, APU, or FPU (the APU/FPU have DISSQ's that parallel the core). Instructions always enter into the lowest empty OR emptying queue position, behind any instructions already in the queue. In other words, the queue fills from the bottom up, instructions must stay in order, and no bubbles exist in the queue unless all positions are empty. Although some instruction decode is performed during the PDCD stage (and travels through the DISSQ), a significant portion of additional decode is performed in DISSQ positions 0/1. Instructions exit (issue) from queue positions 0/1 based on the decode and pipeline (core/FPU) availability.

Figure B-6. DISSQ



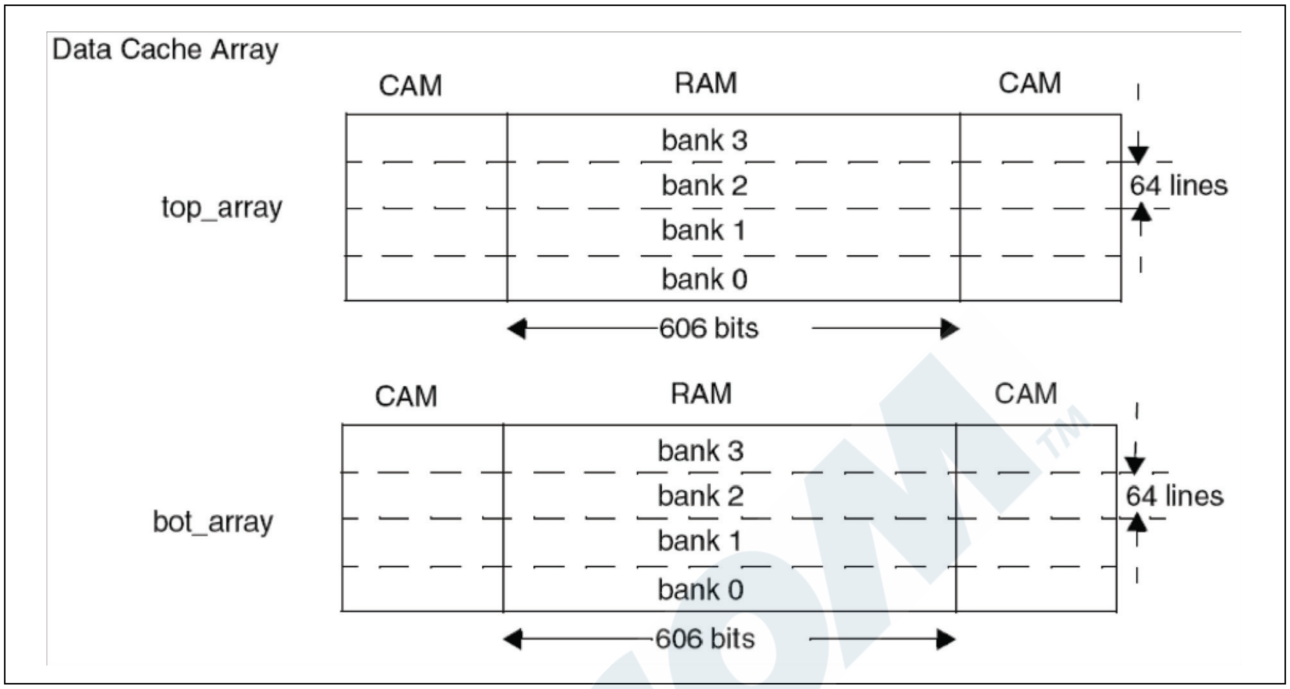
The following list describes the rules for submission into (and movement through) the DISSQ:

- The number of ops (0, 1, or 2) that are able to load (or “submit”) into the DISSQ is based on the number of currently empty queue positions only. The fetcher can submit a maximum of 2 ops per cycle.
- Entries are placed into the queue in program order, based on the lowest position that is empty or is becoming empty. If the queue is empty, then positions DISS0 and DISS1 are used, in order. However, based on the first bullet (above), if the queue is full then NO instructions submit to DISS.
- Instructions shift “down” the DISSQ in program order based on the number on ops issuing (0, 1, or 2) in a given cycle.
- It is never the case that a lower queue position is empty while a higher queue position is full. The queue always shifts such that emptying positions are “filled in”. Also, a new entry is never placed in the queue ahead of a previously existing entry.

B.7 Load/Store and Data Cache Effects

The PPC465 Data Cache is a 32K 64-way set associative non-blocking cache instantiated as two 16 K CAMRAMs (top_array/bot_array) as shown below.*Figure B-7* describes the CAMRAM organization.

Figure B-7. CAMRAM Organization



Each 16 K CAMRAM is organized as: 4 banks X 64 lines/bank X 32 data bytes/line* X 2 sides = 16K. The reason each RAM data width in the diagram says 606 is because it includes parity bits and other memory attributes. *Figure B-1* outlines all memory/cache management instructions which may need to access the Data Cache Array. Note that all of the stages/pseudo-stages mentioned in the table will be discussed in detail in the sections below in addition to a description of the array arbitration rules.

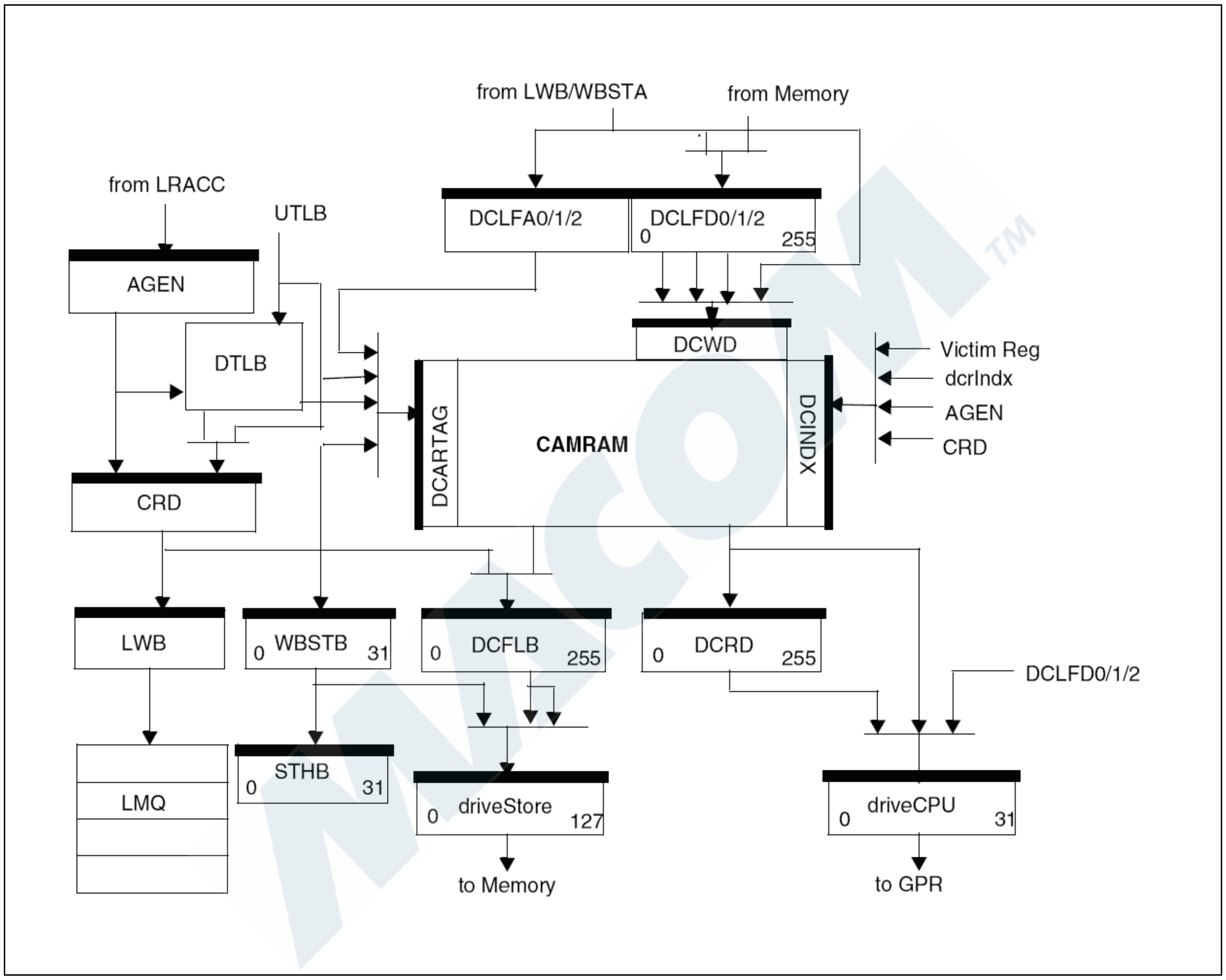
Table B-1. TAG Packet Definition

Instruction Type	CAMRAM Operation
load	CAMRAM search read in AGEN or CRD
dcread	CAMRAM indexed read in AGEN or CRD
dcbi	CAM only search read in AGEN or CRD; if hit, CAM only index write in LWB to clear valid bit
dcbt/dcbtst	CAM only search read in AGEN or CRD
dcbf	CAMRAM search read in AGEN or CRD; if hit, CAM only index write in LWB to clear valid bit
dcbst	CAMRAM search read in AGEN or CRD; if hit, RAM only index write in WBSTB or STHB to clear dirty bits
dcbz	CAM only search read in AGEN or CRD; RAM only index write in WBSTB or STHB to clear data
store	CAM only search read in AGEN or CRD; RAM only index write in WBSTB or STHB to write store data
dccci	CAM flash invalidate in LWB
reload dump	CAMRAM index read followed by CAMRAM index write from DCLFB

B.7.1 Data Cache Resources

Figure B-8 describes the data cache resource. Note that AGEN/CRD/LWB are the only ‘true’ pipestages as depicted in *Figure B-1*. All other resources are used to increase the performance of load/store and cache management instructions.

Figure B-8. CAMRAM Organization



B.7.1.1 AGEN (Address Generation)

AGEN represents the address generation stage in the L-Pipe. When an instruction is dispatched from LRACC to AGEN, two source registers are sent which are added to form the instruction’s effective address. This address is then sent to the DTLB (Data Translation Lookaside Buffer) for translation. If the effective address hits in the DTLB, the resulting translated real address is sent to the DCAR (Data Cache Address Register) in the CAMRAM for a lookup. A DTLB miss will result in a MMU request in CRD which will provide the address translation via the UTLB (Unified Translation Lookaside Buffer). Note that an instruction which misses in the DTLB in AGEN will incur a three-cycle penalty in CRD as it performs a lookup in the UTLB. Another possible hold is a simple ‘propagate’ hold from CRD. In general, an instruction in any pipe stage will incur a penalty equal to the number of cycles the pipe stage below it is holding for any reason.

Replication is another operation performed in the AGEN stage. Loads or stores which are strings, multiples, or misalgned must replicate such that multiple operations will travel through the L-Pipe to satisfy a single instruction. In general, when an access is replicated, the AGEN access is replicated into CRD and flagged as a replicated piece. The AGEN access remains in AGEN, but the address is updated to the next address. In the case of multiples and strings, where replication may occur several times for each access, AGEN continues to replicate itself into CRD while also remaining in AGEN with the address being updated. When no more replication is needed for a given access in AGEN, the 'final' piece moves from AGEN into CRD (note, the final piece is not flagged as replicated).

B.7.1.2 CRD (Cache Read)

The DCAR represents the start of the CRD stage. Note that if an access in AGEN was not able to access the CAMRAM for any reason (e.g., DTLB miss), it must access the CAMRAM from CRD before it can continue through the L-Pipe. See "Cache Arbitration" on page 677. for all reasons why the CAMRAM might not be accessed in AGEN). If an instruction in CRD is unable to access the CAMRAM because another stage is winning arbitration, it must hold in CRD until it wins arbitration. Thus, an instruction in CRD may incur a penalty equal to the number of cycles it takes to win arbitration to the CAMRAM.

A second set of holds in CRD involves cache management operations which want to update the CAMRAM while there are previous instructions which also want to update the CAMRAM. Thus, a dcbf, dcbi, or dcbst will hold in CRD as long as a cacheable line fill buffer exists or a store hit is in WBSTB or STHB. Similarly, a load must hold in CRD if its address matches a store hit in WBSTB or STHB. Note that loads or stores which match the address of a cacheable line fill buffer are allowed to drop into LWB referencing that line fill buffer and thus do not need to hold in CRD.

Finally, an instruction in CRD must hold if a resource it wants to use next cycle will be unavailable. The simplest example of this type of hold is the simple propogate hold from LWB. Unlike AGEN, however, a CRD instruction may drop into either WBSTB (store, dcbz, dcbst) or DCFLB (dcbf or dcbst). Thus, in addition to the LWB propogate hold, a store, dcbz, or dcbst will hold in CRD if WBSTB will be unavailable next cycle. Similarly, a dcbf or dcbst will hold in CRD if DCFLB will be unavailable next cycle. Thus, an instruction in CRD may incur extra penalty cycles as long as LWB, WBSTB, or DCFLB are unavailable.

B.7.1.3 LWB/WBSTB (Load Write Back/WriteBack Store Buffer)

The DCRD (Data Cache Read Data) register represents the start of the LWB stage. For loads, the DCRD feeds a mux which selects load data to get into the driveCPU register which is then sent to the target GPR. For stores, WBSTB will be occupied in addition to LWB. From WBSTB, the store data may go into a DCLFD, directly into the cache, or into drive store and out to main memory based on the attributes associated with the store (for example, writethru versus writeback). Note that subsequent loads may enter and leave LWB while a store occupies WBSTB as long as the loads are not to the same address as the store. In that case, WBSTB must hold as it will lose CAMRAM arbitration to the series of loads (See "Cache Arbitration" on page 677. for more details).

In general, LWB must hold and wait for commitment from the Central Scrutinizer. Note that instructions which access the CAMRAM from LWB (dcbf, dcbi, dccci) will always win arbitration in the cycle after commitment and thus will never incur penalty cycles due to losing arbitration to another resource. However, other holds will cause instructions in LWB to incur extra penalty cycles as well. If LWB is a load hit, it must wait until driveCPU is available to forward its data. If LWB is a load miss or a non-cacheable load, it will hold if all four LMQ entries are full and not leaving. An instruction which wants to allocate a line fill buffer must hold in LWB if all three line fill buffers are valid and not retiring and do not match the address of the instruction. Similarly, a store in WBSTB will hold if driveStore is valid and not leaving and the addresses do not match (note that store gathering is the exception to this rule). Thus, LWB will incur penalty cycles as it waits for commitment and may then incur extra penalty cycles as it waits for the next resource to become available. Another set of holds exists in LWB to prevent read after write hazards. Basically, if an instruction in LWB wants to allocate a line fill buffer (this assumes it does not match a cacheable line fill buffer which already exists), it will hold if it matches the address of a previous store (in WBSTB or driveStore) or flush (in the DCFLB). This prevents read after write hazards because a read request to main memory can only be made from a line fill buffer. By holding in LWB and preventing allocation of a line fill buffer, a read request can never

be presented before a previous write request to the same address. Thus, LWB may incur penalty cycles equal to the number of cycles it must wait until it may safely allocate a line fill buffer without the potential of a read after write hazard.

B.7.1.4 STHB (Store Hit Buffer)

Store hits which are committed and unable to win CAMRAM arbitration in WBSTB will drop into STHB and store to the CAMRAM from there. The STHB will simply wait until it can win arbitration to the CAMRAM and then write its data into the cache. Note that store hits in general may write data into the RAM while the CAM is being accessed for a different instruction (dcbi, dcbt/tst, dcbz, store).

B.7.1.5 DCLFB (Data Cache Line Fill Buffer)

The following instructions may allocate a DCLFB:

- load (cacheable or inhibited)
- cacheable stores with allocate
- cacheable, non-writeThru dcbz (faulty if inhibited or writeThru)
- cacheable dcbt

A non-cacheable DCLFB may only satisfy the load that caused its allocation. Once it satisfies that load, it is no longer valid. A cacheable DCLFB, on the other hand, may satisfy many cacheable loads and accept the data from many cacheable stores. Cacheable loads can be serviced by a DCLFB if a cacheable DCLFB exists that matches the line address of the load and the bytes that are being requested by the load are valid in the DCLFB (note that data may be valid in the DCLFB because of a previous store). If the DCLFB matches the line address, but the bytes are not valid yet, (note, the entire line will be requested for cacheable instructions) the access will wait in the LMQ or in LWB if LWB is holding.

Because cacheable stores are allowed to update the DCLFBs, data modified by a store must not be used to satisfy an prior load in the LMQ. All stores in CRD that conflict with loads in LWB or in the LMQ will hold in CRD until the conflict is resolved -- usually by the satisfying of the prior load requests. Since this conflict resolution exists, any committed store in LWB may update the DCLFB without regard to data already in the DCLFB. However, data from memory destined for the DCLFB must not overwrite any existing valid bytes of data.

B.7.1.6 LMQ (Load Miss Queue)

The LMQ is a 4-entry in-order queue of load misses (or non-cacheable loads). The purpose is to allow delayed loads, which are not cache hits, to leave the LWB stage of the L-Pipe and wait until the load data is ready to be transferred. Because the loads are able to leave LWB, the L-Pipe is able to continue serving new instruction instead of waiting for the load instruction to transfer its data. If the LMQ is full, a load miss or noncacheable load must hold in LWB until an LMQ entry becomes available (note, this excludes the case of a load miss which matches the line address of a valid cacheable DCLFB).

B.7.1.7 DCFLB (Data Cache Flush Buffer)

The DCFLB contains the address and data of a line that was read from the cache via a reload dump (castout), dcbf, or dcbst. The DCFLB will represent a valid request which must get into driveStore if it is a dirty dcbf/dcbst hit or a valid dirty castout. The DCFLB must wait for commitment for dcbf/dcbst's before dropping into driveStore (castouts are always considered to be committed). Besides that, the only other penalty cycles which may be incurred in DCFLB are when driveStore is unavailable and not leaving (note that DCFLB can never gather with driveStore so there is no exception to this rule).

B.7.1.8 driveStore

The driveStore buffer contains stores or line flushes that are destined for main memory. Stores are always sourced from WBSTB and must be either non-cacheable, write-through, or cache misses without allocate that do not match the line address of a valid cacheable DCLFB. Flush data from dirty dcbf/dcbst hits or valid dirty castouts are always sourced from DCFLB. If both WBSTB and DCFLB contain valid operations destined for driveStore, DCFLB will always win arbitration. Note that the DCFLB contains 256-bits of data and thus may require two transfers to move all of its data into driveStore (a full cache line will be stored if a double-word in each half of the line is dirty).

The driveStore buffer will be allowed to clear when all of its data has been accepted by the memory synchronization interface and its address has been acknowledged by the memory bus. Thus, an operation in driveStore may incur penalty cycles due to either the synchronization logic being backed up or by the memory bus being too busy to service the request. Note that the PPC465 store synchronization logic contains two 128-bit data buffers and may thus hold either two store requests, two half-line castouts, or a single full-line castout before causing driveStore to hold.

B.7.2 Cache Arbitration

This section describes the rules for each resource gaining access to the CAMRAM. See "PPC465 Pipeline Overview" on page 653. for a list of all possible CAMRAM accesses and the stages from which the requests will be made.

LWB and STHB have the highest priority to the CAMRAM and will always win arbitration when they are "ready". LWB will always win in the cycle after it is committed and STHB will always win in the cycle it becomes valid. LWB and STHB will never both try to win arbitration at the same time because any access which wants to access the CAMRAM from LWB (dcbf, dcbi, dccci) can never make it to LWB while there are stores in WBSTB/STHA which want to write to the cache.

The next highest priority in general is WBSTB with the exception of loads in AGEN. Recall that stores which want to write to the RAM will hold in WBSTB to allow subsequent loads to access the cache and move through the L-Pipe as long as none of the loads have the same address as the store. The DCFLB has the next highest priority after WBSTB followed by CRD and then AGEN.

B.8 Interrupt Effects

In the PPC465 design, the process of "taking on interrupt" spans two cycles called irptCycleA and irptCycleB. This is necessitated by the need to allow any outstanding, committed SPR updates to actually update the SPR before any subsequent interrupt vector is taken. Due to timing constraints, a committed mtspr does not actually update the target SPR until the cycle after it is committed, such that the SPR has the new value in the second cycle after the mtspr is committed. For example, if an illegal op exception occurs in the samecycle that a mtevpr is committed, then in the next cycle the associated SXR[PGM] bit has been set, indicating a program interrupt, and the EVPR is getting updated due to the mtevpr. If the fetcher is allowed to request the interrupt vector in this same cycle, then the old EVPR value will be used in calculating the PFTH effective address since the EVPR does not actually contain the new data until the next cycle. In breaking the interrupt into two cycles, the PFTH request for the interrupt vector is delayed until the second cycle (irptCycleB), at which time any SPR updates are complete (and, in the case of the example above, the correct PFTH effective address is calculated).

During the first interrupt cycle (irpt cycleA) the interrupt logic is performing a flush operation (global flush is signaled to all pipeline stages and pseudo-stages), and the program counter is steered into (C)SRR0 if appropriate. Note that refetch operations (either due to a committed C-sync op or due to stop request) are similarly handled in two cycles, with the PFTH request occurring in cycleB. The fact that mtmsr is handled as a C-sync operation that requires a refetch dictates that the refetch take two cycles for the same reasons that interrupts take two cycles, since the actual refetch must be delayed until after the new MSR value is set, such that the new context is used. Because interrupts span 2 cycles, all new interrupt requests are blocked during irptCycleB. That is, during irptCycleB the processor is doing what was decided during the previous irptCycleA, and new interrupt requests (such as an async) are ignored during interrupts in irptCycleB. One cycle later, the irptCycleA/B sequence is repeated if another interrupt request exists and it was not disabled by the updating of the MSR during the first irptCycleB. This same type of blocking (during cycle B) occurs for refetch/stop as well.

Ultimately, only instructions which have not been committed will incur penalty cycles due to an interrupt. When global flush is signaled, all uncommitted resources will be flushed from the pipe and will pay penalty cycles equal to the number of cycles it takes for a given instruction to be refetched and then get back into the resource from which it was flushed.

Index

Revision Log

Revision Date	Version	Description
04/01/2010	1.0	Initial creation of 465 processor UM.
05/07/2010	1.1	Updated the description of the ERPN in Table 4-2. Added the following note to Section 6.1.11 L2 Cache Fixed Address Mode (FAM): “Note: TLB memory pages that map to a FAR must have the FAR storage attribute bit set.” Added FAR storage attribute description to Table 7-1. Added FAR storage attribute to Figure 7-5.
05/20/2011	1.2	Updated section 6.1.11.3. Doc Issue 12533: Updated section 6.1.12.11. Updated section 7.1.1. Updated section 7.6.3. Doc Issue 11305: Updated the following floating point instructions in section 13.6: lfd, lfdu, lfdux, lfdx, stfd, stfdu, stfdux, stfdx, stfsux.
12/02/2011	1.3	Doc Issue 12875: Added new section 5.7.12 “L2 Cache Example Initialization Code” on page 121. Doc Issue 18355: Added new section “Processor Complex ” on page 37.
10/01/2012	1.31	Removed "Preliminary". Updated sections *Who Should Use This Book* on page 23 and *Overview* on page 27. Doc Issue 29225: Added additional Programming Notes to instructions **“lwarx” on page 429** and to **“stwcx.” on page 517**.