Updates to e500mc Core Reference Manual, Rev. 3, as of 2015-07-09

This section provides updates to the *e500mc Core Reference Manual*, Rev 3. We are providing known corrections, but do not guarantee that the list is exhaustive. For convenience, the section number and page number of the item in the reference manual are provided.

Note: This PDF file contains updates embedded as inline sticky notes; use the provided links and scroll to the inline location. Future versions of the document will incorporate the sticky notes into the source text of the document. Freescale recommends viewing this file with Adobe Acrobat Reader.

Section, Page No.	Changes
9.11.6, 9-78	In Table 9-47, “Performance Monitor Event Selection,” changed LRU to LSU in row Com:24.

e500mc Core Reference Manual

Supports
e500mc (all revisions)

e500mcRM
Rev. 3
03/2013

freescale™

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: e500mcRM
Rev. 3, 03/2013

Contents

Contents

Contents

Contents

Chapter 3
Instruction Model

Contents

Contents

Contents

Chapter 5
Core Caches and Memory Subsystem

Contents

Contents

Contents

Contents

Contents

Contents

Chapter 10
Execution Timing

Contents

Contents

Figures

Figures

Figures

Tables

Tables

Tables

Tables

Tables

Tables

About This Book

This document includes the register model, instruction model, MMU, memory subsystem, debug and performance monitor facilities of the e5500. The primary objective of this core reference manual is to describe the functionality of the e500mc embedded microprocessor core for software and hardware developers. This book is intended as a companion to the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors: A Programmer's Reference Manual for Freescale Embedded Processors* and *Power ISA™ Version 2.06*. Features defined by the Power instruction set architecture (ISA) are described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*; this manual focuses on features that are specific to the e500mc microprocessor.

Locate errata or updates for this document at http://www.freescale.com. Information in this book is subject to change without notice, as described in the disclaimers on the title page of this book. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation.

Audience

It is assumed that the reader understands operating systems, microprocessor system design, and the basic principles of RISC processing and has access to the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* and *Power ISA™ Version 2.06*.

Suggested Reading

This section lists additional reading that provides background for the information in this manual as well as general information about the architecture.

General Information

The following documentation is available from Power.org from their website *http://www.power.org*:

- *Power ISA™ Version 2.06B*, July 2010

The following documentation, published by Morgan-Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA, provides useful information about computer architecture in general:

- *Computer Architecture: A Quantitative Approach*, Third Edition, by John L. Hennessy and David A. Patterson
- *Computer Organization and Design: The Hardware/Software Interface*, Second Edition, David A. Patterson and John L. Hennessy

Related Documentation

Freescale documentation is available from the sources listed on the back cover of this manual. The document order numbers are included in parentheses for ease in ordering:

- *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors: A Programmer's Reference Manual for Freescale Embedded Processors*

This book provides a higher-level view of the programming model as it is defined by the Power ISA and Freescale implementation standards.

- Integrated device reference manuals
  These books provide details about individual implementations of embedded devices that incorporate embedded cores, such as the e500mc.
- Addenda/errata to reference manuals
  Because some processors have follow-on parts, an addendum is provided that describes the additional features and functionality changes. These addenda are intended for use with the corresponding user's manuals.
- Hardware specifications
  Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations.
- Technical summaries
  Each device has a technical summary that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation's user's manual.
- Application notes
  These short documents address specific design issues useful to programmers and engineers working with Freescale processors.

Additional literature is published as new processors become available. For a current list of documentation, refer to http://www.freescale.com.

Conventions

This document uses the following notational conventions:

cleared/set	When a bit takes the value zero, it is said to be cleared; when it takes a value of one, it is said to be set.
mnemonics	Instruction mnemonics are shown in lowercase bold.
italics	Italics indicate variable command parameters, for example, **bcctr***x*. Book titles in text are set in italics Internal signals are set in italics, for example, $\overline{qual\ BG}$
0x0	Prefix to denote hexadecimal number
0b0	Prefix to denote binary number
rA, **r**B, **r**S	Instruction syntax used to identify a source GPR
rD	Instruction syntax used to identify a destination GPR
frA, **fr**B, **fr**C	Instruction syntax used to identify a source FPR
frD	Instruction syntax used to identify a destination FPR
REG[FIELD]	Abbreviations for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[PR] refers to the privilege mode bit in the machine state register.
x:y	A bit range from bit x to bit y inclusive.

x-y	A bit range from bit x to bit y inclusive.
x	In some contexts, such as signal encodings, an unitalicized x indicates a don't care.
x	An italicized *x* indicates an alphanumeric variable.
n	An italicized *n* indicates an numeric variable.
¬	NOT logical operator
&	AND logical operator
|	OR logical operator
—	Indicates reserved bits or bit fields in a register. Although these bits can be written to as ones or zeros, they are always read as zeros.

Terminology Conventions

Table i lists certain terms used in this manual that differ from the architecture terminology conventions.

Table i. Terminology Conventions

Architecture Specification	This Manual
Extended mnemonics	Simplified mnemonics
Privileged mode (or privileged state)	Supervisor level
Hypervisor mode (or hypervisor state)	Hypervisor level
Problem mode (or problem state)	User level
Out-of-order memory accesses	Speculative memory accesses
Storage (locations)	Memory
Storage (the act of)	Access

Chapter 1
e500mc Overview

This chapter provides a general overview of the e500mc microprocessor core. It includes the following:

- An overview of architecture features as implemented on the e500mc and a summary of the core feature set
- A summary of the instruction pipeline and flow
- An overview of the programming model
- An overview of interrupts and exceptions handling
- A description of the memory management architecture
- High-level details of the e500mc core memory and coherency model
- A brief description of the CoreNet interface
- A list of differences between different versions of the e500 core from e500v2

The e500mc core provides features that an integrated device may not implement or may implement in a more specific way. Differences are summarized in the documentation for the integrated device.

1.1 Overview

The e500mc core is a low-power implementation of the resources for embedded processors defined by the Power ISA™. The core is a 32-bit implementation and implements 32 32-bit general-purpose registers; however it supports accesses to 36-bit physical addresses. The block diagram in Figure 1-1 shows how the e500mc functional units operate independently and in parallel. Note that this conceptual diagram does not attempt to show how these features are implemented physically.

The e500mc is a superscalar processor that can issue two instructions and complete two instructions per clock cycle. Instructions complete in order, but can execute out of order. Execution results are available to subsequent instructions through the rename buffers, but those results are recorded into architected registers in program order, maintaining a precise exception model.

The processor core integrates two simple instruction units (SFX0, SFX1), a multiple-cycle instruction unit (MU), a branch unit (BU), a floating-point unit (FPU), and a load/store unit (LSU).

The LSU supports 32-bit integer and 64-bit floating-point operands.

The ability to execute six instructions in parallel and the use of simple instructions with short execution times yield high efficiency and throughput. Most integer instructions execute in one clock cycle.

The core includes on-chip first-level instruction and data memory management units (MMUs) and an on-chip second-level unified MMU.

- The first-level MMUs for both instruction and data translation are each composed of two subarrays: an 8-entry fully-associative array of translation look-aside buffer (TLB) entries for

variable-sized pages, and a 64-entry 4-way set-associative array of TLB entries for fixed sized pages that provide virtual to physical memory address translation for variable-sized pages and demand-paged fixed pages respectively. These arrays are maintained entirely by the hardware with a true least-recently-used (LRU) algorithm, and are a cache of the second level MMU.

- The second-level MMU contains a 64-entry, fully-associative unified (instruction and data) TLB array that provides support for variable-sized pages. It also contains a 512-entry, 4-way set-associative unified TLB for 4-Kbyte page size support. These second-level TLBs are maintained completely by the software.

The e500mc includes independent on-chip, 32-Kbyte, eight-way set-associative, physically addressed L1 caches for instructions and data and a unified 128-KB, eight-way set-associative, physically addressed, backside L2 cache.

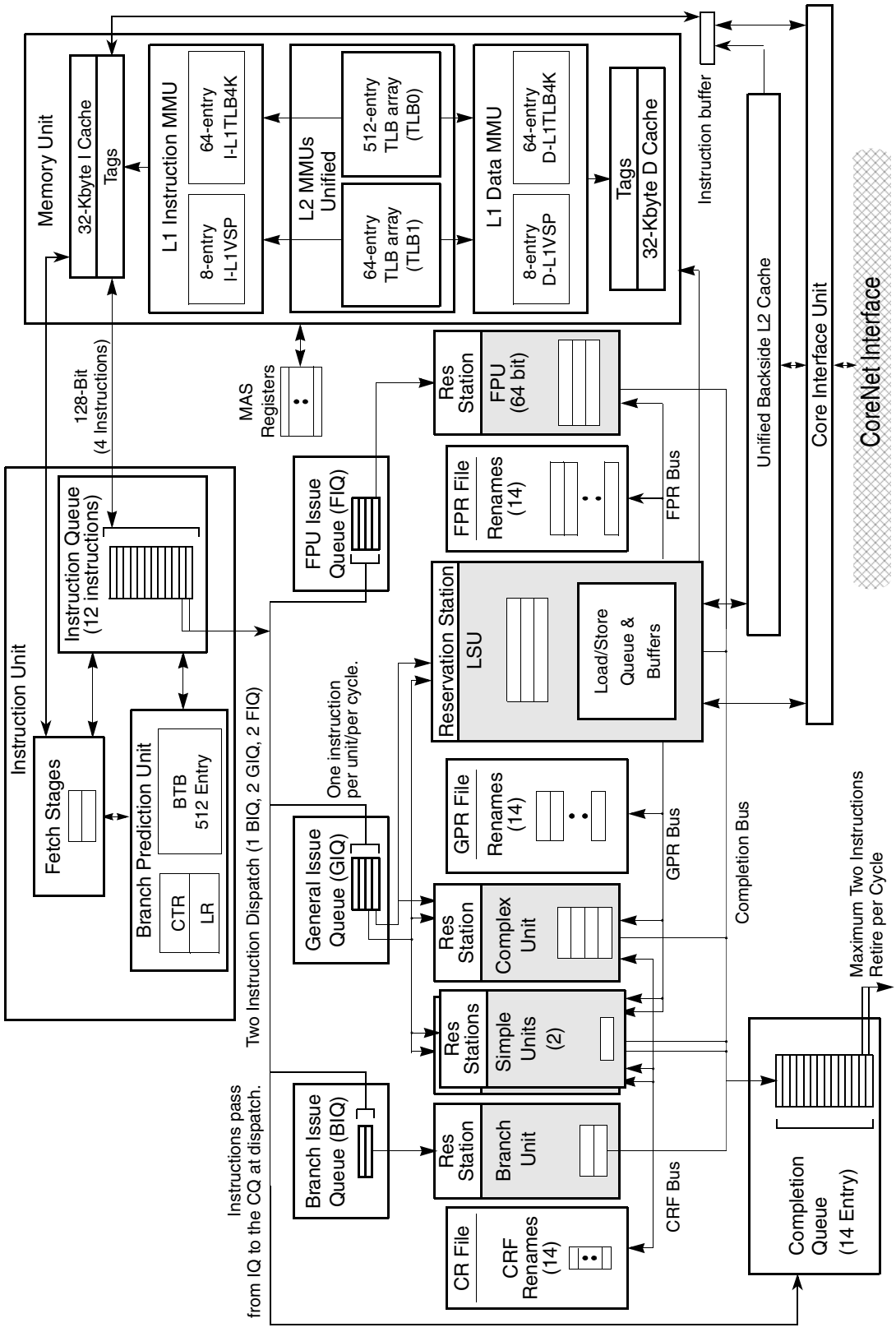


Figure 1-1. e500mc Block Diagram

Cache lines on the e500mc are 16 words (64 bytes) wide. The core allows cache-line-based user-mode locks on cache contents. This provides embedded applications with the capability for locking interrupt routines or other important (time-sensitive) instruction sequences into the instruction cache. It also allows data to be locked into the data cache, which supports deterministic execution time.

The e500mc shown as "Core" in Figure 1-2, is designed to be implemented in multicore integrated devices, and many of the features are defined to support multicore implementations, in particular to partition the cores in such a way that multiple operating systems can be run with the integrated device.

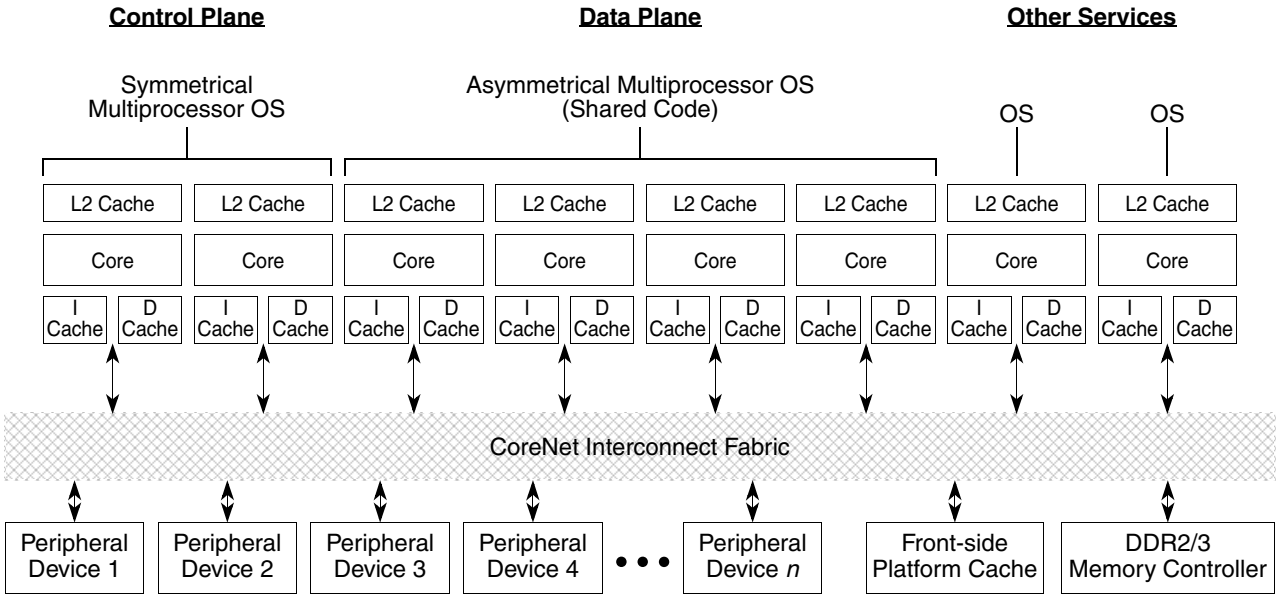


Figure 1-2. Example Partitioning Scenario of a Multicore Integrated Device

The architecture defines the resources required to allow orderly and secure interactions between the cores, memory, peripheral devices, and virtual machines. These include a hypervisor and guest supervisor privilege levels, that determine whether certain activities, such as memory accesses and management, cache management, and interrupt handling, are to be carried on at a system-wide level (hypervisor level) or by the operating system within a partition (guest supervisor level).

In particular, e500mc implements the following categories as defined by *PowerISA 2.06*:

- Base
- Embedded
- Alternate Time Base
- Cache Specification
- Decorated Storage
- Embedded.Enhanced Debug
- Embedded.External PID
- Embedded.Hypervisor
- Embedded.Little-Endian

- Embedded.Performance Monitor
- Embedded.Processor Control
- Embedded.Cache Locking
- External Proxy
- Floating Point and Floating Point.Record
- Memory Coherence
- Store Conditional Page Mobility
- Wait

The above categories define instructions, registers, and processor behavior associated with a given category. For a more complete and canonical definition of the e500mc register and instruction set, see Chapter 2, "Register Model," and Chapter 3, "Instruction Model," respectively.

The CoreNet interface provides the primary on-chip interface between the cores and the rest of the SoC. CoreNet is a tag-based interface fabric that provides interconnections among the cores, peripheral devices, and system memory in a multicore implementation.

1.2 Feature Summary

Key features of the e500mc are summarized as follows:

- Implements 32-bit architecture, with 36-bit physical addressing
- 32 32-bit General Purpose Registers (GPR)
- 32 64-bit Floating Point Registers (FPR)
- FPR-based floating-point, binary compatible with e300 and e600 cores
- Multicore architecture support
  - Hypervisor programming model (category Embedded.Hypervisor in PowerISA 2.06). Many resources are hypervisor privileged, allowing the hypervisor to completely partition the system. Performance sensitive resources used by the guest supervisor are manipulated directly by hardware while less performance sensitive resources require hypervisor software to intervene to provide partitioning and isolation.
  - A set of topology independent interprocessor doorbell interrupts implemented through the Message Send and Message Clear instructions (category Embedded.Processor Control in PowerISA 2.06).
- CoreNet interface fabric. Provides interconnections among the cores, peripheral devices, and system memory in a multicore implementation.
- Decorated Storage, when used with specifically enabled SoC devices, allowing for high performance atomic "fire and forget" operations on memory locations performed directly by the targeted device
- Cache features
  - Separate 32-Kbyte, eight-way set associative level 1 (L1) instruction and data caches. The L1 cache contains sixty-four 8-way sets of 16 words. See Section 5.4, "L1 Cache Structure."
  - Enhanced Error Detection and Correction

- Parity checking on L1 tags and data
- One-bit-per-word instruction parity checking
- One-bit-per-byte L1 data parity checking
- L2 cache ECC single-bit correction, double-bit detection. L2 cache tags parity detection only.
- Write shadow mode. This allows all modified data in the L1 cache to be written through to the L2 cache. This also allows for automatic invalidations to correct cache tag and data errors since modified data is written through and protected with ECC at another level of the memory hierarchy.
- Non-maskable Interrupt for soft-reset type capability

— Two-cycle L1 cache array access, three cycle load-to-use latency

— Pseudo-LRU (PLRU) replacement algorithm

— Cache coherency. CoreNet supports four-state cache coherency: modified-exclusive, exclusive, shared, and invalid (MESI).

- Provides snooping
- Modified and exclusive data intervention allowing cache contents can be shared without requiring memory to be updated

— Integrated 128-KB, eight-way set-associative backside L2 cache

- Supports data- and instruction-only cache operation
  The L2 cache can be programmed as instruction, data, or unified, which control whether a cache line is allocated on a instruction or data miss (or both or neither).
  The L2 Cache supports way partitioning effectively assigning a certain number of ways to allocate on instruction misses and a certain number of ways to allocate on data misses.

— 64-byte (16-word) cache-line, coherency-granule size

— Cache locking. Allows instructions and data to be locked into their respective caches on a cache block basis. Locking is performed by a set of touch and lock set instructions. This functionality can be separately enabled for user mode or supervisor mode.

- Interrupt model. Supports base, critical, debug, and machine-check interrupt levels with separate interrupt resources (save/restore registers and interrupt return instructions).

— Interrupts have an implicit priority by how their enable bits are masked when an interrupt is taken. Unless software enables or disables the appropriate interrupt enables while in the interrupt handler, the priority (from highest to lowest) is:

- Machine Check
- Debug
- Critical
- Base class

— Standard embedded category interrupts

- Less than 10-cycle interrupt latency
- Interrupt vectors formed by concatenation of interrupt vector prefix register (IVPR) and interrupt vector offset register (IVOR*n*)

  - Exception syndrome register (ESR)
- — Extended multicore interrupt model to support hypervisor and guest mode privilege levels
  - System Call instruction to generate a system call or a hypervisor-level system call (hypercall) interrupt. Executing **sc** or **sc 0** generates a system call and **sc 1** generates hypercall interrupt.
  - Doorbell interrupts defined to allow one processor to signal an interrupt to another core (doorbell, doorbell critical, guest doorbell, guest doorbell critical, and guest doorbell machine check)
  - Ability to configure whether certain interrupts are delivered directly to the guest supervisor state, or by default to the hypervisor state
  - Embedded Hypervisor Privilege interrupt to capture guest supervisor attempts to access hypervisor resources
  - TLBs can be programmed to always force a data storage interrupt (DSI) to generate a virtualization fault to hypervisor state
- — External interrupt proxy provides automatic hardware acknowledgement of external interrupts signaled by the programmable interrupt controller (PIC) on the integrated device, (replacing the “read IACK” step) increasing responsiveness to external interrupts from peripheral devices and reducing interrupt latency. See Section 4.9.6.3, “External Proxy.”

- Memory management unit (MMU)
  - — 32-bit effective address to 36-bit physical address translation
  - — Virtual address fields in TLB entries
    - GS field indicates whether the access is guest or supervisor privilege level.
    - AS field indicates one of the two address spaces (from IS or DS in the MSR)
    - LPID field identifies the logical partition with which the memory access is associated
    - PID field identifies the process ID with which the memory access is associated
  - — Extended PID translation mechanism provides an alternative set of load, store, and cache operations for efficiently transferring large blocks of memory or performing cache operations across disjunct address spaces, such as an operating system copying a buffer into a non-privileged area.
  - — TLB entries for variable- (4 Kbytes to 4 Gbytes) and fixed-size (4-Kbyte) pages
  - — Data L1 MMU
    - 8-entry, fully-associative TLB array for variable-sized pages
    - 64-entry, 4-way set-associative TLB for 4-Kbyte pages
  - — Instruction L1 MMU
    - 8-entry, fully-associative TLB array for variable-sized pages
    - 64-entry, 4-way set-associative TLB for 4-Kbyte pages
  - — Unified L2 MMU
    - 64-entry, fully-associative TLB array (TLB1) for variable-sized pages
    - A 512-entry, 4-way set-associative unified (for instruction and data accesses) TLB array (TLB0) supports only 4-Kbyte pages

  - Software reload for TLBs
  - Real memory support for as much as 64 Gbytes (2^{36})
  - Support for big-endian and true little-endian memory on a per-page basis
- Performance monitor
  - Provides the ability to monitor and count dozens of predefined events, such as processor clocks, misses in the instruction cache or data cache, types of instructions decoded, or mispredicted branches.
  - Can be configured to trigger either a performance monitor interrupt or an event to the Nexus facility when configured conditions are met.
  - Performance Monitor Registers (PMRs) are used to configure and track performance monitor operations. These registers are accessed with the Move to PMR and Move from PMR instructions (**mtpmr** and **mfpmr**).
- Power management
  - Low-power design
  - Power-saving modes: core-halted and core-stopped
  - Asynchronous bus
  - Dynamic power management
  - **wait** instruction, places the core in a Doze-like, low-power mode until an interrupt occurs
- Testability
  - Nexus debug support
  - Debug Notify Halt (**dnh**) instruction. When enabled through an external debug facility, executing **dnh** causes the core to enter the halted state. Normal instruction execution is frozen, instructions are not fetched, interrupts are not taken, and the core does not execute instructions from the architectural instruction stream, and control of the processor is managed by the external debug facility.

1.3 Instruction Flow

The e500mc core is a pipelined, superscalar processor with parallel execution units that allow instructions to execute out of order but record their results in order. Pipelining breaks instruction processing into discrete stages, so multiple instructions in an instruction sequence can occupy the successive stages: as an instruction completes one stage, it passes to the next, leaving the previous stage available to a subsequent instruction. So, even though it may take multiple cycles for an instruction to pass through all of the pipeline stages, once a pipeline is full, instruction throughput is much shorter than the latency.

A superscalar processor is one that, in a single cycle, issues multiple independent instructions into separate execution units, allowing parallel execution. The core has six execution units, one each for branch (BU), load/store (LSU), floating-point (FPU), and complex integer operations (CFX), and two for simple arithmetic operations (SFX0 and SFX1).

The parallel execution units allow multiple instructions to execute in parallel and out of order. For example, a low-latency addition instruction that is issued to an SFX after an integer divide is issued to the CFX should finish executing before the higher latency divide instruction. Most instructions can make

results available to a subsequent instruction, but cannot update the architected GPR specified as its target operand ahead of the multiple-cycle divide instruction.

The common pipeline stages are as follows:

- Instruction fetch—Includes the clock cycles necessary to request an instruction and the time the memory system takes to respond to the request. Instructions retrieved are latched into the instruction queue (IQ) for subsequent consideration by the dispatcher.
  Instruction fetch timing depends on many variables, such as whether an instruction is in the on-chip instruction cache or the L2 cache. Those factors increase when it is necessary to fetch instructions from system memory and include the processor-to-bus clock ratio, the amount of bus traffic, and whether any cache coherency operations are required.
  Because there are so many variables, unless otherwise specified, the instruction timing examples in this chapter assume optimal performance and show the portion of the fetch stage in which the instruction is in the instruction queue. The fetch1 and fetch2 stages are primarily involved in retrieving instructions.
- The decode/dispatch stage fully decodes each instruction; most instructions are dispatched to the issue queues (however, **isync**, **rfi**, **sc**, **nop**s, and some other instructions do not go to issue queues).
- The issue queues, BIQ, GIQ, and FIQ, can accept as many as one, two, and two instructions, respectively, in a cycle. The following simplification covers most cases:
  — Instructions dispatch only from the two lowest IQ entries—IQ0 and IQ1.
  — A total of two instructions can be dispatched to the issue queues per clock cycle.

  Dispatch is treated as an event at the end of the decode stage. The issue stage reads source operands from rename registers and register files and determines when instructions are latched into the execution unit reservation stations. Note that the e500mc has 14 rename registers, one for each completion queue entry, so instructions cannot stall because of a shortage of rename registers.
  — Space must be available in the CQ for an instruction to decode and dispatch (this includes instructions that are assigned a space in the CQ but not in an issue queue).

  The general behavior of the issue queues is described as follows:
  — The GIQ accepts as many as two instructions from the dispatch unit per cycle. SFX0, SFX1, CFX, and all LSU instructions (including 64-bit loads and stores) are dispatched to the GIQ, shown in Figure 1-3.

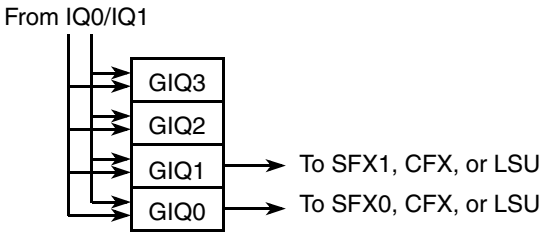


Figure 1-3. GPR Issue Queue (GIQ)

  — Instructions can be issued out-of-order from the bottom two GIQ entries (GIQ1–GIQ0). GIQ0 can issue to SFX0, CFX, and LSU. GIQ1 can issue to SFX1, CFX, and LSU.
  Note that SFX1 executes a subset of the instructions that can be executed in SFX0. The ability to identify and dispatch instructions to SFX1 increases the availability of SFX0 to execute more

computational-intensive instructions.
An instruction in GIQ1 destined for SFX1 or the LSU need not wait for an CFX instruction in GIQ0 that is stalled behind a long-latency divide.

— FIQ and BIQ only issue one instruction per cycle each to their respective reservation stations.

- The execute stage accepts instructions from its issue queue when the appropriate reservation stations are not busy. In this stage, the operands assigned to the execution stage from the issue stage are latched.
The execution unit executes the instruction (perhaps over multiple cycles), writes results on its result bus, and notifies the CQ when the instruction finishes. The execution unit reports any exceptions to the completion stage. Instruction-generated exceptions are not taken until the excepting instruction is next to retire.
  — Branch unit—The branch unit (BU) executes (resolves) all branch and CR logical instructions. Branches resolve in execution stage. If a branch is mispredicted, it takes five cycles for the next instruction to reach the execute stage.
  — Integer units. Two simple units (SFX0 and SFX1) handle add, subtract, shift, rotate and logical operations. The complex integer unit (CFX) executes multiplication and divide instructions Most integer instructions have a one-cycle latency, so results of these instructions are available one clock cycle after an instruction enters the execution unit.
  Integer multiply and divide instructions have longer latency, and the multiply and divide can overlap execution in most cases. Multiply operations are also pipelined.
  — The load/store unit (LSU), shown in Figure 1-4, has the following features:
    – Three-cycle load latency
    – Fully pipelined
    – Load miss queue
    – Load hits can continue to be serviced when the load miss queue is full.
    – As many as nine load misses to five distinct cache lines can be pipelined in parallel while L1 cache hits continue to be serviced.

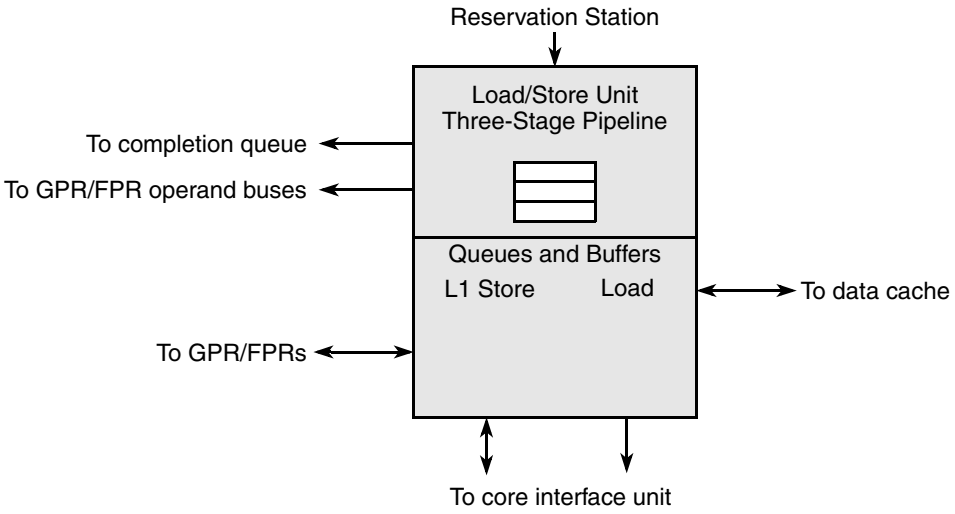


Figure 1-4. Three-Stage Load/Store Unit

- The complete and write-back stages maintain the correct architectural machine state and commit results to the architecture-defined registers in order. If completion logic detects a mispredicted branch or an instruction containing an exception status, subsequent instructions are cancelled, their execution results in rename registers are discarded, and the correct instruction stream is fetched. The complete stage ends when the instruction is retired. Two instructions can be retired per clock cycle. If no dependencies exist, as many as two instructions are retired in program order. Section 10.3.2, "Dispatch, Issue, and Completion Considerations," describes completion dependencies.
  The write-back stage occurs in the clock cycle after the instruction is retired.

1.4 Programming Model Overview

In general, the e500mc implements the registers and instructions as defined by the architecture (the Power ISA and Freescale implementation standards) and are fully described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture Processors*. The following sections provide a high level description and a listing of those resources that are implemented on the e500mc.

1.4.1 Register Model Overview

In general, registers on the e500mc are implemented as defined by the architecture. Any e500mc-specific differences from or extensions to the architecture are described in Chapter 2, "Register Model," of this manual.

The e500mc implements the following types of registers:

- Registers that contain values that are specified by using operands that are part of the instruction syntax defined by the Power ISA. These are as follows:
  — Thirty-two, 32-bit general purpose registers (GPRs), specified as follows:

  - **r**D indicates a GPR that is used as the destination or target of an integer computational, logical or load instruction.
  - **r**S indicates a GPR that is used as the source of an integer computational, logical, or store instruction.
  - **r**A, **r**B, or **r**C indicate GPRs that are used to hold values that are operated upon for computational or logical instructions, or that are used for an effective address (EA) or a decoration.
- Thirty-two, 64-bit floating-point registers (FPRs), specified as follows:
  - **fr**D indicates an FPR that is used as the destination or target of a floating-point instruction.
  - **fr**S indicates an FPR that is used as the source of a floating-point instruction.
  - **fr**A, **fr**B, or **fr**C indicate FPRs are used to hold values that are operated upon for floating-point instructions.

- Registers that are updated automatically to record a condition that occurs as a by-product of a computation:
  - The condition register, CR consists of eight 4-bit fields that record the results of certain operations which are typically used for testing and branching. The CR is implemented as defined in PowerISA. In addition, the CR can be accessed with special move to/move from instructions.
  - The integer exception register, XER, records conditions such as carries and overflows. The XER is an SPR and can be accessed with move to/move from SPR instructions (**mtspr** and **mfspr**).
  - The floating-point status and control register, FPSCR, records and controls exception conditions, such as overflows, controls the rounding mode, and indicates the type of result for certain floating-point operations.
  - The machine state register, MSR, is a supervisor-level register, although some fields can be written only by hypervisor-level software. It is used to configure operational behavior, such as setting the privilege level and enabling asynchronous interrupts. When an interrupt is taken, the MSR is stored into the appropriate save and restore register 1 (*x*SRR1) as determined by the interrupt type. The value in the *x*SRR1 is restored in the MSR when the appropriate return from interrupt is executed. The MSR, which is not an SPR, is accessed by the move to/move from MSR instructions (**mtmsr** and **mfmsr**). The external interrupt enable bit can be written separately with a Write MSR External Enable instruction (**wrtee** and **wrteei**).
- Most registers are defined as special-purpose registers (SPRs). All SPRs can be accessed by move to/move from SPR instructions (**mtspr** and **mfspr**), executed by software running at the appropriate privilege level, as indicated by the SPR summary in Table 2-2. Note that some SPRs are also updated by other mechanisms, such as the save and restore registers, which record the machine state when an exception is taken, and configuration and status registers, which are affected by internal signals. SPRs are listed in Section 2.2.1, “Special-Purpose Registers (SPRs).”
- Performance monitor registers (PMRs) are architecture-defined registers provided for configuring and programming the core-specific performance monitor. PMRs are similar to SPRs in that they are accessed by move to/from PMR instructions (**mtpmr** and **mfpmr**).

1.4.2 Instruction Model Overview

In general, instructions on the e500mc are implemented as defined by the architecture. Any e500mc-specific differences from or extensions to the architecture are described in Chapter 3, "Instruction Model," of this manual.

Table 3-47 lists the instructions implemented in the e500mc.

1.5 Core Revisions

This manual differentiates between different revisions of the e500mc in a few places, where it may be relevant. The revision of the core can be obtained by performing a **mfspr** to the PVR register. The following names are used in this manual to distinguish between the revisions of the core:

Rev 1.x Revision 1 of the core. PVR = 0x8023_xx1x.

Rev 2.x Revision 2 of the core. PVR = 0x8023_xx2x.

Rev 3.x Revision 3 of the core. PVR = 0x8023_xx3x.

1.6 Summary of Differences Between Previous e500 Cores

This section contains a series of differences outlining the changes from previous e500 cores. The changes described here are at a high level to help understand the programming model changes.

1.6.1 Changes from e500v2 to e500mc

e500mc contains several differences from the e500v2 core. Significant programming model changes occur from the removal of Signal Processing Engine (SPE) and the embedded floating point functionality and the addition of FPR based floating point as well as hypervisor partitioning support. User mode software can be recompiled if the software does not use explicit SPE or embedded floating point intrinsics. User level software that used any floating point software must also be re-linked since the manner in which floating point arguments to functions are passed is different. The floating point model of the e500mc is compatible with the e300 and e600 cores and should provide a seamless transition when migrating software from the e300 or the e600 to the e500mc.

A summary of the changes to the core is show in Table 1-1. This table is intended to be a general summary and not an explicit list of differences. Users should use this list to understand what major areas may require changes to their software when porting from the e500v2.

Table 1-1. Summary of e500mc and e500v2 Differences

Feature	e500v2	e500mc	Notes
Backside L2 cache	not present	present	An integrated backside L2 cache is present in e500mc. The backside L2 cache is described throughout this document.
SPE and embedded floating point	present	not present	SPE and embedded floating point (floating point done in the GPRs) is not present in e500mc. This makes the GPRs 32 bits in size as opposed to 64 bits.
FPR based floating-point	not present	present	FPR based floating-point (category Floating-Point) is present in e500mc. The floating point is binary compatible with e300 and e600. See Section 3.4.4.1, "Floating-Point Instructions."
Embedded hypervisor	not present	present	A new privilege level and associated instructions and registers are provided in e500mc to support partitioning and virtualization. Changes appear throughout the document.
Power management	uses MSR[WE] and HID0[DOZE,NAP,SLEEP] to enter power management states	uses SoC programming model to control power management and removes MSR[WE], HID0[DOZE,NAP,SLEEP]. Also adds the **wait** instruction.	How power management functions are invoked is now mostly controlled by writing SoC registers. See Chapter 8, "Power Management."
External proxy	not present	present	External proxy is a mechanism which allows the core to acknowledge an external input interrupt from the PIC when the interrupt is taken and provide the interrupt vector in a core register. See Section 4.9.6.3, "External Proxy."
Additional interrupt level for Debug interrupts	not present	present	A separate interrupt level for debug interrupts is provided and the associated save /restore registers DSRR0/DSRR1. See Section 4.9.16, "Debug Interrupt—IVOR15."
Processor signaling	not present	present	The **msgsnd** and **msgclr** instructions are provided to perform topology independent core to core doorbell interrupts. See Section 3.4.11.4, "Message Clear and Message Send Instructions."
External PID load/store	not present	present	Instructions are provided for supervisor/hypervisor level software to perform load and store operations using a different address space context. See Section 3.4.11.2, "External PID Load Store Instructions."
Decorated storage	not present	present	Instructions are provided for performing load and store operations to devices that include meta data that is interpreted by the target address. Devices in some SoCs utilize this facility for performing atomic memory updates like increments and decrements. See Section 3.4.3.2.8, "Decorated Load and Store Instructions."

Table 1-1. Summary of e500mc and e500v2 Differences (continued)

Feature	e500v2	e500mc	Notes
Lightweight synchronization	not present	Adds the **lwsync** instruction.	The **lwsync** instruction is provided for a faster form of memory barrier for load/store ordering to memory that is cached and coherent. See Section 3.4.10.1, “User-Level Cache Instructions” and Section 5.5.5, “Load/Store Operation Ordering.”
CoreNet	uses Core Complex Bus (CCB) as interconnect	uses CoreNet as an interconnect	CoreNet is a scalable non-retry based fabric used as an interconnect between cores and other devices in the SoC.
Cache stashing	not present	present	The capability to have certain SoC devices “stash” or pre-load data into a designated core L1 or L2 data cache is provided. The core is a passive recipient of such requests. See Section 5.2.2, “Cache Stashing.”
Machine check	provides machine check interrupt and HID0[RFXE] to control how the core treats machine check interrupts	provides error report, asynchronous machine check, and NMI interrupts. HID0[RFXE] is removed.	Machine check interrupts are divided into synchronous error reports, asynchronous machine checks, and NMI. How errors are reported are more conducive to a multi-core environment. See Section 4.9.3, “Machine Check Interrupt—IVOR1.”
Write shadow	not present	present	The capability to have all data written to the L1 data cache be “written through” to the L2 cache (or to memory) is provided. This provides a method of ensuring that any L1 cache error can be recovered from without loss of data. See Section 5.4.2, “Write Shadow Mode.”
Cache block size	32 bytes	64 bytes	e500mc contains a larger cache block/line/coherence granule size.
Number of variable size TLB entries	16	64	e500mc contains a larger number of variable size TLB entries and larger number of available page sizes. See Section 6.3.2, “L2 TLB Arrays.”

Chapter 2
Register Model

This chapter describes implementation-specific details of the register model as it is implemented on the e500mc core processors. It identifies all registers that are implemented on the e500mc core, but, with a few exceptions, does not include full descriptions of those registers and register fields that are implemented exactly as they are defined by the architecture (the Power ISA™ and the Freescale implementation standards). The *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* describes these registers.

It is important to note that a device that integrates the e500mc core may not implement all of the fields and registers that are defined here, and may interpret some fields more specifically than can be defined here. For specific details, refer to the e500mc Core Integration chapter in the reference manual for the device that incorporates the e500mc core.

Only registers associated with the programming model of the core are described in this chapter. Note that debug registers that are associated with external debug mode (EDM) are described in Chapter 9, "Debug and Performance Monitor Facilities."

2.1 Register Model Overview

Although this chapter organizes registers according to their functionality, they can be differentiated according to how they are accessed, as follows:

- General-purpose registers (GPRs)
  Used as source and destination operands for integer computation operations and for specifying the effective address. See Section 2.4.1, "General-Purpose Registers (GPRs)."
- Floating-point registers (FPRs)
  Used as source and destination operands for floating-point computation operations. See Section 2.5.1, "Floating-Point Registers (FPRs)."
- Special-purpose registers (SPRs)
  Accessed with the Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions. Section 2.2.1, "Special-Purpose Registers (SPRs)," lists SPRs.
- System-level registers that are not SPRs. These are as follows:
  - Machine state register (MSR). Accessed with the Move to Machine State Register (**mtmsr**) and Move from Machine State Register (**mfmsr**) instructions. See Section 2.7.1, "Machine State Register (MSR)."
  - Condition register (CR) bits are grouped into eight 4-bit fields, CR0–CR7, which are set as follows:
    - Specified CR fields can be set by a move to the CR from a GPR (**mtcrf**).

  - A specified CR field can be set by a move to the CR from another CR field (**mcrf**), or from the XER (**mcrxr**).
  - CR0 can be set as the implicit result of an integer instruction.
  - CR1 can be set as the implicit result of a floating-point instruction.
  - A specified CR field can be set as the result of an integer or floating-point compare instruction.

  See Section 2.6.1, "Condition Register (CR)."

- Performance monitor registers (PMRs). Similar to SPRs, PMRs are accessed by using dedicated move to/move from instructions (**mtpmr** and **mfpmr**). See Section 2.18, "Performance Monitor Registers (PMRs)."

2.2 e500mc Register Model

The following sections describe the e500mc core register model as defined by the architecture and the additional implementation-specific registers unique to the e500mc.

Freescale processors implement the following types of software-accessible registers:

- Registers used for integer operations such as general purpose registers (GPRs) and the integer exception register (XER). These registers are described in Section 2.4, "Registers for Integer Operations."
- Condition register (CR)—Used to record conditions such as overflows and carries that occur as a result of executing arithmetic instructions. CR is described in Section 2.6, "Registers for Branch Operations."
- Machine state register (MSR)—Used by the operating system to configure parameters such as user/supervisor mode, address space, and enabling of asynchronous interrupts. MSR is described in Section 2.7.1, "Machine State Register (MSR)."
- Special-purpose registers (SPRs). Accessed explicitly using **mtspr** and **mfspr** instructions and listed in Table 2-2 in Section 2.2.1, "Special-Purpose Registers (SPRs)."
- Performance monitor registers (PMRs). Accessed with move to and move from PMR instructions (**mtpmr** and **mfpmr**). PMRs are described in Section 2.18, "Performance Monitor Registers (PMRs)".

SPRs are grouped by function, as follows:

- Section 2.6, "Registers for Branch Operations"
- Section 2.7, "Processor Control Registers"
- Section 2.8, "Timer Registers"
- Section 2.9, "Interrupt Registers"
- Section 2.10, "Software-Use SPRs (SPRGs, GSPRGs, and USPRG0)"
- Section 2.11, "Branch Unit Control and Status Register (BUCSR)"
- Section 2.12, "Hardware Implementation-Dependent Register 0 (HID0)"
- Section 2.14, "L1 Cache Registers"
- Section 2.15, "L2 Cache Registers"

- Section 2.16, “MMU Registers”
- Section 2.17, “Internal Debug Registers”
- Section 2.18, “Performance Monitor Registers (PMRs)”

2.2.1 Special-Purpose Registers (SPRs)

SPRs are on-chip registers that control the use of the debug facilities, timers, interrupts, memory management unit, and other architected processor resources and are accessed with the **mtspr** and **mfspr** instructions.

Table 2-2 summarizes SPRs. Access is given by the lowest level of privilege required to access the SPR. The access methods listed in Table 2-1 appear in the access column of Table 2-2.

Table 2-1. SPR Access Methods

Access Method	Denotes access is available for...
User	Both **mfspr** and **mtspr** regardless of privilege level
User RO	Only **mfspr** regardless of privilege level
Guest supervisor	Both **mfspr** and **mtspr** when operating in supervisor mode (MSR[PR] = 0), regardless of the state of the MSR[GS] bit (that is, it is available in hypervisor state as well). For details, see Section 2.7.1, “Machine State Register (MSR).
Guest supervisor RO	Only **mfspr** when operating in supervisor mode (MSR[PR] = 0), regardless of the state of the MSR[GS] bit (that is, it is available in hypervisor state as well)
Hypervisor	Both **mfspr** and **mtspr** when operating in hypervisor mode (MSR[GS,PR] = 00)
Hypervisor RO	Only **mfspr** when operating in hypervisor mode (MSR[GS,PR] = 00)
Hypervisor WO	Only **mtspr** when operating in hypervisor mode (MSR[GS,PR] = 00)
Hypervisor R/Clear	Both **mfspr** and **mtspr** when operating in hypervisor mode (MSR[GS,PR] = 00); however, an **mtspr** only clears bit positions in the SPR that correspond to the bits set in the source GPR.

An **mtspr** or **mfspr** instruction that specifies an unsupported SPR number is considered an invalid instruction. The e500mc takes an illegal-operation program exception on all accesses to undefined SPRs (or read accesses to SPRs that are write-only and write accesses to SPRs that are read-only), regardless of MSR[GS,PR] and SPRN[5]values. For supported SPR numbers which are privileged, an **mfspr** or **mtspr** while in user mode (MSR[PR] = 1) causes a privilege operation program exception.

NOTE

The behavior of e500mc in user mode when attempting to access an unsupported privileged SPR number causes an illegal-operation program exception, not a privilege operation program exception as specified by the architecture.

Attempted access to a supported SPR while in guest supervisor state, which is Hypervisor-privileged, causes an embedded Hypervisor privilege exception. For example, attempting to read an SPR, which has "Hypervisor RO" privilege, while in guest supervisor state causes an embedded hypervisor privilege exception and subsequent interrupt. See Section 4.9.19, "Hypervisor Privilege Interrupt—IVOR41" for a complete list of actions that cause embedded hypervisor privilege exceptions.

This table summarizes SPRs.

Table 2-2. Special-Purpose Registers (SPRs)

SPR Abbreviation	Name	Defined SPR Number	Access	Section/Page
ATBL	Alternate time base register lower	526	User RO	2.8.6/2-16
ATBU	Alternate time base register upper	527	User RO	2.8.6/2-16
BUCSR	Branch unit control and status register[1]	1013	Hypervisor	2.11/2-27
CDCSR0	Core device control and status register	696	Hypervisor	2.13/2-28
CSRR0	Critical save/restore register 0	58	Hypervisor	2.9.1/2-16
CSRR1	Critical save/restore register 1	59	Hypervisor	2.9.1/2-16
CTR	Count register	9	User	2.6.3/2-11
DAC1	Data address compare 1[1]	316	Hypervisor	2.17.8/2-72
DAC2	Data address compare 2[1]	317	Hypervisor	2.17.8/2-72
DBCR0	Debug control register 0 [1]	308	Hypervisor	2.17.2/2-63
DBCR1	Debug control register 1[1]	309	Hypervisor	2.17.3/2-65
DBCR2	Debug control register 2[1]	310	Hypervisor	2.17.4/2-67
DBCR4	Debug control register 4[1]	563	Hypervisor	2.17.5/2-68
DBSR	Debug status register[1]	304	Hypervisor R/Clear	2.17.6/2-69
DBSRWR	Debug status register write[1]	306	Hypervisor	2.17.6/2-69
DDAM	Debug data acquisition message.	576	User	2.17.11/2-75
DEAR	Data exception address register	61	Guest supervisor[2]	2.8.5/2-16
DEC	Decrementer	22	Hypervisor	2.8.4/2-15
DECAR	Decrementer auto-reload	54	Hypervisor[3]	2.8.4/2-15
DEVENT	Debug event	975	User	2.17.10/2-74
DSRR0	Debug save/restore register 0	574	Hypervisor	2.9.1/2-16
DSRR1	Debug save/restore register 1	575	Hypervisor	2.9.1/2-16
EPCR	Embedded processor control register	307	Hypervisor	2.7.3/2-12
EPLC	External PID load context[1]	947	Guest supervisor[4]	2.16.7.1/2-61

Table 2-2. Special-Purpose Registers (SPRs) (continued)

SPR Abbreviation	Name	Defined SPR Number	Access	Section/Page
EPR	External proxy register	702	Guest supervisor RO[2]	2.9.5/2-19
EPSC	External PID store context[1]	948	Guest supervisor[4]	2.16.7.2/2-61
ESR	Exception syndrome register	62	Guest supervisor[2]	2.9.6/2-19
GDEAR	Guest data exception address register	381	Guest supervisor	2.8.5/2-16
GEPR	Guest external proxy register	380	Guest supervisor	2.9.5/2-19
GESR	Guest exception syndrome register	383	Guest supervisor	2.9.6/2-19
GIVOR2	Guest data storage interrupt offset	440	Hypervisor	2.9.3/2-18
GIVOR3	Guest instruction storage interrupt offset	441	Hypervisor	2.9.3/2-18
GIVOR4	Guest external input interrupt offset	442	Hypervisor	2.9.3/2-18
GIVOR8	Guest system call interrupt offset	443	Hypervisor	2.9.3/2-18
GIVOR13	Guest data TLB error interrupt offset	444	Hypervisor	2.9.3/2-18
GIVOR14	Guest instruction TLB error interrupt offset	445	Hypervisor	2.9.3/2-18
GIVPR	Guest interrupt vector prefix	447	Hypervisor	2.9.3/2-18
GPIR	Guest processor ID register	382	Guest supervisor[5]	2.9.7/2-21
GSPRG0	Guest SPR general 0	368	Guest supervisor	2.10/2-25
GSPRG1	Guest SPR general 1	369	Guest supervisor	2.10/2-25
GSPRG2	Guest SPR general 2	370	Guest supervisor	2.10/2-25
GSPRG3	Guest SPR general 3	371	Guest supervisor	2.10/2-25
GSRR0	Guest save/restore register 0	378	Guest supervisor	2.9.1/2-16
GSRR1	Guest save/restore register 1	379	Guest supervisor	2.9.1/2-16
HID0	Hardware implementation dependent register 0[1]	1008	Hypervisor	2.12/2-27
IAC1	Instruction address compare 1[1]	312	Hypervisor	2.17.7/2-72
IAC2	Instruction address compare 2[1]	313	Hypervisor	2.17.7/2-72
IVOR0	Critical input interrupt offset	400	Hypervisor	2.9.4/2-18
IVOR1	Machine check interrupt offset	401	Hypervisor	2.9.4/2-18
IVOR2	Data storage interrupt offset	402	Hypervisor	2.9.4/2-18
IVOR3	Instruction storage interrupt offset	403	Hypervisor	2.9.4/2-18
IVOR4	External input interrupt offset	404	Hypervisor	2.9.4/2-18
IVOR5	Alignment interrupt offset	405	Hypervisor	2.14.5/2-35
IVOR6	Program interrupt offset	406	Hypervisor	2.9.4/2-18
IVOR7	Floating-point unavailable interrupt offset.	407	Hypervisor	2.9.4/2-18

Table 2-2. Special-Purpose Registers (SPRs) (continued)

SPR Abbreviation	Name	Defined SPR Number	Access	Section/Page
IVOR8	System call interrupt offset	408	Hypervisor	2.9.4/2-18
IVOR9	APU unavailable interrupt offset	409	Hypervisor	2.9.4/2-18
IVOR10	Decrementer interrupt offset	410	Hypervisor	2.9.4/2-18
IVOR11	Fixed-interval timer interrupt offset	411	Hypervisor	2.9.4/2-18
IVOR12	Watchdog timer interrupt offset	412	Hypervisor	2.9.4/2-18
IVOR13	Data TLB error interrupt offset	413	Hypervisor	2.9.4/2-18
IVOR14	Instruction TLB error interrupt offset	414	Hypervisor	2.9.4/2-18
IVOR15	Debug interrupt offset	415	Hypervisor	2.9.4/2-18
IVOR35	Performance monitor interrupt offset	531	Hypervisor	2.9.4/2-18
IVOR36	Processor doorbell interrupt offset	532	Hypervisor	2.9.4/2-18
IVOR37	Processor doorbell critical interrupt offset	533	Hypervisor	2.9.3/2-18
IVOR38	Guest processor doorbell interrupt offset	432	Hypervisor	2.9.4/2-18
IVOR39	Guest processor doorbell critical and machine check interrupt offset	433	Hypervisor	2.9.4/2-18
IVOR40	Hypervisor system call interrupt offset	434	Hypervisor	2.9.4/2-18
IVOR41	Hypervisor privilege interrupt offset	435	Hypervisor	2.9.4/2-18
IVPR	Interrupt vector prefix	63	Hypervisor	2.9.3/2-18
L1CFG0	L1 cache configuration register 0	515	User RO	2.14.4/2-34
L1CFG1	L1 cache configuration register 1	516	User RO	2.14.5/2-35
L1CSR0	L1 cache control and status register 0[1]	1010	Hypervisor	2.14.1/2-29
L1CSR1	L1 cache control and status register 1[1]	1011	Hypervisor	2.14.2/2-32
L1CSR2	L1 cache control and status register 2[1]	606	Hypervisor	2.14.3/2-33
L2CAPTDATAHI[6]	L2 cache error capture data high	988	Hypervisor	2.15.4/2-42
L2CAPTDATALO[6]	L2 cache error capture data low	989	Hypervisor	2.15.4/2-42
L2CAPTECC[6]	L2 cache error capture ECC syndrome	990	Hypervisor	2.15.4/2-42
L2CFG0[6]	L2 cache configuration register 0	519	User RO	2.15/2-36
L2CSR0[6]	L2 cache control and status register 0[1]	1017	Hypervisor	2.15.2/2-37
L2CSR1[6]	L2 cache control and status register 1[1]	1018	Hypervisor	2.15/2-36
L2ERRADDR[6]	L2 cache error address	722	Hypervisor	2.15.4/2-42
L2ERRATTR[6]	L2 cache error attribute	721	Hypervisor	2.15.4/2-42
L2ERRCTL[6]	L2 cache error control	724	Hypervisor	2.15.4/2-42
L2ERRDET[6]	L2 cache error detect	991	Hypervisor	2.15.4/2-42
L2ERRDIS[6]	L2 cache error disable	725	Hypervisor	2.15.4/2-42
L2ERREADDR[6]	L2 cache error extended address	723	Hypervisor	2.15.4/2-42

Table 2-2. Special-Purpose Registers (SPRs) (continued)

SPR Abbreviation	Name	Defined SPR Number	Access	Section/Page
L2ERRINJCTL[6]	L2 cache error injection control	987	Hypervisor	2.15.4/2-42
L2ERRINJHI[6]	L2 cache error injection mask high	985	Hypervisor	2.15.4/2-42
L2ERRINJLO[6]	L2 cache error injection mask low	986	Hypervisor	2.15.4/2-42
L2ERRINTEN[6]	L2 cache error interrupt enable	720	Hypervisor	2.15.4/2-42
LPIDR	Logical PID register[1]	338	Hypervisor	2.16.1/2-49
LR	Link register	8	User	2.6.2/2-11
MAS0	MMU assist register 0[1]	624	Guest supervisor	2.16.6.1/2-53
MAS1	MMU assist register 1[1]	625	Guest supervisor	2.16.6.2/2-54
MAS2	MMU assist register 2[1]	626	Guest supervisor	2.16.6.3/2-55
MAS3	MMU assist register 3[1]	627	Guest supervisor	2.16.6.4/2-56
MAS4	MMU assist register 4[1]	628	Guest supervisor	2.16.6.5/2-57
MAS5	MMU assist register 5[1]	339	Hypervisor	2.16.6.6/2-58
MAS6	MMU assist register 6[1]	630	Guest supervisor	2.16.6.7/2-58
MAS7	MMU assist register 7[1]	944	Guest supervisor	2.16.6.8/2-59
MAS8	MMU assist register 8[1]	341	Hypervisor	2.16.6.9/2-60
MCAR	Machine check address register	573	Hypervisor RO	2.9.8/2-21
MCARU	Machine check address register upper	569	Hypervisor RO	2.9.8/2-21
MCSR	Machine check syndrome register	572	Hypervisor	2.9.9/2-22
MCSRR0	Machine-check save/restore register 0	570	Hypervisor	2.9.1/2-16
MCSRR1	Machine-check save/restore register 1	571	Hypervisor	2.9.1/2-16
MMUCFG	MMU configuration register	1015	Hypervisor RO	2.16.4/2-51
MMUCSR0	MMU control and status register 0[1]	1012	Hypervisor	2.16.3/2-50
MSRP	MSR protect[1]	311	Hypervisor	2.7.2/2-12
NPIDR[7]	Nexus processor ID register	517	User	2.17.12/2-75
NSPC	Nexus SPR access configuration	984	Hypervisor	2.17.9/2-73
NSPD	Nexus SPR access data	983	Hypervisor	2.17.9/2-73
PID	Process ID register[1]	48	Guest supervisor	2.16.2/2-50
PIR	Processor ID register	286	Guest supervisor[2]	2.9.7/2-21
PVR	Processor version register	287	Guest supervisor RO	2.7.4/2-13
SPRG0	SPR general 0	272	Guest supervisor[2]	2.10/2-25
SPRG1	SPR general 1	273	Guest supervisor[2]	2.10/2-25

Table 2-2. Special-Purpose Registers (SPRs) (continued)

SPR Abbreviation	Name	Defined SPR Number	Access	Section/Page
SPRG2	SPR general 2	274	Guest supervisor[2]	2.10/2-25
SPRG3	SPR general 3	259	User RO[2]	2.10/2-25
SPRG3	SPR general 3	275	Guest supervisor[2]	2.10/2-25
SPRG4	SPR general 4	260	User RO	2.10/2-25
SPRG4	SPR general 4	276	Guest supervisor	2.10/2-25
SPRG5	SPR general 5	261	User RO	2.10/2-25
SPRG5	SPR general 5	277	Guest supervisor	2.10/2-25
SPRG6	SPR general 6	262	User RO	2.10/2-25
SPRG6	SPR general 6	278	Guest supervisor	2.10/2-25
SPRG7	SPR general 7	263	User RO	2.10/2-25
SPRG7	SPR general 7	279	Guest supervisor	2.10/2-25
SPRG8	SPRG8	604	Hypervisor	2.10/2-25
SPRG9	SPRG9	605	Guest supervisor	2.10/2-25
SRR0	Save/restore register 0	26	Guest supervisor[2]	2.9.1/2-16
SRR1	Save/restore register 1	27	Guest supervisor[2]	2.9.1/2-16
SVR	System version register	1023	Guest supervisor RO	2.7.5/2-13
TBL(R)	Time base lower	268	User RO	2.8.3/2-15
TBL(W)	Time base lower	284	Hypervisor	2.8.3/2-15
TBU(R)	Time base upper	269	User RO	2.8.3/2-15
TBU(W)	Time base upper	285	Hypervisor	2.8.3/2-15
TCR	Timer control register	340	Hypervisor	2.8.1/2-14
TLB0CFG	TLB configuration register 0	688	Hypervisor RO	2.16.5/2-51
TLB1CFG	TLB configuration register 1	689	Hypervisor RO	2.16.5/2-51
TSR	Timer status register	336	Hypervisor R/Clear	2.8.2/2-15
USPRG0 (VRSAVE)	User SPR general 0[8]	256	User	2.10/2-25
XER	Integer exception register	1	User	2.4.2/2-10

[1] Writing to these registers requires synchronization, as described in Section 3.3.3, “Synchronization Requirements.”

[2] When these registers are accessed in guest supervisor state, the accesses are mapped to their analogous guest SPRs (for example, DEAR is mapped to GDEAR). See Section 2.3, “Register Mapping in Guest–Supervisor State.”

[3] DECAR is defined by the architecture to be write-only, however the e500mc allows it to be read.

4 Certain fields in the register are only writeable when in hypervisor state.
5 This register is only writeable in hypervisor state, but can be read in guest supervisor state.
6 On Cores that do not provide an L2 cache, these registers still exist, but always read as zero.
7 NPIDR contents are transferred to the Nexus port whenever it is written.
8 USPRG0 is a separate physical register from SPRG0.

2.3 Register Mapping in Guest–Supervisor State

Accesses to certain hypervisor state registers are automatically redirected to the appropriate guest state registers when in the guest–supervisor state. This helps to improve emulation efficiency and provides a common programming model for operating systems that may want to run either under control of a hypervisor or directly on the hardware without a hypervisor. This also removes the requirement for the hypervisor state software to handle hypervisor privilege interrupts for these registers and to make the required emulated changes to the guest state for these high-use registers.

Accesses to the registers listed in “Register Accessed” column in Section Table 2-3., “Register Mapping in Guest–Supervisor State are changed by the processor to the registers listed in “Register mapped to” column in the table when MSR[PR] = 0 and MSR[GS] = 1. Access to these registers are not mapped when in hypervisor state (MSR[PR] = 0 and MSR[GS] = 0) or when operating unprivileged (MSR[PR] = 1), except that an unprivileged access to SPRG3 (SPR 259) is also mapped to GSPRG3.

Table 2-3. Register Mapping in Guest–Supervisor State

Register Accessed	Register Mapped to	Notes
SRR0	GSRR0	Access mapped during **mtspr**, **mfspr**.
SRR1	GSRR1	Access mapped during **mtspr**, **mfspr**.
EPR	GEPR	Access mapped during **mfspr**.
ESR	GESR	Access mapped during **mtspr**, **mfspr**.
DEAR	GDEAR	Access mapped during **mtspr**, **mfspr**.
PIR	GPIR	Access mapped during **mfspr**.
SPRG0	GSPRG0	Access mapped during **mtspr**, **mfspr**.
SPRG1	GSPRG1	Access mapped during **mtspr**, **mfspr**.
SPRG2	GSPRG2	Access mapped during **mtspr**, **mfspr**
SPRG3	GSPRG3	Access mapped during **mtspr**, **mfspr**
SPRG3 (259)	GSPRG3	Access mapped during **mfspr**

2.4 Registers for Integer Operations

The following sections describe registers defined for integer computational instructions.

2.4.1 General-Purpose Registers (GPRs)

GPR0–GPR31 provide operand space for supporting integer operations. The instruction formats provide 5-bit fields for specifying the GPRs to be used in the execution of the instruction. Each GPR is a 32-bit register and can be used to contain effective address and integer data.

The GPRs are implemented as defined by the Power ISA and as described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

2.4.2 Integer Exception Register (XER)

NOTE

XER is an SPR. The e500mc implements the XER as it is defined by the architecture.

XER bits are set based on the operation of an instruction considered as a whole, not on intermediate results. For example, the Subtract from Carrying instruction (**subfc**) specifies the result as the sum of three values, but it sets bits in the XER based on the entire operation, not on an intermediate sum.

2.5 Registers for Floating-Point Operations

The following sections describe registers defined for floating-point computational instructions.

2.5.1 Floating-Point Registers (FPRs)

FPR0–FPR31 provide operand space for supporting floating-point operations. The instruction formats provide 5-bit fields for specifying the FPRs to be used in the execution of the instruction. Each FPR is a 64-bit register and can be used to contain single-precision or double-precision floating-point data.

The FPRs are implemented as defined by the Power ISA and as described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

2.5.2 Floating-Point Status and Control Register (FPSCR)

The FPSCR contains all floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard. The FPSCR is implemented as defined by the Power ISA and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

For e500mc, if FPSCR[NI] is set, denormalized values are treated as appropriately signed 0 values. That is, if a denormalized number is an input to a floating point operation, that denormalized number is treated as 0 with the same sign as the denormalized number. If the result of a floating point operation produces a denormalized number, the result produced and written to the destination register is an appropriately signed 0.

2.6 Registers for Branch Operations

This section describes registers used by branch and condition register operations.

2.6.1 Condition Register (CR)

The e500mc implements the condition register as it is defined by the architecture for integer instructions.

2.6.2 Link Register (LR)

The e500mc implements the link register as it is defined by the architecture.

The link register can be used to provide the branch target address for a Branch Conditional to LR instruction, and it holds the return address after branch and link instructions.

Note that the link register is an SPR.

2.6.3 Count Register (CTR)

The e500mc implements the count register as it is defined by the architecture. The count register can be used to hold a loop count that can be decremented and tested during execution of branch instructions that contain an appropriately encoded BO field. If the count register value is 0 before being decremented, it is –1 afterward. The count register can be used to hold the branch target address for a Branch Conditional to CTR (**bcctr***x*) instruction.

Note that the count register is an SPR.

2.7 Processor Control Registers

This section addresses machine state, processor ID, processor version registers.

2.7.1 Machine State Register (MSR)

The machine state register (MSR), shown in Figure 2-1, is used to define the processor state, that is, enabling and disabling of interrupts and debugging exceptions, address translation for instruction and data memory accesses, enabling and disabling some functionality, and specifying whether the processor is in supervisor or user mode.

When the core runs in guest–supervisor state (MSR[GS] = 1, MSR[PR] = 0), some MSR bits are not writable. If the MSR is written in guest–supervisor state in any manner, including a **mtmsr**, **rfgi**, or **rfi**, or as the result of taking an interrupt serviced in guest state, MSR[GS] is not changed.

Certain MSR bits may be changed in guest–supervisor state if permission to do so is enabled by the hypervisor program. MSR[UCLE,DE,PMM] are writable if the corresponding MSRP-defined bits are cleared. See Section 2.7.2, “Machine State Register Protect Register (MSRP).” MSRP is writable only in hypervisor state. When MSR is written in guest state, bits protected by MSRP bits that are set, are not written and remain unmodified. All other MSR bits are written with the updated values. An attempt to write the MSRP in guest–supervisor state results in a hypervisor privilege exception.

Changing PR, GS, IS, or DS using the **mtmsr** instruction requires a context-synchronizing operation before the effects of the change are guaranteed to be visible. Prior to the context synchronization, these bits can change at any time and with any combination. Changes in DS, or IS can cause an implicit branch since these bits are used to compute the virtual address for instruction translation and instructions may be

fetched and executed from any context from any permutation of these bits. Software should guarantee that a translation exists for each of the permutations of these address space bits and that translation has the same characteristics, including permissions and RPN fields. For this reason, it is unwise to use **mtmsr** to change these bits and such changes should only be done through return from interrupt type instructions, which provide the context synchronization atomically with instruction execution.

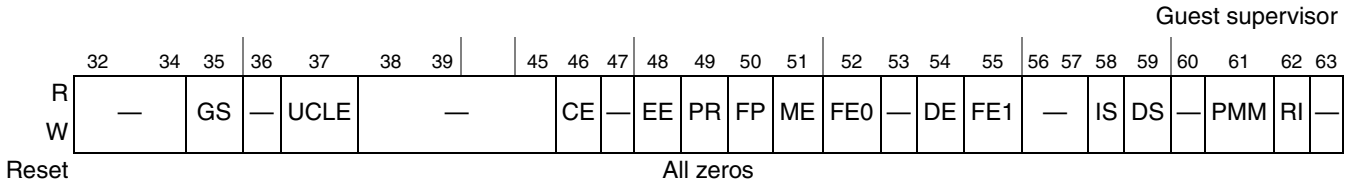


Figure 2-1. Machine State Register (MSR)

When an interrupt occurs, MSR contents of the interrupted process are automatically saved to the save/restore register 1 (*x*SRR1) appropriate to the interrupt, and the MSR is altered to values predetermined for the interrupt taken. At the end of the interrupt handler, the appropriate return from interrupt instruction restores the values in the save/restore register 1 (*x*SRR1) to the MSR.

MSR contents are read into a GPR using **mfmsr**. The contents of a GPR can be written to MSR using **mtmsr**. The write MSR external enable instructions (**wrtee** and **wrteei**) can be used to set or clear MSR[EE] without affecting other MSR bits.

The e500mc does not implement the WE bit found in some previous e500 cores. Power management operations on SoCs using the e500mc are handled through an SoC programming model. Refer to the reference manual for the integrated device.

2.7.2 Machine State Register Protect Register (MSRP)

The e500mc implements the MSRP as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. It provides the ability to write MSR[UCLE,DE,PMM] when the machine is in the guest–supervisor state (MSR[PR] = 0 and MSR[GS] = 1) by any operation that modifies the MSR (**mtmsr**, **rfi**, **rfgi**, and MSR change on an interrupt directed to the guest state). An attempt to read or write MSRP when not in the hypervisor state results in a hypervisor privilege exception when MSR[PR] = 0 and a privilege exception when MSR[PR] = 1.

MSRP settings also affect the execution of Cache Locking instructions and **mtpmr**/**mfpmr** instructions.

A change to MSRP requires a context synchronizing operation to be performed before the effects of the change are guaranteed to be visible in the current context.

2.7.3 Embedded Processor Control Register (EPCR)

The EPCR controls whether certain interrupts are directed to the hypervisor state or to the guest–supervisor state and suppresses debug events when in hypervisor state. The e500mc implements the EPCR as it is defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

2.7.4 Processor Version Register (PVR)

The e500mc implements the PVR, shown in Figure 2-2, as defined by the architecture. The read-only value identifies the core's version and revision level of the processor, distinguishing between processors that differ in attributes that may affect software.

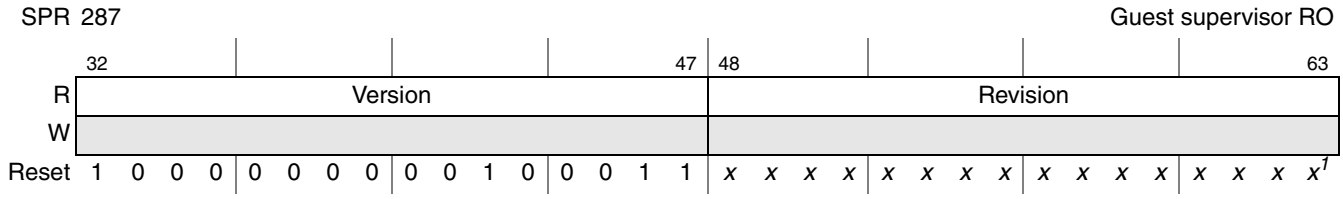


Figure 2-2. Processor Version Register (PVR)

1 *xxxx* may represent different revisions or manufacturing information for the core. Normally software uses the upper 16 bits of PVR to identify the core.

This table describes the PVR fields.

Table 2-4. PVR Field Descriptions

Bits	Name	Description
32–47	Version	A 16-bit number that identifies the version of the processor. Different version numbers indicate major differences between processors, such as which optional facilities and instructions are supported.
48–63	Revision	A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between processors having the same version number, such as clock rate and engineering change level.

2.7.5 System Version Register (SVR)

SVR, shown in this figure, contains a read-only SoC-dependent value; consult the documentation for the integrated device.

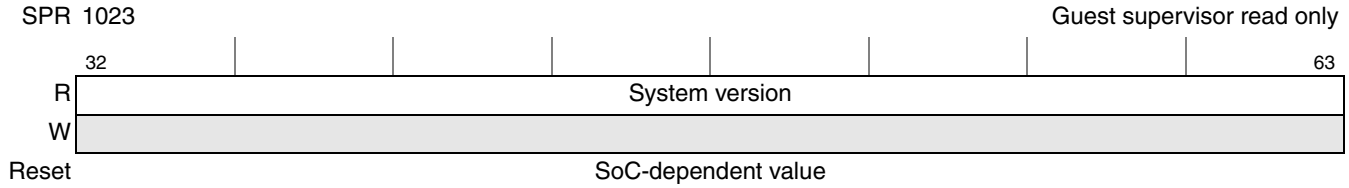


Figure 2-3. System Version Register (SVR)

2.8 Timer Registers

The time base (TB), decrementer (DEC), fixed-interval timer (FIT), and watchdog timer provide timing functions for the system. The e500mc provides the ability to select any of the TB bits to trigger watchdog and fixed-interval timer events, as shown in this figure.

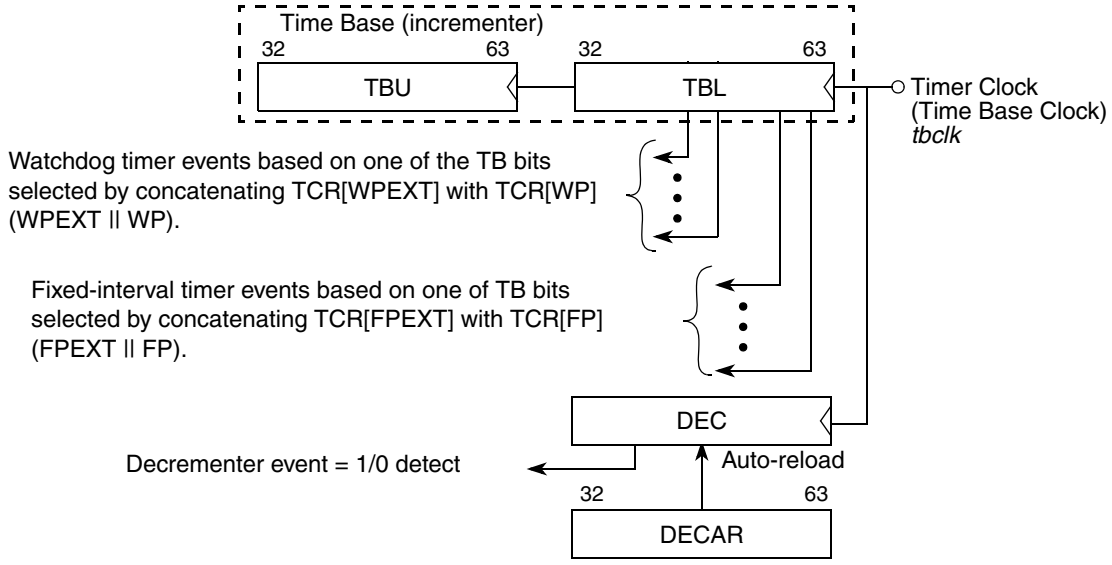


Figure 2-4. Relationship of Timer Facilities to the Time Base

Note the following characteristics of the e500mc time base implementation:

- e500mc time base is clocked only by the SoC (TBCLK)
- The only enable/disable control over the time base is the TBEN core signal, controlled by the SoC through a memory-mapped register, allowing control of stopping and starting the time base on any core. Refer to the reference manual for the integrated device.
- **mftb** works as it did in the original PowerPC architecture

The e500mc registers involved in timing are described as follows:

- The TB is a long-period counter driven at an implementation-dependent frequency.
- The DEC provides a way to signal an exception after a specified period of time base tics.
- Software can select from one of 64 TB bits to signal a fixed-interval interrupt whenever the bit transitions from 0 to 1. It is typically used to trigger periodic system maintenance functions. .
- The watchdog timer, also a selected TB bit, provides a way to signal a critical exception when the selected bit transitions from 0 to 1. It is typically used for system error recovery. If software does not respond in time to the initial interrupt by clearing the associated status bits in the TSR before the next expiration of the watchdog timer interval, a watchdog timer-generated processor reset may result, if so enabled.

All timer facilities must be initialized during start-up.

2.8.1 Timer Control Register (TCR)

The e500mc implements the timer control register as defined by *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. The implementation of the integrated device determines the behavior of the TCR[WRC]. Consult the "Register Summary" chapter in the core section of the integrated device reference manual.

The architecture definition for timer control register fields is described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

2.8.2 Timer Status Register (TSR)

Except as described in this section, the e500mc implements the timer status register as it is defined by the architecture. The 32-bit TSR contains status on timer events and the most recent watchdog timer-initiated processor reset. All TSR bits function as write-1-to-clear, except TSR[WRS] which is nonwriteable (nonclearable).

As a write-1-to-clear register, TSR can be changed only by software by writing a mask of 1 bits indicating which bit positions are to be cleared. When the TSR is written by an **mtspr**, WRS bits are not cleared, regardless of the mask bits supplied with the GPR used for writing. Logically, the instruction **mttsr r**A becomes the following:

```
mask = RA & 0xcfffffff;
TSR = TSR & ~mask;
```

This change prevents software from clearing a watchdog time-out that should result in the action defined in TCR[WRC], in which these bits are reflected into the TSR[WRS] when the watchdog times out. Without this change, it is theoretically possible that these bits could be cleared prior to the SoC seeing the bits change, causing the watchdog action to fail.

2.8.3 Time Base (TBU and TBL)

The e500mc implements the time base registers as they are defined by the architecture. The time base (TB) is a 64-bit register, but the architecture provides SPRs to access the upper 32 bits and lower 32 bits. Reading the lower 32 bits of the time base (TBL, SPR 268), places the lower 32 bits of the time base into the destination GPR. Reading the upper 32 bits of the time base (TBU, SPR 269) places the upper 32 bits of the time base into the lower 32 bits of the destination GPR. Writing the time base is done only through writing the upper 32 (SPR 285) and lower 32 (SPR 284) time base bits through two separate **mtspr** instructions. The time base register provides timing functions for the system. The time base register is a volatile resource and must be initialized during start-up.

For e500mc, the time base can be read in hypervisor state through the SPRs used for writing (284 and 285), although the architecture defines it as a write-only register.

NOTE

Software should not read the time base through these registers as future processors may not allow such behavior.

2.8.4 Decrementer Register (DEC)

The e500mc implements the decrementer register as it is defined by the architecture. The decrementer register is a 32-bit decrementing counter that is decremented at the same rate as the time base is incremented. It provides a way to signal a decrementer interrupt after a specified number of time base tics have occurred. It can be configured to signal an interrupt when DEC is decremented from 1 to 0. The DEC can be configured through the TCR to perform different actions when it is decremented from 1 to 0:

- It can stop decrementing;
- It can be auto-reloaded from DECAR (see Section 2.8.5, “Decrementer Auto-Reload Register (DECAR).”);
- It can signal a decrementer exception and take an asynchronous interrupt when External Interrupts are enabled or when the processor is in guest state (MSR[GS]=1).

The decrementer register is typically used as a general-purpose software timer. Note that writing DEC with zeros by using an **mtspr[DEC]** does not automatically generate a decrementer exception.

2.8.5 Decrementer Auto-Reload Register (DECAR)

The e500mc implements the DECAR as it is defined by the architecture. If the auto-reload function is enabled (TCR[ARE] = 1), the auto-reload value in DECAR is written to the decrementer register when it decrements from 1 to 0.

For e500mc, the DECAR can be read in hypervisor state, although the architecture defines it as a write-only register.

NOTE

Software should not read the DECAR as future processors may not allow such behavior.

2.8.6 Alternate Time Base Registers (ATBL and ATBU)

The alternate time base counter (ATB) register is implemented as defined by the architecture and described in the *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors*. The ATB is a 64-bit counter that increments at an implementation dependent frequency. The ATB is a 64-bit register, but the architecture provides SPRs to access the upper 32 bits and lower 32-bits. Reading the lower 32 bits of the time base (ATBL), places the lower 32 bits of the time base into the destination GPR. Reading the upper 32 bits of the time base (ATBU) places the upper 32 bits of the time base into the lower 32 bits of the destination GPR.

On the e500mc, the frequency of the ATB increment is the core frequency. ATB is read-only accessible in user and supervisor mode.

The ATBL register is a 64-bit register

2.9 Interrupt Registers

This section describes the following register bits and their fields:

2.9.1 Save/Restore Registers (*x*SRR0/*x*SRR1)

The e500mc implements the following sets of save restore registers, which support the different types of interrupts implemented on the e500mc.

- Standard save/restore registers (SRR0 and SRR1)
- Critical save/restore registers (CSRR0 and CSRR1)

- Debug save/restore registers (DSRR0 and DSRR1)
- Machine check save/restore registers (MCSRR0 and MCSRR1)
- Guest save/restore registers (GSRR0 and GSRR1). Note that when executing in guest state (MSR[GS] = 1), accesses to SRR0/SRR1 are mapped to GSRR0/GSRR1 when any **mfspr** or **mtspr** instruction is executed. See Section 2.3, "Register Mapping in Guest–Supervisor State."

These registers are implemented as they are defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. On an interrupt, *x*SRR0 holds the address of the instruction where the interrupted process should resume, typically either the current or subsequent instruction. The instruction is interrupt-specific, although for instruction-caused exceptions, it is typically the address of the instruction that caused the interrupt. When the appropriate Return from Interrupt instruction (**rfi**, **rfci**, **rfdi**, **rfmci**, or **rfgi**) executes, instruction execution continues at the address in *x*SRR0.

When **rfi** is executed from guest supervisor state, the instruction is mapped to **rfgi** and uses GSRR0 and GSRR1.

*x*SRR1 is provided to save machine state when an interrupt is taken and to restore it when control is passed back, typically to the interrupted process. When an interrupt is taken, certain MSR settings, specific to the interrupt, are placed in *x*SRR1. When the appropriate Return from Interrupt instruction executes, *x*SRR1 contents are placed into MSR. *x*SRR1 bits that correspond to reserved MSR bits are also reserved.

Note that a pair of save/restore registers is affected only by the corresponding interrupt or an **mtspr** that explicitly targets one of the registers. Reserved MSR bits may be altered by Return from Interrupt instructions if set in the xSRR1 register.

For specific information about how the save/restore registers are set, see the individual interrupt descriptions in Chapter 4, "Interrupts and Exceptions."

2.9.2 (Guest) Data Exception Address Register (DEAR/GDEAR)

The e500mc implements DEAR/GDEAR as it is defined by the architecture. DEAR is loaded with the effective address of a data access (caused by a load, store, or cache management instruction) that results in an alignment, data TLB miss, or DSI exception.

GDEAR is the same as the DEAR. When a DSI or a data TLB error interrupt is taken in the guest state, GDEAR is set to the EA of the data access causing the exception instead of DEAR.

GDEAR is supervisor privileged (MSR[PR] = 0) and is read/write. Accesses to DEAR in guest–supervisor state (MSR[GS]=10, MSR[PR] = 10) are mapped to GDEAR for **mtspr** and **mfspr** instructions, in the same manner as other guest registers.

Note that even when DSI interrupts are directed to the guest state by means of EPCR[DSIGS], the DSI may be directed to the hypervisor if a virtualization fault is set on the TLB entry that caused the DSI. See Section 2.7.3, "Embedded Processor Control Register (EPCR). Therefore, the DEAR is set instead of GDEAR.

2.9.3 (Guest) Interrupt Vector Prefix Register (IVPR/GIVPR)

The e500mc implements IVPR and guest IVPR (GIVPR) as they are defined by the architecture. They are used with IVORs and GIVORs, respectively, to determine the vector address. (G)IVPR[32–47] provides the high-order 16 bits of the address of the exception processing routines. The 16-bit vector offsets (IVORs) are concatenated to the right of (G)IVPR to form the address of the exception processing routine.

When an interrupt is directed to the hypervisor state, IVPR and IVOR*n* are used to form the address of the exception processing routine. When an interrupt is directed to the guest–supervisor state, GIVPR and GIVOR*n* are used to form the address of the exception processing routine.

IVPR and GIVPR are 32 bit registers on e500mc.

2.9.4 (Guest) Interrupt Vector Offset Registers (IVORs/GIVORs)

The e500mc implements the IVORs and guest IVORs (GIVORs) as defined by the architecture, but use only (G)IVOR*n*[48–59], as shown in Figure 2-5, to hold the quad-word index from the base address provided by the IVPR for each interrupt type.

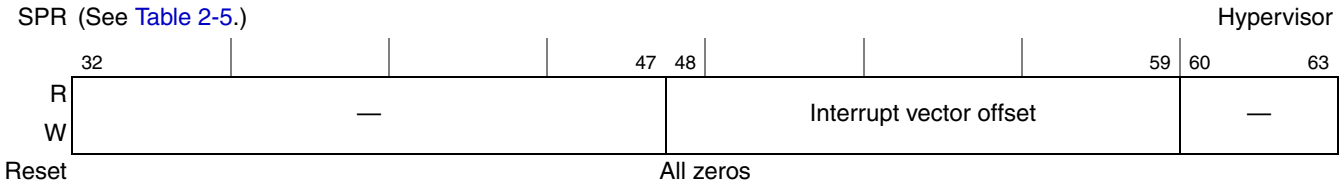


Figure 2-5. (Guest) Interrupt Vector Offset Registers ((G)IVORs)

This table shows the (G)IVORs implemented on the e500mc.

Table 2-5. IVOR Assignments

IVOR Number	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	APU unavailable
IVOR10	Decrementer
IVOR11	Fixed-interval timer interrupt
IVOR12	Watchdog timer interrupt
IVOR13	Data TLB error

Table 2-5. IVOR Assignments (continued)

IVOR Number	Interrupt Type
IVOR14	Instruction TLB error
IVOR15	Debug
IVOR35	Performance monitor
IVOR36	Processor doorbell interrupt
IVOR37	Processor doorbell critical interrupt
IVOR38	Guest processor doorbell
IVOR39	Guest processor doorbell critical and machine check
IVOR40	Hypervisor system call
IVOR41	Hypervisor privilege
Guest-Type IVORs	
GIVOR2	Guest data storage interrupt
GIVOR3	Guest instruction storage interrupt
GIVOR4	Guest external input
GIVOR8	Guest system call
GIVOR13	Guest data TLB error
GIVOR14	Guest instruction TLB error

2.9.5 (Guest) External Proxy Register (EPR/GEPR)

The external proxy register (EPR/GEPR) is implemented as it is defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. It is used to convey the peripheral-specific interrupt vector associated with the external input interrupt triggered by the programmable interrupt controller (PIC) in the integrated device. The external proxy facility is described in Section 4.9.6.3, "External Proxy."

When executing in the guest supervisor state, any read accesses to the EPR are mapped to GEPR upon executing **mfspr**. See Section 2.3, "Register Mapping in Guest–Supervisor State," for more details.

EPR is not writable, however GEPR is writeable.

2.9.6 (Guest) Exception Syndrome Register (ESR/GESR)

The (ESR/GESR) are defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. Figure 2-6 shows the ESR/GESR as it is implemented on the e500mc. GESR is used to post–exception syndrome status when an interrupt is taken that is directed to the guest state. ESR is used to post–exception syndrome status when an interrupt is taken that is directed to the hypervisor state. GESR fields are identical to those in the ESR.

When executing in the guest supervisor state any accesses to the ESR are mapped to GESR upon executing **mtspr** or **mfspr**. See Section 2.3, "Register Mapping in Guest–Supervisor State," for more details.

The (G)ESR provides a way to differentiate among exceptions that can generate an interrupt type. When an interrupt is generated, bits corresponding to the specific exception that generated the interrupt are set and all other (G)ESR bits are cleared. Other interrupt types do not affect (G)ESR contents. The (G)ESR does not need to be cleared by software. Table 2-6 shows (G)ESR bit definitions. For machine check exceptions, the e500mc uses the MCSR, described in Section 2.9.9, "Machine Check Syndrome Register (MCSR)."

The (G)ESR implementation differs from the architecture in the following respects:

- The e500mc does not implement AP, PUO, SPV, VLEMI, MIF, or XTE
- The e500mc implements the EPID field.

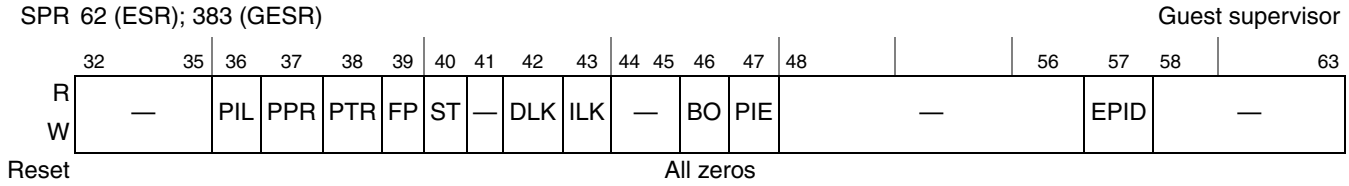


Figure 2-6. (Guest) Exception Syndrome Register (ESR/GESR)

This table describes (G)ESR fields and shows associated interrupts.

NOTE

(G)ESR information is incomplete, so system software may need to identify the type of instruction that caused the interrupt, examine the TLB entry, and examine the (G)ESR to identify the exception or exceptions fully. For example, a data storage interrupt may be caused both by a protection violation exception and by a byte-ordering exception. System software would have to look beyond (G)ESR[BO], such as the state of MSR[PR] in (G)SRR1 and the TLB entry page protection bits, to determine whether a protection violation also occurred.

Table 2-6. ESR/GESR Field Descriptions

Bits	Name	Syndrome	Interrupt Types
32–35	—	Reserved	—
36	PIL	Illegal instruction exception	Program
37	PPR	Privileged instruction exception	Program
38	PTR	Trap exception	Program
39	FP	Floating-point operations	Alignment, data storage, data TLB, program
40	ST	Store operation	Alignment, DSI, DTLB error
41	—	Reserved	—
42	DLK	Data cache locking. Set when a DSI occurs because **dcbtls**, **dcbtstls**, or **dcblc** is executed in user mode while MSR[UCLE] = 0.	DSI

Table 2-6. ESR/GESR Field Descriptions (continued)

Bits	Name	Syndrome	Interrupt Types
43	ILK	Instruction cache locking. Set when a DSI occurs because **icbtls** or **icblc** is executed in user mode while MSR[UCLE] = 0.	DSI
44	—	Not supported on the e500mc. Defined by the architecture as AP (auxiliary processor operation).	—
45	—	Not supported on the e500mc. Unimplemented operation exception. On the e500mc, unimplemented instructions are handled as illegal instructions.	Program
46	BO	Byte-ordering exception	DSI, ISI
47	PIE	Imprecise exception.	Program
48–56	—	Reserved	—
57	EPID	Indicates whether translation was performed using context from EPLC or EPSC. Set when a DSI, DTLB, or Alignment error occurs during execution of an external PID instruction.	Data storage, Data TLB error Alignment
58–63	—	Reserved	—

2.9.7 (Guest) Processor ID Register (PIR/GPIR)

The e500mc implements the PIR/GPIR as defined by the Power ISA and the processor control architecture as described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. The processor sets the initial value of PIR at reset, after which it is writeable by hypervisor software. The initial value of the PIR is a processor-unique value within the coherence domain and is described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. The initial value of GPIR at reset is 0. Hypervisor software is expected to initialize GPIR to a reasonable value when a partition is initialized.

When executing in the guest supervisor state any **mfspr** accesses to the PIR are mapped to GPIR. The **mtspr** accesses are not mapped, and guest supervisor attempts to change PIR or GPIR cause an embedded hypervisor privilege interrupt. See Section 2.3, "Register Mapping in Guest–Supervisor State," for more details.

2.9.8 Machine Check Address Register (MCAR/MCARU)

When the core takes a machine check interrupt, MCAR indicates the address of the data associated with the machine check exception. MCAR is a 64-bit address and may contain a physical address or an effective address. The MCARU is a 32-bit alias to the upper 32 bits of MCAR. Not all machine check (or error report) interrupts that occur have addresses associated with them. Errors that cause MCAR contents to be updated are implementation-dependent.

MCAR is implemented as defined in the architecture, except as follows: For a certain subset of asynchronous machine check exception causes, MCAR indicates the address of the data or instruction access associated with the machine check. The MCSR[MAV] and MCSR[MEA] status bits indicate whether hardware has updated the MCAR and whether the MCAR contains an effective address or a real address. MCAR is not modified if a machine check occurs and at the time of the interrupt, MCSR[MAV] is already set.

This table shows the MCAR address and MCSR[MAV,MEA] at error time.

Table 2-7. MCAR Address and MCSR[MAV,MEA] at Error Time

MCSR[MAV] State		MCSR[MEA]: Next State	MCAR/MCARU	Comment
Current	Next			
1	x	x	Unaltered	MCAR unmodified if currently valid (hold value if already valid)
0	1	0	MCAR[28–63]	Updated with a real address.
0	1	1	MCAR[0–63]	Updated with the EA associated with the error. If the detected error is a multiway hit in the L2MMU (MCSR[L2MMU_MHIT]), the lower 12 bits of the EA are cleared providing an EPN for the translation.

2.9.9 Machine Check Syndrome Register (MCSR)

In addition to the MCSR fields defined by the architecture, the e500mc implements a number of other implementation-specific fields, as shown in Table 2-8. When the core takes a machine check interrupt, it updates MCSR to differentiate between machine check conditions. The MCSR indicates the type of error detected and software can use this information to determine whether the error is recoverable and what steps may be necessary to correct the error.

MCSR bits are divided into the following categories:

- Async bits. Set asynchronously whenever an error event occurs. Any unit that detects an error automatically posts the error by setting one of these bits. If machine check interrupts are enabled (MSR[ME] = 1 or MSR[GS] = 1), a machine check interrupt occurs when any of these bits in the MCSR is non-zero.
- Error report bits. Set when a synchronous error report type of machine check occurs.
- MCAR status bits (MAV, MEA). These give information about the MCAR.

MCSR is shown in this figure.

SPR 572 — Hypervisor, Write 1 to Clear

	32	33	34	35	36	37
R	MCP	ICERR (ICPERR)	DCERR (ICPERR)	TLBPERR	L2MMU_MHIT	—
W	w1c	w1c	w1c	w1c	w1c	
Reset	All zeros					

	40	42	43	44	45	46	47
R	—		NMI	MAV	MEA	—	IF
W			w1c	w1c	w1c		w1c
Reset	All zeros						

	48	49	50	51	55
R	LD	ST	LDG	—	
W	w1c	w1c	w1c		
Reset	All zeros				

	56	61	62	63
R	—		TLBSYNC	BSL2_ERR
W			w1c	w1c
Reset	All zeros			

Figure 2-7. Machine Check Syndrome Register (MCSR)

This table describes the MCSR fields.

Table 2-8. Machine Check Syndrome Register (MCSR)

Bit	Name	Description	Exception Type[1]	Additional Gating Condition[2]
32	MCP	Machine check input signal asserted. Set immediately on recognition of assertion of the MCP input. This input comes from the SoC and is a level sensitive signal. This usually occurs as the result of an error detected by the SoC.	Async	HID0[EMCP]
33	ICERR (ICPERR)	Instruction cache tag or data array parity error	Async	L1CSR1[ICECE] and L1CSR1[ICE]
34	DCERR (DCPERR)	Uncorrectable L1 data cache data or tag error.	Async	L1CSR0[CECE] and L1CSR0[CE]
35	—	Reserved	—	—
36	L2MMU_MHIT	L2 MMU simultaneous hit.	Async	HID0[EN_L2MMU_MHD]
37–42	—	Reserved	—	—
43	NMI	Nonmaskable interrupt.	NMI	None

Table 2-8. Machine Check Syndrome Register (MCSR) (continued)

Bit	Name	Description	Exception Type[1]	Additional Gating Condition[2]
44	MAV	MCAR address valid. The address contained in the MCAR was updated by the processor and corresponds to the first detected error condition that contained an associated address. Subsequent machine check errors that have associated addresses are not placed in MCAR unless MAV is 0 at the time the error is logged. 0 The address contained in MCAR is not valid. 1 The address contained in MCAR is valid. **Note:** Software should first read MCAR before clearing MAV. MAV should be cleared before MSR[ME] is set.	Status	—
45	MEA	MCAR effective address. Meaningful only if MAV=1. 0 The MCAR contains a physical (real) address. 1 The MCAR contains an EA.	Status	—
46	—	Reserved	—	—
47	IF	Instruction fetch error report. An error occurred during the attempt to fetch the instruction corresponding to the address in MCSRR0 or during an attempted fetch of a younger instruction than that pointed by MCSRR0.	Error report	None
48	LD	Load instruction error report. An error occurred during the attempt to execute the load instruction at the address contained in MCSRR0.	Error report	None
49	ST	Store instruction error report. An error occurred during an attempt to translate the address of the store type instruction (or instruction that is processed by the store queue) located at the address in MCSRR0.	Error report	None
50	LDG	Guarded load instruction error report. Set along with LD if the load encountering the error was a guarded load (WIMGE = *xxx*1*x*) and that guarded load did not encounter one of the data cache errors. Set only if the error encountered by the load was an L2 or CoreNet error.	Error report	None
51	—	Reserved	—	—
52–61	—	Reserved	—	—
62	TLBSYNC	Simultaneous **tlbsync** operations detected. The system should never have two outstanding **tlbsync** operations on CoreNet.	Async	None
63	BSL2_ERR	L2 cache error	Async	L2CSR0[L2E] and L2ERRDIS [3]

[1] "Exception Type" indicates which exception type caused the update of a given MCSR bit:
- Error report. Indicates that this bit is set only for error report exceptions that cause machine check interrupts. These bits are only updated when the machine check interrupt is taken. Error report exceptions are not gated by MSR[ME]. These are synchronous exceptions.
- NMI. Indicates that this bit is only set for the nonmaskable interrupt type exceptions which cause machine check interrupts. This bit is only updated when the machine check interrupt is taken. NMI exceptions are not gated by MSR[ME]. This is an asynchronous exception.
- Async. Indicates that this bit is set for an asynchronous machine check exception. These bits are set immediately upon detection of the error in the MCSR. Once bit is set in the MCSR, a machine check interrupt occurs if MSR[ME]=1. If MSR[ME]=0, the MCSR bits remain set unless cleared by software, and a machine check occurs when MSR[ME] is set.
- "Status" indicates that this bit provides additional status about the logging of an asynchronous machine check exception.

[2] "Additional Gating Condition" indicates any other state that, if not enabled, inhibits the recognition this particular error condition.

[3] For description of L2ERRDIS, see Section 2.15.4, "L2 Error Registers."

The setting of MCSR[LD] and MCSR[ST] identifying the type of instruction is implementation dependent. For e500mc, LD is set by instructions that load data into a register and complete when the load data is committed to the architected register. ST is set by instructions that perform store operations and instructions that are processed through the store queue in the LSU. The treatment of an instruction as a load or store for the purposes of permission checking and debug events may differ from whether the LD or ST bit is set for an error report.

The following instructions set MCSR[LD] if an error report occurs:

dcbt, **dcbtst**, **icbt**, **lbz**, **lbzu**, **lbzx**, **lbzux**, **lha**, **lhau**, **lhax**, **lhaux**, **lhz**, **lhzu**, **lhzx**, **lhzux**, **lhbrx**, **lmw**, **lwarx**, **lwz**, **lwzu**, **lwzx**, **lwzux**, **lwbrx**, **lbepx**, **lhepx**, **lwepx**, **dcbtep**, **dcbtstep**, **lbdx**, **lhdx**, **lwdx**, **lfddx**, **lfd**, **lfdu**, **lfdux**, **lfdx**, **lfdepx**, **lfs**, **lfsu**, **lfsux**, **lfsx**

The following instructions set MCSR[ST] if an error report occurs:

dcba, **dcbal**, **dcbf**, **dcbi**, **dcblc**, **dcbst**, **dcbtls**, **dcbtstls**, **dcbz**, **dcbzl**, **dsn**, **icbi**, **icblc**, **icbtls**, **stb**, **stbu**, **stbx**, **stbux**, **sth**, **sthu**, **sthx**, **sthux**, **sthbrx**, **stmw**, **stw**, **stwu**, **stwx**, **stwux**, **stwbrx**, **stwcx.**, **stbepx**, **sthepx**, **stwepx**, **dcbfep**, **dcbstep**, **icbiep**, **dcbzep**, **dcbzlep**, **stbdx**, **sthdx**, **stwdx**, **stfddx**, **stfd**, **stfdu**, **stfdux**, **stfdx**, **stfdepx**, **stfiwx**, **stfs**, **stfsu**, **stfsux**, **stfs**

2.10 Software-Use SPRs (SPRGs, GSPRGs, and USPRG0)

The e500mc implements the software-use SPRs (SPRG0–SPRG7, SPRG8, SPRG9, GSPRG0-GSPRG3, USPRG0) as defined by the architecture. Their functionality is defined by the user and they are accessed as shown in this table.

Table 2-9. SPRGs, GSPRGs, and USPRG0

Abbreviation	Name	SPR Number	Access
GSPRG0	Guest SPR general 0	368	Guest supervisor
GSPRG1	Guest SPR general 1	369	Guest supervisor
GSPRG2	Guest SPR general 2	370	Guest supervisor
GSPRG3	Guest SPR general 3	371	Guest supervisor
SPRG0	SPR general 0	272	Guest supervisor[1]
SPRG1	SPR general 1	273	Guest supervisor[1]

Table 2-9. SPRGs, GSPRGs, and USPRG0 (continued)

Abbreviation	Name	SPR Number	Access
SPRG2	SPR general 2	274	Guest supervisor[1]
SPRG3	SPR general 3	259	User RO
SPRG3	SPR general 3	275	Guest supervisor[1]
SPRG4	SPR general 4	260	User RO
SPRG4	SPR general 4	276	Guest supervisor
SPRG5	SPR general 5	261	User RO
SPRG5	SPR general 5	277	Guest supervisor
SPRG6	SPR general 6	262	User RO
SPRG6	SPR general 6	278	Guest supervisor
SPRG7	SPR general 7	263	User RO
SPRG7	SPR general 7	279	Guest supervisor
SPRG8	SPRG8	604	Hypervisor
SPRG9	SPRG9	605	Guest supervisor
USPRG0 (VRSAVE)	User SPR general 0[2]	256	User

1 When these registers are accessed in guest supervisor state, the access are mapped to their analogous guest SPRs (for example, SPRG0 is mapped to GSPRG0). See Section 2.3, “Register Mapping in Guest–Supervisor State.”

2 USPRG0 is a separate physical register from SPRG0.

NOTE

Operating system software should always use SPRG0, SPRG1, SPRG2, SPRG3 when accessing GSPRG0, GSPRG1, GSPRG2, and GSPRG3 because in guest–supervisor state, these accesses are mapped to their equivalent guest registers. This allows the programming model for the operating system software to be the same regardless of whether the operating system is operating in guest state under a hypervisor or is executing directly on the bare metal.
SPRGs and GSPRGs are 32 bits for 32-bit implementations and 64 bits for 64-bit implementations. For e500mc, these registers are 32 bits. USPRG0 (VRSAVE) is a 32-bit register regardless of whether the processor is a 32-bit or 64-bit implementation.

2.11 Branch Unit Control and Status Register (BUCSR)

The BUCSR, shown in Figure 2-8, is an e500mc-specific register used for general control and status of the branch prediction mechanisms which include the branch target buffer (BTB). Writing to BUCSR requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

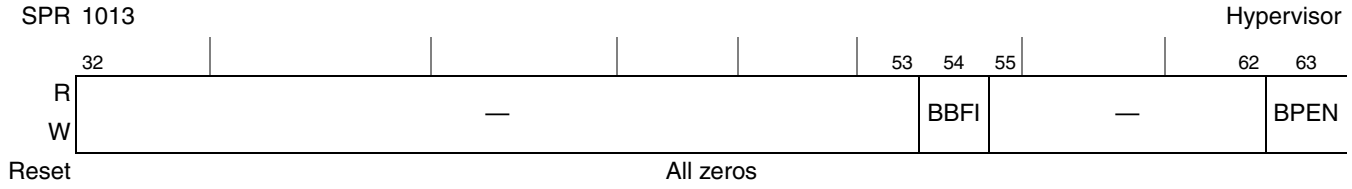


Figure 2-8. Branch Unit Control and Status Register (BUCSR)

This table describes the BUCSR fields.

Table 2-10. BUCSR Field Descriptions

Bits	Name	Description
32–53	—	Reserved
54	BBFI	Branch buffer flash invalidate. Setting BBFI flash clears the valid bit of all entries in the branch prediction mechanisms; clearing occurs independently from the value of the enable bit (BPEN). BBFI is cleared by hardware and always reads as 0.
55–62	—	Reserved
63	BPEN	Branch prediction enable 0 Branch prediction disabled 1 Branch prediction enabled (enables BTB to predict branches)

2.12 Hardware Implementation-Dependent Register 0 (HID0)

This section describes HID0, shown in Figure 2-9, as it is implemented on the e500mc core.

NOTE

Some HID fields may not be implemented in a device that incorporates the e500mc core and some fields may be defined more specifically by the incorporating device. For specific details it is important to refer to the "Register Summary" chapter in the device's reference manual.

HID0 is used for configuration and control. Writing to HID0 requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

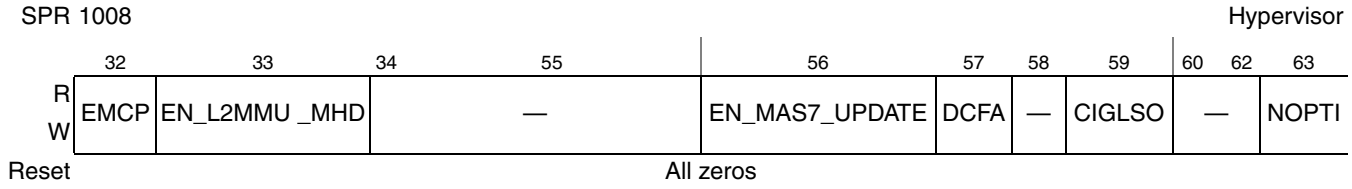


Figure 2-9. Hardware Implementation-Dependent Register 0 (HID0)

This table describes the HID0 fields.

Table 2-11. HID0 Field Descriptions

Bits	Name	Description
32	EMCP	Enable machine check signal. Used to mask out further machine check exceptions caused by asserting the internal machine check signal from the integrated device. 0 Machine check signalling is disabled. 1 Machine check signalling is enabled. If HID0[EMCP] = 1, asserting the machine check signal from the integrated device causes MCSR[MCP] to be set to 1. If MSR[ME] = 1 or MSR[GS] = 1, a machine check exception and subsequent interrupt occurs.
33	EN_L2MMU_MHD	Enable L2MMU multiple-hit detection. An L2MMU multiple hit occurs when more than one entry matches a given translation. This most likely occurs when software mistakenly loads the TLB with more than one entry that matches the same translation, but can also occur if a soft error occurs in a TLB entry. 0 Machine check signalling is disabled. 1 A multiple L2 MMU hit sets MCSR[L2MMU_MHIT] to 1. If MSR[ME] = 1 or MSR[GS] = 1, a machine check exception and subsequent interrupt occurs.
34–55	—	Reserved
56	EN_MAS7_UPDATE	Enable MAS7 update. Enables updating MAS7 by **tlbre** and **tlbsx**. 0 MAS7 is not updated by a **tlbre** or **tlbsx**. 1 MAS7 is updated by a **tlbre** or **tlbsx**.
57	DCFA	Data cache flush assist. Force data cache to ignore invalid sets on miss replacement selection. 0 The data cache flush assist facility is disabled 1 The miss replacement algorithm ignores invalid entries and follows the replacement sequence defined by the PLRU bits. This reduces the series of uniquely addressed load or **dcbz** instructions to eight per set. The bit should be set just before beginning a cache flush routine and should be cleared when the series of instructions is complete.
58	—	Reserved
59	CIGLSO	Cache-inhibited guarded load/store ordering. 0 Loads and stores to storage that are marked as cache inhibited and guarded have no ordering implied except what is defined in the rest of the architecture. 1 Loads and stores to storage that are marked as cache inhibited and guarded are ordered.
60–62	—	Reserved
63	NOPTI	NOP the data and instruction cache touch instructions. Note that “cache and lock set” and “cache and lock clear” instructions are not affected by the setting of this bit. 0 **dcbt**, **dcbtep**, **dcbtst**, **dcbtstep**, and **icbt** are enabled, and operate as defined by the architecture and the rest of this document. 1 **dcbt**, **dcbtep**, **dcbtst**, **dcbtstep**, and **icbt** are treated as NOPs. When touch instructions are treated as NOPs because HID0[NOPTI] is set, they do not cause DAC debug events. That is, if a DAC comparison would have caused a debug event, the debug event is also NOPed and does not occur.

2.13 Core Device Control and Status Register (CDCSR0)

The core device control and status register is implemented as described in *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors*. The e500mc core is aware of the following device programming models:

- the Floating Point Device. The device is present and ready.

For e500mc, writes to CDCSR0 are ignored.

This figure shows the core device control and status register 0 format.

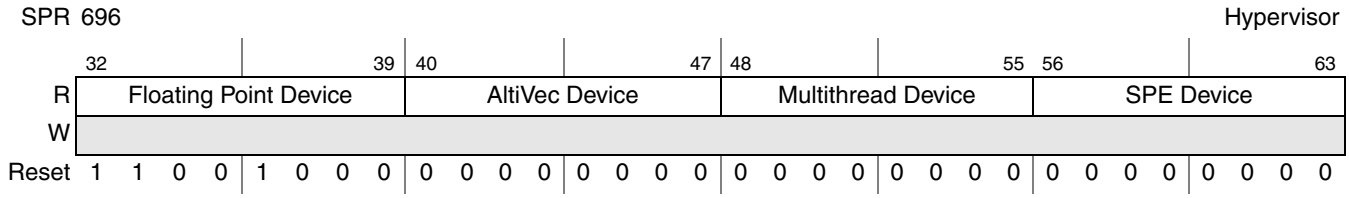


Figure 2-10. Core Device Control and Status Register 0 (CDCSR0) Format

2.14 L1 Cache Registers

The L1 cache registers provide control and configuration and status information for the L1 cache implementation.

2.14.1 L1 Cache Control and Status Register 0 (L1CSR0)

L1CSR0 is used for general control and status of the L1 data cache. The e500mc implements the L1CSR0 fields shown in Figure 2-11 as they are defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. It does not implement the following:

- Cache way partitioning bits (L1CSR0[32–42]).
- Data cache lock overflow allocate bit, CLOA, (L1CSR0[56]).
- Cache operation aborted bit, CABT (L1CSR0[61]). Cache operations are never aborted on e500mc.

For L1CSR0[CEA], e500mc only supports the value 0b00 and always invalidates the entire contents (tags and data arrays) and generates a machine check or error report on the occurrence of an error detection when L1CSR0[CECE] is set. Writing any other value to this field is ignored.

For L1CSR0[CEDT], e500mc only supports the value 0b00 and detects single-bit tag and single-bit data errors. Writing any other value to this field is ignored.

Note that on the e500mc, setting L1CSR2[DCWS] automatically sets L1CSR0[CFI]. Also, when setting L1CSR0[CEI], it is required that L1CSR0[CECE] also be set in the same **mtspr** instruction.

Writing to L1CSR0 requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

SPR 1010 Hypervisor

	32–35	36–39	40	41	42	43	44–45	46	47
R W	—	—	—	—	—	—	CEA	—	CECE
Reset	All zeros								

	48	49	50–51	52	53	54	55	56	57–58	59	60	61	62	63
R W	CEI	—	CEDT	CSLC	CUL	CLO	CLFC	—	CEIT	—	DCBZ32	—	CFI	CE
Reset	All zeros													

Figure 2-11. L1 Cache Control and Status Register 0 (L1CSR0) Fields Implemented on e500mc

Table 2-12. L1CSR0 Field Descriptions

Bits	Name	Description
32–43	—	Reserved
44–45	CEA	Data cache error action 00 Error detection causes a machine check interrupt (and possibly error report interrupts). For e500mc, if the core is in write shadow mode (L1CSR2[DCWS] = 1), the entire data cache is invalidated. 01 Reserved for e500mc. 10 Reserved for e500mc. 11 Reserved The setting of CEA has no effect if L1CSR0[CECE] = 0. Reading CEA is not guaranteed to reflect the last written value in some implementations, however, it returns either the last written value or 0. e500mc only supports the value 0b00 for ICEA
46	—	Reserved
47	CECE CPE DCPE	(Data) Cache error checking enable. 0 Error detection of the cache disabled 1 Error detection of the cache enabled
48	CEI CPI DCPI	(Data) Cache error injection enable. See Section 5.4.5, “Cache Error Injection.” 0 Error error injection disabled 1 Error injection enabled. Cache error checking must also be enabled (CECE = 1) when this bit is set. Note that if the programmer attempts to set L1CSR0[CEI] (using **mtspr**) without setting L1CSR0[CECE], L1CSR0[CEI] is not set (enforced by hardware).
49–51	—	Reserved
50–51	CEDT	Data cache error detection/correction type 00 Detect tag parity errors and data parity errors 01 reserved for e500mc 10 reserved 11 reserved

Table 2-12. L1CSR0 Field Descriptions (continued)

Bits	Name	Description
52	CSLC DCSLC	(Data) Cache snoop lock clear. Sticky bit set by hardware if a cache line lock was cleared by a snoop operation which caused an invalidation. Note that the lock for that line is cleared whenever the line is invalidated. This bit can be cleared only by software. 0 The cache has not encountered a snoop that invalidated a locked line. 1 The cache has encountered a snoop that invalidated a locked line.
53	CUL DCUL	(Data) Cache unable to lock. Sticky bit set by hardware. This bit can be cleared only by software. 0 Indicates a lock set instruction was effective in the cache 1 Indicates a lock set instruction was not effective in the cache
54	CLO DCLO	(Data) Cache lock overflow. <E.CL> Sticky bit set by hardware. This bit can be cleared only by software. 0 Indicates a lock overflow condition was not encountered in the cache 1 Indicates a lock overflow condition was encountered in the cache
55	CLFC DCLFC	(Data) Cache lock bits flash clear. <E.CL> Clearing occurs regardless of the enable (L1CSR0[CE]) value. 0 Default. 1 Hardware initiates a cache lock bits flash clear operation. Cleared when the operation is complete. **Note:** During a flash clear operation, writing a 1 causes undefined results; writing a 0 has no effect
56	—	Reserved
57–58	CEIT	Cache error injection type. Controls the type of error injection to be performed. 00 Inject single-bit data error and inject single bit tag error 01 reserved 10 reserved 11 reserved
60	DCBZ32	Data cache operation length. 0 **dcba** and **dcbz** (**dcbzep**) instruction number of bytes operated on is all bytes in cache line 1 **dcba** and **dcbz** (**dcbzep**) number of bytes operated on is 32
61	—	Reserved
62	CFI DCFI	(Data) Cache flash invalidate. Invalidation occurs regardless of the enable (L1CSR0[CE]) value. 0 No cache invalidate. 1 Cache flash invalidate operation. A cache invalidation operation is initiated by hardware. Once complete, this bit is cleared. **Note:** During an invalidation operation, writing a 1 causes undefined results; writing a 0 has no effect.
63	CE DCE	(Data) Cache enable. 0 The cache is not enabled. (not accessed or updated) 1 Enables cache operation. **Note:** CE should not be set when the cache is disabled until after the cache has been properly initialized by flash invalidating the cache . This applies both to the first time the cache is enabled as well as sequences that want to re-enable the cache after software has disabled it. **Note:** If the cache is enabled and software wishes to disable it by writing a 0 to CE, software should first flush the cache to ensure that any modified data resident in the cache is pushed to memory. If the cache is not flushed, coherency is lost and any lines in the cache may provide stale data when the cache is re-enabled.

2.14.2 L1 Cache Control and Status Register 1 (L1CSR1)

L1CSR1 is used for general control and status of the L1 instruction cache. The e500mc implements the L1CSR1 fields shown in Figure 2-12 as they are defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. It does not implement L1CSR1[ICLOA,ICABT] (bits 56 and 61).

For L1CSR1[ICEA], e500mc only supports the value 0b00 and always invalidates the entire contents (tags and data arrays) and generates a machine check or error report on the occurrence of a parity error when L1CSR1[ICECE] is set. Writing any other value to this field is ignored.

For L1CSR1[ICEDT], e500mc only supports the value 0b00 for parity detection. Writing any other value to this field is ignored.

When setting L1CSR1[ICEI], it is required that L1CSR1[ICECE] also be set in the same **mtspr** instruction.

Writing to L1CSR1 requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

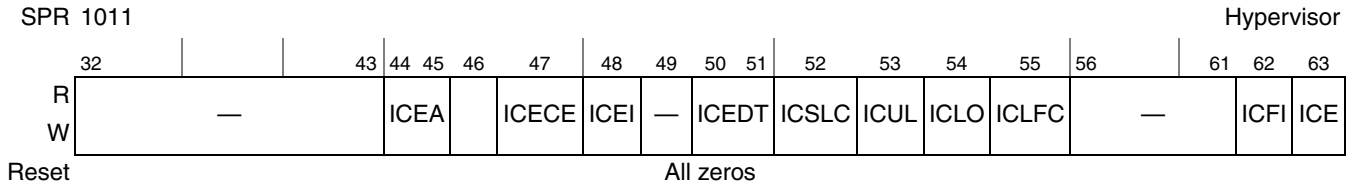


Figure 2-12. L1 Cache Control and Status Register 1 (L1CSR1) Fields Implemented on the e500mc

This table describes L1CSR1 fields implemented on the e500mc.

Table 2-13. L1CSR1 Field Descriptions

Bits	Name	Description
32–43	—	Reserved
44–45	ICEA	Instruction cache error action 00 Error detection causes a machine check (and possibly an error report). The location in the instruction cache which caused the error is invalidated (other instruction cache locations may also be invalidated).e500mc 01 Reserved for e500mc 10 Reserved for e500mc 11 Reserved The setting of ICEA has no effect if L1CSR1[ICECE] = 0. Reading ICEA is not guaranteed to reflect the last written value in some implementations, however, it returns either the last written value or 0. **Note:** e500mc only supports the value 0b00 for ICEA.
47	ICECE	Instruction error checking enable. 0 Error checking of the cache disabled 1 Error checking of the cache enabled
48	ICEI	Instruction cache error injection enable. 0 Error injection disabled 1 Error injection enabled. Note that cache error checking must also be enabled (L1CSR1[ICECE] = 1) when this bit is set, otherwise, results are undefined and erratic behavior may occur. If L1CSR0[ICECE] = 0, ICEI cannot be set (i.e,). L1CSR0[ICEI] = L1CSR0[ICECE] & L1CSR0[ICEI]).

Table 2-13. L1CSR1 Field Descriptions (continued)

Bits	Name	Description
49	—	Reserved
50–51	ICEDT	Instruction cache error detection type. 00 Parity detection. 01 Reserved for e500mc 10 Reserved 11 Reserved The setting of ICEDT has no effect if L1CSR1[ICECE] = 0. Reading ICEDT is not guaranteed to reflect the last written value in some implementations, however, it returns either the last written value or 0. **Note:** e500mc only supports the value 0b00 for ICEDT.
52	ICSLC	Instruction cache snoop lock clear. Sticky bit set by hardware if a cache line lock was cleared by a snoop operation which caused an invalidation. Note that the lock for that line is cleared whenever the line is invalidated. This bit can be cleared only by software. 0 The cache has not encountered a snoop that invalidated a locked line. 1 The cache has encountered a snoop that invalidated a locked line.
53	ICUL	Instruction cache unable to lock. Sticky bit set by hardware. This bit can be cleared only by software. 0 Indicates a lock set instruction was effective in the cache 1 Indicates a lock set instruction was not effective in the cache
54	ICLO	Instruction cache lock overflow. Sticky bit set by hardware. This bit can be cleared only by software. 0 Indicates a lock overflow condition was not encountered in the cache 1 Indicates a lock overflow condition was encountered in the cache
55	ICLFC	Instruction cache lock bits flash clear. Clearing occurs regardless of the enable (L1CSR1[ICE]) value. 0 Default. 1 Hardware initiates a cache lock bits flash clear operation. This bit is cleared when the operation is complete. **Note:** Writing a 1 while a flash clear operation is in progress causes undefined results. Writing a 0 during a flash clear operation is ignored
56–61	—	Reserved for implementation dependent use.
62	ICFI	Instruction cache flash invalidate. Invalidation occurs regardless of the enable ($L1CSR1_{ICE}$) value. 0 No cache invalidate. 1 Cache flash invalidate operation. A cache invalidation operation is initiated by hardware. Once complete, this bit is cleared. **Note:** Writing a 1 during an invalidation operation causes undefined results. Writing a 0 during an invalidation operation is ignored.
63	ICE	Instruction cache enable. 0 The cache is not enabled. (not accessed or updated) 1 Enables cache operation. **Note:** ICE should not be set when the cache is disabled until after the cache has been properly initialized by flash invalidating the cache . This applies both to the first time the cache is enabled as well as sequences that want to re-enable the cache after software has disabled it.

2.14.3 L1 Cache Control and Status Register 2 (L1CSR2)

L1CSR2, shown in Figure 2-13, provides additional control and status for the primary L1 data cache of the processor. The e500mc implements L1CSR2 as it is defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*, with the following exceptions:

- Setting L1CSR2[DCWS] automatically sets L1CSR0[CFI] to flash invalidate the data cache when turning on write shadow mode to purge the cache of any modified data. Software should perform a flush operation on the data cache prior to setting L1CSR2[DCWS].
- While write shadow mode is active (L1CSR2[DCWS] = 1), the L2 cache is required to be enabled and in general be able to allocate lines when store or store type operations are performed. See Table 5-1 for supported write shadow configurations.
- Although the architecture defines DCSTASHID as L1CSR2[54–63], the e500mc implements only 8 bits (L1CSR2[56–63]) and supports only stash ID values of 8 to 255.

Writing to L1CSR2 requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

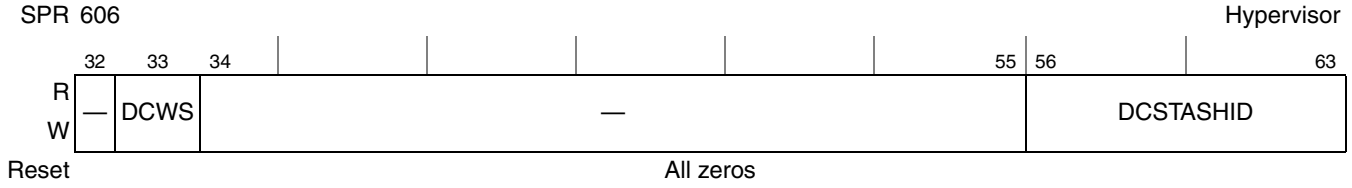


Figure 2-13. L1 Cache Control and Status Register 2 (L1CSR2) Fields Implemented on the e500mc

This table describes how L1CSR2 fields are implemented on the e500mc.

Table 2-14. L1CSR2 Field Descriptions

Bits	Name	Description
32	—	Implementation dependent.
33	DCWS	Data cache write shadow. Note that on the e500mc, changing L1CSR2[DCWS] automatically sets L1CSR0[CFI]. Set by software to place the primary data cache into write shadow mode. When write shadow mode is enabled, data that is written to the primary data cache is also written through to the backside L2 (or other parts of the memory hierarchy) so that any subsequent failures in the primary data cache can be recovered from by invalidating the data cache. 0 The primary data cache is not in write shadow mode. 1 The primary data cache is in write shadow mode. **Note:** Software should flush and invalidate the primary data cache before setting DCWS to ensure that no modified data exists in the primary data cache. **Note:** Only certain cache configurations are supported when write shadow mode is enabled. See Table 5-1.
34–55	—	Reserved
56–63	DCSTASHID	Data cache stash ID. Contains the cache target identifier for external stash operations directed to this processor's data cache. Clearing DCSTASHID prevents the primary cache from accepting external stash operations. Note that the e500mc supports only stash ID values of 8 and larger (that is values between 8 and 255); values from 1 to 7 are illegal.

2.14.4 L1 Cache Configuration Register 0 (L1CFG0)

L1CFG0, shown in this figure, provides configuration information for the L1 data cache.

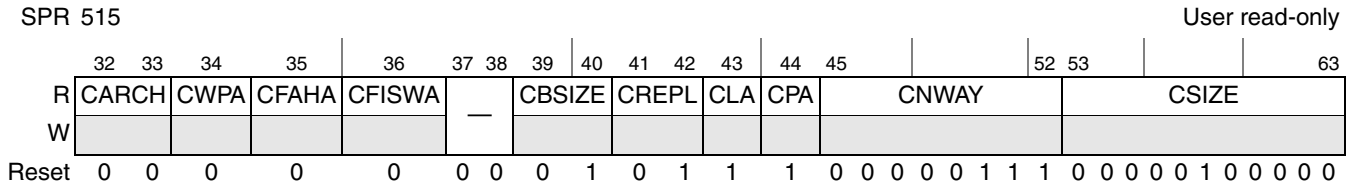


Figure 2-14. L1 Cache Configuration Register 0 (L1CFG0) Fields Implemented on the e500mc

The *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* describes these fields as they are defined in the Power ISA. This table describes how they are implemented on the e500mc.

Table 2-15. L1CFG0 Field Descriptions

Bits	Name	Description
32–33	CARCH	Cache architecture. 0 indicates harvard (split instruction and data)
34	CWPA	Cache way partitioning available. 0 indicates unavailable
35	CFAHA	Cache flush all by hardware available 0 indicates unavailable
36	CFISWA	Direct cache flush available. 0 indicates unavailable
37–38	—	Reserved
39–40	CBSIZE	Cache block size. 1 indicates 64 bytes
41–42	CREPL	Cache replacement policy 1 indicates psuedo-LRU policy
43	CLA	Cache locking available 1 indicates available
44	CPA	Cache parity available. 1 indicates available
45–52	CNWAY	Cache number of ways. 7 indicates 8 ways
53–63	CSIZE	Cache size. 32 indicates 32 Kbytes.

2.14.5 L1 Cache Configuration Register 1 (L1CFG1)

L1CFG1, shown in this figure, provides configuration information for the L1 instruction cache.

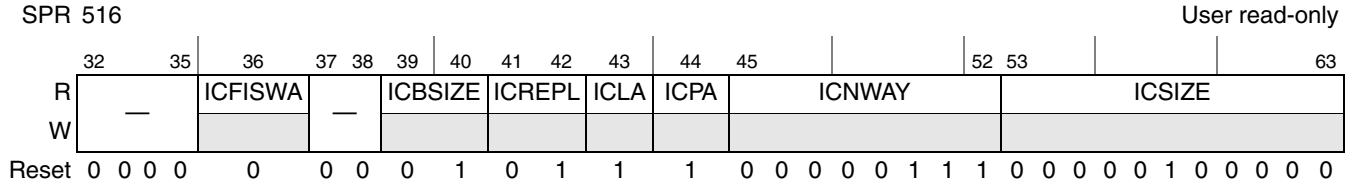


Figure 2-15. L1 Cache Configuration Register 1 (L1CFG1)

This table describes the L1CFG1 fields.

Table 2-16. L1CFG1 Field Descriptions

Bits	Name	Description
32–35	—	Reserved
36	ICFISWA	Direct cache flush available. 0 indicates unavailable
37–38	—	Reserved
39–40	ICBSIZE	Instruction cache block size. 1 indicates 64 bytes
41–42	ICREPL	Instruction cache replacement policy. 1 indicates pseudo-LRU policy
43	ICLA	Instruction cache locking available. 1 indicates available
44	ICPA	Instruction cache parity available. 1 indicates available
45–52	ICNWAY	Instruction cache number of ways. 7 indicates eight ways
53–63	ICSIZE	Instruction cache size. 32 indicates 32 Kbytes

2.15 L2 Cache Registers

L2 cache status, control, and error handling is accomplished through SPRs.

2.15.1 L2 Configuration Register (L2CFG0)

L2CFG0 is provided for software to determine the organization and capabilities of the secondary cache The e500mc implements L2CFG0 as it is defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

L2CFG0, shown in Figure 2-15, provides configuration information for the L2 instruction cache.

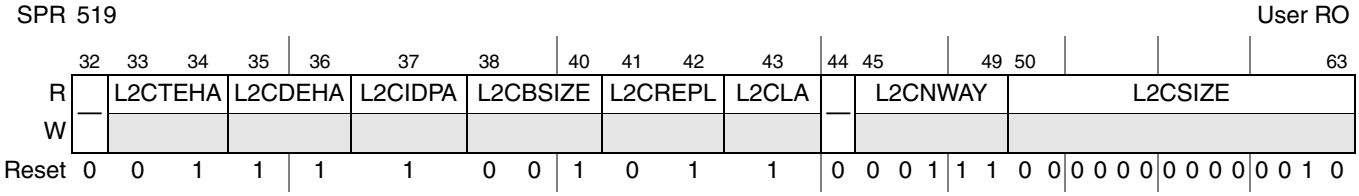


Figure 2-16. L2 Cache Configuration Register 0 (L2CFG0)

This table describes the L2CFG0 settings for the e500mc.

Table 2-17. L2CFG0 Field Descriptions

Bits	Name	Description
32	—	Reserved
33–34	L2CTEHA	L2 cache tags error handling available. 1 indicates parity detection.
35–36	L2CDEHA	L2 cache data error handling available. 0b11 indicates both parity and ECC correction available.
37	L2CIDPA	Cache instruction and data partitioning available. 1 indicates available.
38–40	L2CBSIZE	Cache line size. 1 indicates 64 bytes

Table 2-17. L2CFG0 Field Descriptions (continued)

Bits	Name	Description
41–42	L2CREPL	Cache default replacement policy. This is the default line replacement policy at power-on-reset. If an implementation allows software to change the replacement policy it is not reflected here.1 indicates pseudo-LRU.
43	L2CLA	Cache line locking available. 1 indicates available.
44	—	Reserved
45–49	L2CNWAY	Number of cache ways. 7 indicates 8 ways.
50–63	L2CSIZE	Cache size as a multiple of 64 Kbytes. 2 indicates 128-Kbyte cache.

2.15.2 L2 Cache Control and Status Register (L2CSR0)

L2CSR0, shown in this figure, provides general control and status for the processor's L2 cache.

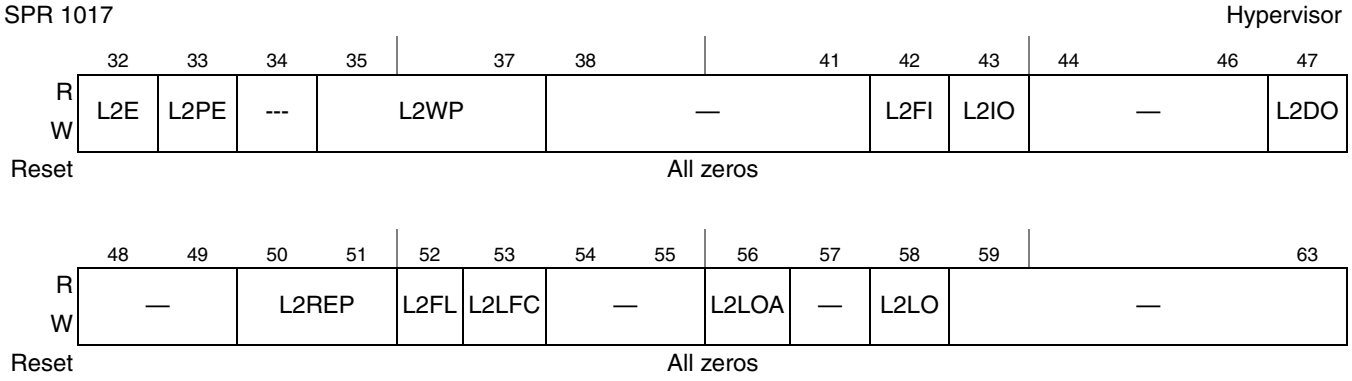


Figure 2-17. L2 Cache Control and Status Register (L2CSR0)

This table describes the L2CSR0 fields.

Table 2-18. L2CSR0 Field Descriptions

Bits	Name	Description
32	L2E	L2 cache enable. Implemented as defined in *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. The e500mc requires software to continue to read this bit after setting it to ensure the desired value has been set before continuing on. **Note:** L2E should not be set when the L2 cache is disabled until after the L2 cache has been properly initialized by flash invalidating the cache and locks. This applies both to the first time the L2 cache is enabled as well as sequences that want to re-enable the cache after software has disabled it. **Note:** If the L2 cache is enabled and software wishes to disable it by writing a 0 to L2E, software should first flush the L2 cache to ensure that any modified data resident in the L2 cache is pushed to memory. If the L2 cache is not flushed, coherency is lost and any lines in the cache may provide stale data when the L2 cache is re-enabled.
33	L2PE	L2 cache parity/ECC error checking enable. Implemented as defined in *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. **Note:** L2PE should not bet set until after the L2 cache has been properly initialized out of reset by flash invalidation. Doing so can cause erroneous detection of errors because the state of the error detection bits are random out of reset. See Section 11.5, "L1 Cache State," for more details on L1 cache initialization. **Note:** When error injection is being performed, the value of L2PE and individual error disables are ignored and errors are always detected. Software should ensure that L2PE is set when performing error injection.
34	—	Reserved

Table 2-18. L2CSR0 Field Descriptions (continued)

Bits	Name	Description
35-37	L2WP	L2 Instruction/Data Way Partitioning If L2IO and L2DO are both 0, the ways of the cache are partitioned to allocate new lines in ways based on whether the allocation is for instructions or data. A value of 0 allows all ways to be used for either instructions or data. A non-zero value specifies the number of ways to be used for allocating instructions. The number of ways specified for data references is the total number of ways minus the value in the L2WP field. 000 All ways are available for instruction allocation and data allocation 001 1 way available for instruction allocation, 7 ways available for data allocation 010 2 ways available for instruction allocation, 6 ways available for data allocation 011 3 ways available for instruction allocation, 5 ways available for data allocation 100 4 ways available for instruction allocation, 4 ways available for data allocation 101 5 ways available for instruction allocation, 3 ways available for data allocation 110 6 ways available for instruction allocation, 2 ways available for data allocation 111 7 ways available for instruction allocation, 1 ways available for data allocation **Performance note:** If the number of ways available for instruction or data allocation is not a power of two, the statistical percentage of total allocations across those available ways over a very long period of time are not evenly distributed. For instance, if 3-ways (say way A, way B, and way C) are available for data allocation, the long term percentage of allocations for A, B, and C are not 33%, 33%, 33%, respectively. Instead, the number of allocations for one of the three ways are closer to 50%, with the other two ways being closer to 25% (50%, 25%, 25%). Instruction and Data way partitioning has no effect on cache locking. Cache lines which are locked due to cache locking instructions are still honored in the presence of way partitioning. If locked lines exist in the L2 cache prior to enabling L2 way partitioning, those locked lines can exist in the “opposite” partition. For instance, a line locked by an **icbtls** instruction can exist in a way which is part of the data partition. To avoid this condition, locks must be flash invalidated prior to enabling way partitioning. Because L2WP only controls how new lines are allocated, L2WP can be changed at any time without affecting the functionality of the L2 Cache.
38–39	(L2CM)	L2 cache coherency mode. This field is not implemented in e500mc, and always reads as 0.
40–41	—	Reserved
42	L2FI	L2 cache flash invalidate. Implemented as defined in *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors*. Note that Lock bits are not cleared by a L2 cache flash invalidate. Lock bits should be cleared by software at boot time to ensure that random states of the lock bits for each line do not limit allocation of those lines. See L2CSR0[L2LFC]. **Note:** When a flash invalidation operation is being performed (i.e. L2FI has been set to 1 by software), software should not attempt to write 1 to this field again until after hardware has reset this bit to 0 to indicate that the invalidate operation is complete. Writing a 1 during an invalidation operation causes undefined results. Writing a 0 during an invalidation operation is ignored.

Table 2-18. L2CSR0 Field Descriptions (continued)

Bits	Name	Description
43	L2IO	L2 cache instruction only. Implemented as defined in *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* except that if L2IO is set and L2DO is not set, storage accesses which are data references (i.e. from load/store instructions) are not serviced from the L2 cache even if the cache had previously allocated and still contains lines from data references that were allocated prior to setting L2IO. In addition, when L2IO is set, the L2 cache does not participate in the coherence protocol (that is, it does not respond to snoops) except that it processes instruction cache invalidations (**icbi**) from any processor. When L2IO is set and the L2 cache contains modified data, that data becomes incoherent. To avoid this situation, if software wishes to set L2IO (and not L2DO), it should first set both L2IO and L2DO to prevent further allocations, then flush any modified data from the L2 cache, then clear L2DO. The e500mc requires software to continue to read this bit after setting it to ensure the desired value has been set before continuing on.
44–46	—	Reserved
47	L2DO	L2 cache data only. Implemented as defined in *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. The e500mc requires software to continue to read this bit after setting it to ensure the desired value has been set before continuing on.
48–49	—	Reserved
50-51	L2REP	L2 line replacement algorithm. The Streaming PLRU modes perform a partial update of the PLRU bits when an L2 line is allocated, and a full update on L2 cache hits. Depending on the access pattern, irregularly-accessed transient data is likely to be evicted before regularly-accessed data. 00 SPLRUA (Streaming Pseudo Least Recently Used with Aging). With this algorithm, the pseudo LRU state for a given index is updated to mark a given way most recently used on each L2 cache hit. On L2 cache allocations, the pseudo LRU state is partially updated on most L2 cache allocations and fully updated on every third L2 cache allocations . 01 Invalid 10 SPLRU (Streaming Pseudo Least Recently Used). With this algorithm, the pseudo LRU state for a given index is updated to mark a given way most recently used on each L2 cache hit. On L2 cache allocations, the pseudo LRU state is partially updated to a state between least recently used and most recently used on all L2 cache allocations. 11 PLRU (Pseudo Least Recently Used).With this algorithm, the pseudo LRU state for a given index is updated to mark a given way most recently used on each L2 cache hit and all L2 cache allocations. Locks for cache lines locked with cache locking instructions are never selected for line replacement unless they are explicitly unlocked, regardless of the replacement algorithm.
52	L2FL	L2 cache flush. Implemented as defined in *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. On e500mc, L2FL should not be set when the L2 cache is not currently enabled (L2E should already be 1). If L2FL is set and the L2 cache is not enabled, the flush does not occur and the L2FL bit remains set.
53	L2LFC	L2 cache lock flash clear. On boot, the processor should set this bit to clear any lock state bits which may be randomly set out of reset, prior to enabling the L2 cache.
54–55	—	Reserved
56	L2LOA	L2 cache lock overflow allocate. Implemented as defined in *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. Note that cache line locking in e500mc L2 is persistent.
57	—	Reserved

Table 2-18. L2CSR0 Field Descriptions (continued)

Bits	Name	Description
58	L2LO	L2 cache lock overflow. Implemented as defined in *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.
59–63	—	Reserved

2.15.3 L2 Cache Control and Status Register 1 (L2CSR1)

L2CSR1, shown in Figure 2-18, provides general control and status for the L2 cache of the processor. The e500mc implements L2CSR1 as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors,* with the following exceptions:

- It implements only the 8 lsbs of the L2STASHID (L2CSR1[L2STASHID]
- It does not support stash ID values of less than 8.

In addition it implements the implementation specific fields L2INSTLOSSLIMITDIS, L2INSTLOSSLIMIT, L2SNPWINLIMITDIS, and L2SNPWINLIMIT.

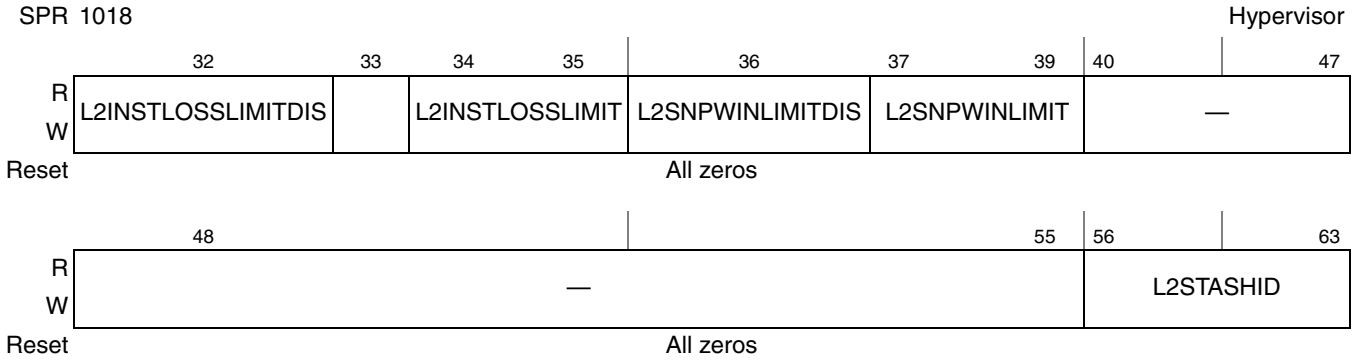


Figure 2-18. L2 Cache Control and Status Register 1 (L2CSR1)

This table describes the L2CSR1 fields.

Table 2-19. L2CSR1 e500mc-Specific Field Descriptions

Bits	Name	Description
32	L2INSTLOSSLIMITDIS	L2 Instruction Loss Limit Disable 0 L2 Instruction Loss Limiting is enabled. 1 L2 Instruction Loss Limiting is disabled.
33	—	Reserved
34–35	L2INSTLOSSLIMIT	Some units of the core can lose arbitration for the backside L2 for multiple cycles. This field specifies how many consecutive cycles instruction fetches can lose backside L2 arbitration before raising its priority. 00 Raise priority after 8 losses (default) 01 Raise priority after 4 losses 10 Raise priority after 8 losses 11 Raise priority after 16 losses

Table 2-19. L2CSR1 e500mc-Specific Field Descriptions (continued)

Bits	Name	Description
36	L2SNPWINLIMITDIS	L2 Snoop Win Limit Disable 0 L2 Snoop Win Limiting is enabled. 1 L2 Snoop Win Limiting is disabled
37–39	L2SNPWINLIMIT	Snoops receive the highest priority when arbitrating for the backside L2. In a system with very active snooping, this can starve other units from winning access to the backside L2. This field specifies how many consecutive snoops can win arbitration before allowing another unit to win. 000 Limit to 8 consecutive snoops (default) 001 Limit to 2 consecutive snoops 010 Limit to 4 consecutive snoops 011 Limit to 8 consecutive snoops 100 Limit to 16 consecutive snoops 101 Limit to 32 consecutive snoops 110 Limit to 64 consecutive snoops 111 Limit to 128 consecutive snoops
40–55	—	Reserved
56–63	L2STASHID	L2 cache stash ID. Contains the cache target identifier to be used for external stash operations directed to this processor's L2 cache. A value of 0 for L2STASHID prevents the L2 cache from accepting external stash operations. Note that the e500mc supports only stash ID values of 8 and larger (that is values between 8 and 255); values from 1 to 7 are illegal.

2.15.4 L2 Error Registers

L2 cache error detection, reporting, and injection allow flexible handling of ECC and parity errors in the L2 data and tag arrays. The e500mc implements the L2 error detection registers as they are defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. Deviations from the architecture are described here.

2.15.4.1 L2 Cache Error Disable Register (L2ERRDIS)

L2ERRDIS, shown in this figure, provides general control for disabling error detection in the L2 cache of the processor. The e500mc implements L2ERRDIS as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* except that it does not implement the TMBECCDIS and TSBECCDIS fields, and implements the implementation specific field TMHITDIS.

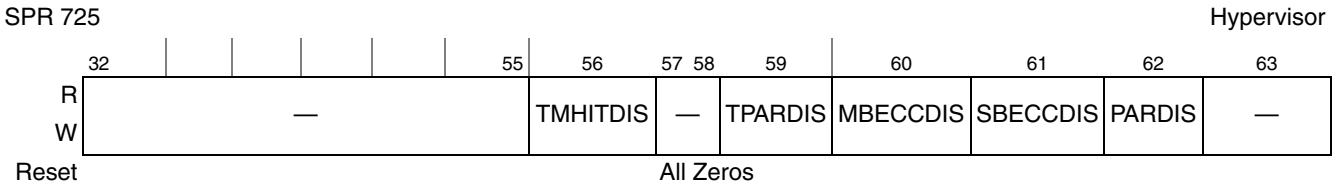


Figure 2-19. L2 Cache Error Disable Register (L2ERRDIS)

This table describes the L2ERRDIS fields.

Table 2-20. L2ERRDIS Field Descriptions

Bits	Name	Description
32–56	—	Reserved
56	TMHITDIS	Tag multi-way hit error disable. 0 Tag multi-way hit detection enabled if L2CSR0[L2PE] = 1. 1 Tag multi-way hit error detection disabled. **Note:** This field is not part of the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. **Note:** When error injection is being performed, the value of TMHITDIS and L2CSR0[L2PE] are ignored and errors are always detected. Software should ensure that L2PE is set and TMHITDIS is clear when performing error injection to the tags.
57-58	—	Reserved
59	TPARDIS	Tag parity error disable. 0 Tag parity error detection enabled. 1 Tag parity error detection disabled. **Note:** When error injection is being performed, the value of TPARDIS and L2CSR0[L2PE] are ignored and errors are always detected. Software should ensure that L2PE is set and TPARDIS is clear when performing error injection to the tags.
60	MBECCDIS	Data Multiple-bit ECC error disable. 0 Data Multiple-bit ECC error detection enabled. 1 Data Multiple-bit ECC error detection disabled. **Note:** When error injection is being performed, the value of MBECCDIS and L2CSR0[L2PE] are ignored and errors are always detected. Software should ensure that L2PE is set and MBECCDIS is clear when performing error injection to the data.
61	SBECCDIS	Data ECC error disable. 0 Data Single-bit ECC error detection enabled. 1 Data Single-bit ECC error detection disabled. **Note:** When error injection is being performed, the value of SBECCDIS and L2CSR0[L2PE] are ignored and errors are always detected. Software should ensure that L2PE is set and SBECCDIS is clear when performing error injection to the data.
62	PARDIS	Data parity error disable. 0 Data parity error detection enabled if L2CSR0[L2PE] = 1, MBECCDIS = 1, and SBECCDIS = 1 1 Data parity error detection disabled. **Note:** When error injection is being performed, the value of PARDIS and L2CSR0[L2PE] are ignored and errors are always detected. Software should ensure that L2PE is set and PARDIS is clear when performing error injection to the data.
63	—	Reserved

2.15.4.2 L2 Cache Error Detect Register (L2ERRDET)

L2ERRDET, shown in Figure 2-20, provides general status and information for errors detected in the L2 cache of the processor. The e500mc implements L2ERRDET as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* except

that it does not implement the TMBECCERR, TSBECCERR, and L2CFGERR fields, and implements the implementation specific fields MULL2ERR and TMHIT.

SPR 991 Hypervisor

	32						55	56	57	58	59		61	62	63
R	MULL2ERR							TMHIT	—		TPARERR	MBECCERR	SBECCERR	PARERR	—
W	w1c							w1c			w1c	w1c	w1c	w1c	
Reset	All Zeros														

Figure 2-20. L2 Cache Error Detect Register (L2ERRDET)

This table describes the L2ERRDET fields.

Table 2-21. L2ERRDET Field Descriptions

Bits	Name	Description
32	MULL2ERR	Multiple L2 errors. Writing a 1 to this bit location resets the bit. 0 Multiple L2 errors of the same type were not detected. 1 Multiple L2 errors of the same type were detected. **Note:** This field is not part of *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.
33–55	—	Reserved
56	TMHIT	Tag multi-way hit detected. Writing a 1 to this bit location resets the bit. 0 Tag multi-way hit not detected. 1 Tag multi-way hit detected. **Note:** This field is not part of *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.
57–58	—	Reserved
59	TPARERR	Tag parity error detected. Writing a 1 to this bit location resets the bit. 0 Tag parity error not detected. 1 Tag parity error detected.
60	MBECCERR	Data Multiple-bit ECC error detected. Writing a 1 to this bit location resets the bit. 0 Tag Multiple-bit ECC error not detected. 1 Tag Multiple-bit ECC error detected.
61	SBECCERR	Data ECC error detected. Writing a 1 to this bit location resets the bit. 0 Tag Single-bit ECC error not detected. 1 Tag Single-bit ECC error detected.
62	PARERR	Data parity error detected. Writing a 1 to this bit location resets the bit. 0 Tag parity error not detected. 1 Tag parity error detected.
63	—	Reserved

2.15.4.3 L2 Cache Error Interrupt Enable Register (L2ERRINTEN)

L2ERRINTEN, shown in Figure 2-21, provides general status and information for errors detected in the L2 cache of the processor. The e500mc implements L2ERRINTEN as defined by the architecture and

described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* with the following exception:

- It does not implement the TMBECCINTEN, TSBECCINTEN, and L2CFGINTEN fields
- It does implement the implementation specific fields TMHITINTEN.

SPR 720 Hypervisor

	32–55	56	57–58	59	60	61	62	63
R W	—	TMHITINTEN	—	TPARINTEN	MBECCINTEN	SBECCINTEN	PARINTEN	—
Reset	All Zeros							

Figure 2-21. L2 Cache Error Interrupt Enable Register (L2ERRINTEN)

This table describes the L2ERRINTEN fields.

Table 2-22. L2ERRINTEN Field Descriptions

Bits	Name	Description
32–55	—	Reserved
56	TMHITINTEN	Tag multi-way hit interrupt reporting enable. 0 Tag multi-way hit interrupt reporting disabled. 1 Tag multi-way hit interrupt reporting enabled through a machine check exception. **Note:** This field is not part of *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.
57–58	—	Reserved
59	TPARINTEN	Tag parity error interrupt reporting enable. 0 Tag parity error interrupt reporting disabled. 1 Tag parity error interrupt reporting enabled through a machine check exception.
60	MBECCINTEN	Data Multiple-bit ECC error interrupt reporting enable. 0 Data Multiple-bit ECC error interrupt reporting disabled. 1 Data Multiple-bit ECC error interrupt reporting enabled through a machine check exception.
61	SBECCINTEN	Data ECC error interrupt reporting enable. 0 Data Single-bit ECC error interrupt reporting disabled. 1 Data Single-bit ECC error interrupt reporting enabled through a machine check exception.
62	PARINTEN	Data parity error interrupt reporting enable. 0 Data parity error interrupt reporting disabled. 1 Data parity error interrupt reporting enabled through a machine check exception.
63	—	Reserved

2.15.4.4 L2 Cache Error Control Register (L2ERRCTL)

L2ERRCTL, shown in Figure 2-21, provides thresholds and counts for errors detected in the L2 cache of the processor. The e500mc implements L2ERRCTL as defined by the architecture and described in the

EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors with the following exception: it does not implement the L2TCCOUNT field.

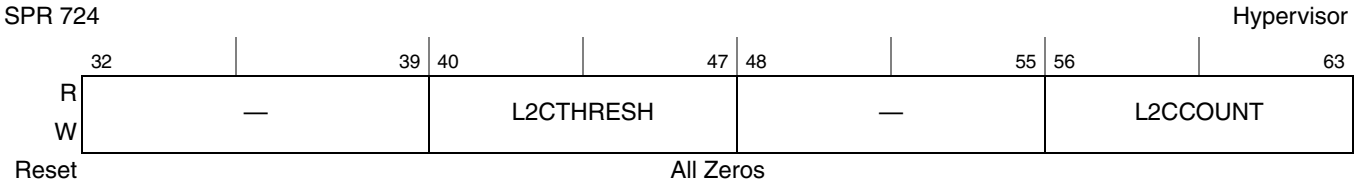


Figure 2-22. L2 Cache Error Control Register (L2ERRCTL)

This table describes the L2ERRCTL fields.

Table 2-23. L2ERRCTL Field Descriptions

Bits	Name	Description
32–39	—	Reserved
40–47	L2CTHRESH	L2 cache threshold. Threshold value for the number of ECC single-bit errors that are detected before reporting an error condition. L2CTHRESH is compared to L2CCOUNT each time a single-bit ECC error is detected.
48–55	—	Reserved
56–63	L2CCOUNT	L2 data ECC single-bit error count. Counts ECC single-bit errors in the L2 data detected. If L2CCOUNT equals the ECC single-bit error trigger threshold (L2CTHRESH), an error is reported if single-bit error reporting for data is enabled. Software should clear this value when such an error is reported to reset the count.

2.15.4.5 L2 Cache Error Address Capture Registers (L2ERRADDR and L2ERREADDR)

L2ERRADDR and L2ERREADDR provides the real address of a captured error detected in the L2 cache of the processor. The e500mc implements these registers as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

2.15.4.6 L2 Cache Error Capture Data Registers (L2CAPTDATALO and L2CAPTDATAHI)

L2CAPTDATALO and L2CAPTDATAHI provides the array data of a captured error detected in the L2 cache of the processor. L2CAPTDATALO captures the lower 32 bits of the doubleword and L2CAPTDATAHI captures the upper 32 bits of the doubleword. The e500mc implements these registers as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

2.15.4.7 L2 Cache Capture ECC Syndrome Register (L2CAPTECC)

L2CAPTECC provides both the calculated and stored ECC syndrome of a captured error detected in the L2 cache of the processor. The e500mc implements this register as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

2.15.4.8 L2 Cache Error Attribute Register (L2ERRATTR)

L2ERRATTR, shown in Figure 2-23, provides extended information for errors detected in the L2 cache of the processor. The e500mc implements L2ERRATTR as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*, and implements the implementation specific fields DWNUM, TRANSSRC, and TRANSTYPE.

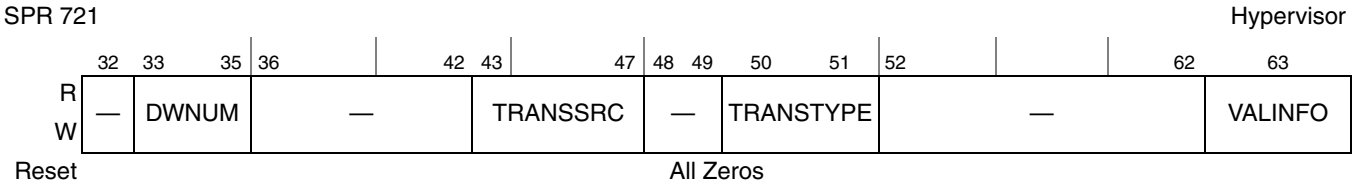


Figure 2-23. L2 Cache Error Attribute Register (L2ERRATTR)

This table describes the L2ERRATTR fields.

Table 2-24. L2ERRATTR Field Descriptions

Bits	Name	Description
32	—	Reserved
33-35	DWNUM	Doubleword number of the detected error (data ECC errors only). **Note:** This field is not part of *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.
36–42	—	Reserved
43-47	TRANSSRC	Transaction source for detected error 00000 External (snoop) 10000 Internal (instruction) 10001 Internal (data) 00001–01111 Not Implemented 10010–11111 Not Implemented **Note:** This field is not part of *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.
48–49	—	Reserved
50-51	TRANSTYPE	Transaction type for detected error 00 Snoop 01 Write 10 Read 11 Not Implemented **Note:** This field is not part of *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.
52–62	—	Reserved
63	VALINFO	L2 capture registers valid. 0 L2 capture registers contain no valid information or no enabled errors were detected. 1 L2 capture registers contain information of the first detected error which has reporting enabled. Software must clear this bit to unfreeze error capture so error detection hardware can overwrite the capture address/data/attributes for a newly detected error.

2.15.4.9 L2 Cache Error Injection Control Register (L2ERRINJCTL)

L2ERRINJCTL, shown in Figure 2-24, provides control for injecting errors into both the tags and data array for the L2 cache of the processor. The contents of L2ERRINJCTL as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* is implementation dependent, and all fields of this register are e500mc implementation-specific.

NOTE

When error injection is being performed, the value of specific error disables in L2ERRDIS and L2CSR0[L2PE] are ignored and errors are always detected. Software must ensure that L2PE is set and individual disables in L2ERRDIS are clear when performing error injection to the data or tags.

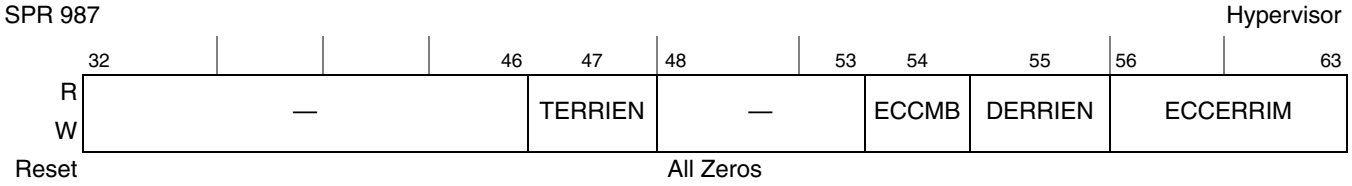


Figure 2-24. L2 Cache Error Injection Control Register (L2ERRINJCTL)

This table describes the L2ERRINJCTL fields.

Table 2-25. L2ERRINJCTL Field Descriptions

Bits	Name	Description
32–46	—	Reserved, should be 0.
47	TERRIEN	L2 tag array error injection. 0 No tag errors are injected. 1 All subsequent entries written to the L2 tag array have the parity inverted. **Note:** This field is not part of *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors.*
48–53	—	Reserved, should be 0.
54	ECCMB	ECC/Parity mirror byte enable. 0 ECC byte mirroring is disabled. 1 Each doubleword's most significant byte is mirrored onto the corresponding ECC/parity byte if DERRIEN = 1. **Note:** This field is not part of *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors.*

Table 2-25. L2ERRINJCTL Field Descriptions (continued)

Bits	Name	Description
55	DERRIEN	L2 data array error injection. 0 No data errors are injected. 1 Subsequent entries written to the L2 data array have data or ECC/parity bits inverted as specified in the data and ECC error injection masks and/or data path byte mirrored onto the ECC as specified by the ECC mirror bit enable. **Note:** If both ECC mirror byte and data error injection are enabled, ECC mask error injection is performed on the mirrored ECC. **Note:** This field is not part of *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.
56-63	ECCERRIM	Error injection mask for the ECC/parity bits. A set bit causes the corresponding ECC/parity bit to be inverted on writes if DERRIEN = 1. **Note:** This field is not part of *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

2.15.4.10 L2 Cache Error Injection Mask Registers (L2ERRINJLO and L2ERRINJHI)

L2ERRINJLO and L2ERRINJHI provide the injection mask describing how errors are to be injected into the data path doubleword in the L2 cache of the processor. L2ERRINJLO provides the mask for the lower 32 bits of the doubleword and L2ERRINJHI provides the mask for the upper 32 bits of the doubleword. A set bit in the injection mask causes the corresponding data path bit to be inverted on data array writes when L2ERRINJCTL[DERRIEN] = 1. The contents of L2ERRINJLO and L2ERRINJHI, as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors,* is implementation-dependent, and all fields of this register are e500mc implementation-specific.

2.16 MMU Registers

This section describes the following MMU registers and their fields:

- Logical Partition ID Register (LPIDR)
- Process ID Register (PID)
- MMU Control and Status Register 0 (MMUCSR0)
- MMU Configuration Register (MMUCFG)
- TLB Configuration Registers (TLBnCFG)
- MMU Assist Registers (MAS0–MAS8)

2.16.1 Logical Partition ID Register (LPIDR)

LPIDR contains the logical partition ID in use for the processor. LPIDR is part of the virtual address and is used during address translation comparing LPID to the Translation Logical Partition ID (TLPID) field in the TLB entry to determine a matching TLB entry. LPIDR is accessible by software only in hypervisor state (MSR[PR] = 0, MSR[GS] = 0). An attempt to read or write to LPIDR when the core is not in the

hypervisor state results in a hypervisor privilege exception when MSR[PR] = 0 and a privilege exception when MSR[PR] = 1.

Only the low-order 6 bits of LPIDR are implemented on e500mc.

When LPIDR is written the results of the change to LPIDR are not guaranteed to be seen until a context synchronizing event occurs.

2.16.2 Process ID Register (PID)

The architecture specifies that a process ID (PID) value be associated with each effective address (instruction or data) generated by the processor. PID values, defined by the PID register, are used to construct virtual addresses for accessing memory. The e500mc implements only the low-order 8 bits for the process ID. Writing to PIDs requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

2.16.3 MMU Control and Status Register 0 (MMUCSR0)

MMUCSR0 shown in Figure 2-25, is used to control the L2 MMUs. The e500mc implements the L2TLB0_FI and L2TLB1_FI TLB flash invalidate bits, which are implemented as they are defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

MMUCSR0 synchronization is described in Section 3.3.3, "Synchronization Requirements."

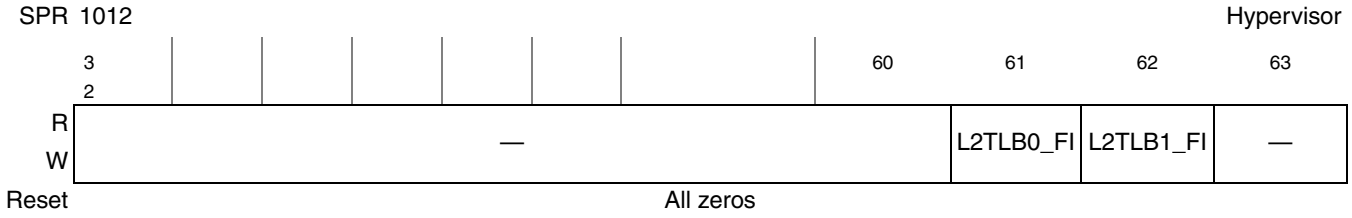


Figure 2-25. MMU Control and Status Register 0 (MMUCSR0)

This table describes the MMUCSR0 fields.

Table 2-26. MMUCSR0 Field Descriptions

Bits	Name	Description
32–60	—	Reserved
61	L2TLB0_FI	TLB0 flash invalidate (write 1 to invalidate) 0 No flash invalidate. Writing a 0 to this bit during an invalidation operation is ignored. 1 TLB1 invalidation operation. Hardware initiates a TLB1 invalidation operation. When this operation is complete, this bit is cleared. Writing a 1 during an invalidation operation causes an undefined operation. This invalidation typically takes 1 cycle.

Table 2-26. MMUCSR0 Field Descriptions (continued)

Bits	Name	Description
62	L2TLB1_FI	TLB1 flash invalidate (write 1 to invalidate) 0 No flash invalidate. Writing a 0 to this bit during an invalidation operation is ignored. 1 TLB1 invalidation operation. Hardware initiates a TLB1 invalidation operation. When this operation is complete, this bit is cleared. Writing a 1 during an invalidation operation causes an undefined operation. This invalidation typically takes 1 cycle.
63	—	Reserved

2.16.4 MMU Configuration Register (MMUCFG)

MMUCFG, shown in this figure, is implemented as defined by the architecture. It provides configuration information about the e500mc MMU.

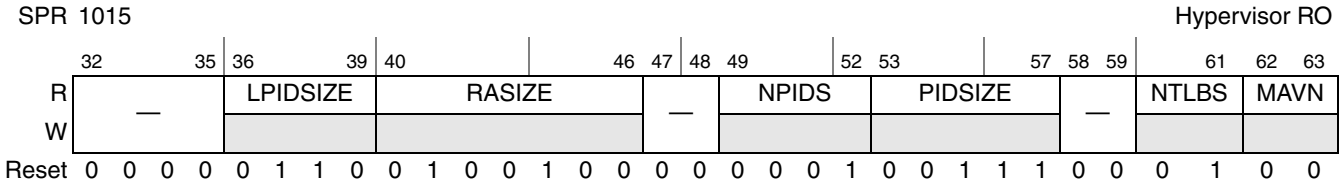


Figure 2-26. MMU Configuration Register (MMUCFG)

This table describes MMUCFG fields.

Table 2-27. MMUCFG Field Descriptions

Bits	Name	Description
32–35	—	Reserved
36–39	LPIDSIZE	LPID size. The number of LPID bits implemented. The processor implements only the least significant LPIDR bits. (0b0110 indicates LPIDR is 6 bits, LPIDR[58–63])
40–46	RASIZE	Real address size supported by the implementation. (0b0100100 indicates 36 physical address bits)
47–48	—	Reserved
49–52	NPIDS	Number of PID registers. Indicates the number of PID registers provided by the processor. (0b0001 indicates one PID register implemented)
53–57	PIDSIZE	PID register size. PIDSIZE is one less than the number of bits in each of the PID registers implemented by the processor. The processor implements only the least significant PIDSIZE+1 bits in the PID. (0b00111 indicates PID is 8 bits. PID[56–63])
58–59	—	Reserved
60–61	NTLBS	Number of TLBs. The value of NTLBS is one less than the number of software-accessible TLB structures that are implemented by the processor. NTLBS is set to one less than the number of TLB structures so that its value matches the maximum value of MAS0[TLBSEL]. (0b01 indicates two TLBs.)
62–63	MAVN	MMU architecture version number. Indicates the version number of the architecture of the MMU implemented by the processor. (0b00 indicates Version 1.0.)

2.16.5 TLB Configuration Registers (TLB*n*CFG)

TLB*n*CFG, shown in this figure, provides configuration information for TLB0 and TLB1 of the L2 MMU.

SPR 688 (TLB0CFG); 689 (TLB1CFG) — Hypervisor RO

	32–39	40–43	44–47	48	49	50–51	52–63
R	ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	—	NENTRY
W							
Reset: TLB0	0 0 0 0 0 1 0 0	0 0 0 1	0 0 0 1	0	0	0 0	0 0 1 0 0 0 0 0 0 0 0 0
TLB1	0 1 0 0 0 0 0 0	0 0 0 1	1 0 1 1	1	1	0 0	0 0 0 0 0 0 1 0 0 0 0 0

Figure 2-27. TLB Configuration Registers 0 and 1 (TLB0CFG, TLB1CFG)

This table describes the TLB*n*CFG fields and shows the values for the e500mc.

Table 2-28. TLB*n*CFG Field Descriptions

Bits	Name	Description
32–39	ASSOC	Associativity of TLB*n* TLB0: 0x04 Indicates associativity is 4-way set associative TLB1: 0x40 Indicates associativity is 64 (fully associative since ASSOC = NENTRY)
40–43	MINSIZE	Minimum page size of TLB*n* TLB0: 0x1 Indicates smallest page size is 4 Kbytes TLB1: 0x1 Indicates smallest page size is 4 Kbytes
44–47	MAXSIZE	Maximum page size of TLB*n* TLB0: 0x1 Indicates maximum page size is 4 Kbytes TLB1: 0xB Indicates maximum page size is 4 Gbytes
48	IPROT	Invalidate protect capability of TLB*n* TLB0: 0 Indicates invalidate protection capability not supported TLB1: 1 Indicates that TLB1 supports invalidate protection capability
49	AVAIL	Page size availability of TLB*n* TLB0: 0 No variable-sized pages available (MINSIZE = MAXSIZE) TLB1: 1 Indicates all page sizes between MINSIZE and MAXSIZE supported
50–51	—	Reserved
52–63	NENTRY	Number of entries in TLB*n* TLB0: 0x200 TLB0 contains 512 entries. TLB1: 0x040 TLB1 contains 64 entries

2.16.6 MMU Assist Registers (MAS0–MAS8)

MAS registers are used to manage pages and TLBs. Contents are written to the TLBs when a TLB Write Entry instruction (**tlbwe**) executes and are read from the TLBs when a TLB Read Entry instruction (**tlbre**) or a TLB Search (**tlbsx**) executes.

Writing to any MAS register requires synchronization prior to executing a TLB manipulation instruction (**tlbwe**, **tlbre**, **tlbilx**) which uses values in the MAS register to perform TLB operations. However, multiple MAS register updates can be performed and a single context synchronization instruction prior to the execution of the TLB manipulation instruction is sufficient to synchronize all the MAS register changes. Synchronization is described in Section 3.3.3, “Synchronization Requirements.”

TLB read (**tlbre**) and TLB write (**tlbwe**) instructions use MAS0[TLBSEL], MAS0[ESEL], and MAS2[EPN] to select which TLB entry to read from or write to. On e500mc, these fields are used as described by this table.

Table 2-29. TLB Selection Fields

TLB Array MAS0[TLBSEL]	MAS0[ESEL]	MAS2[EPN]	MAS0[NV]
0	MAS0[46:47] selects way (low order 2 bits of ESEL)	MAS2[45:51] selects set (low order 7 bits of EPN)	MAS0[62:63] indicates Next Victim value for ESEL
1	MAS0[42:47] selects entry (low order 6 bits of ESEL)	Not used, as TLB1 is fully associative	NV field not defined for this TLB Array

2.16.6.1 MAS Register 0 (MAS0)

MAS0, shown in Figure 2-28, is implemented as defined by the architecture. Only the low-order bit of TLBSEL, the low-order 6 bits of ESEL, and the low-order 2 bits of NV are implemented.

Writing to MAS0 requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

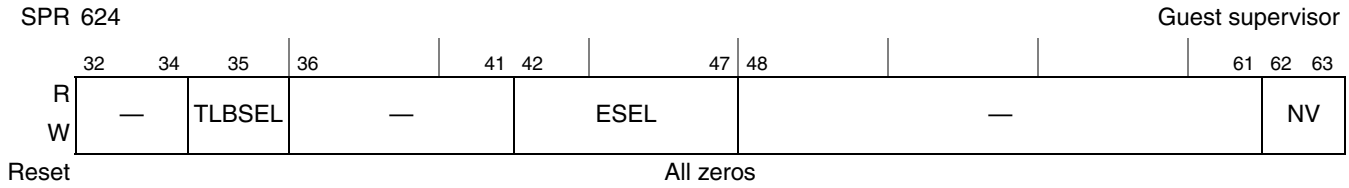


Figure 2-28. MAS Register 0 (MAS0)

The MAS0 fields are described in this table.

Table 2-30. MAS0 Field Descriptions—MMU Read/Write and Replacement Control

Bit	Name	Description
32–34	—	Reserved
35	TLBSEL	Selects TLB for access. 0 TLB0 1 TLB1
36–41	—	Reserved
42–47	ESEL	Entry select. Number of the entry in the selected array to be used for **tlbwe**. Updated on TLB error exceptions (misses) and **tlbsx** hit and miss cases. Only certain bits are valid, depending on the array selected in TLBSEL. Other bits should be 0.

Table 2-30. MAS0 Field Descriptions—MMU Read/Write and Replacement Control (continued)

Bit	Name	Description
48–61	—	Reserved
62–63	NV	Next victim. Can be used to identify the next victim to be targeted for a TLB miss replacement operation for those TLBs that support the NV field. For the e500mc, NV is the next victim value to be written to TLB0[NV] on execution of **tlbwe**. This field is also updated on TLB error exceptions (misses), **tlbsx** hit and miss cases, and on execution of **tlbre**. This field is updated based on the calculated next victim value for TLB0 (based on the round-robin replacement algorithm, described in Section 6.3.2.2, "Replacement Algorithms for L2 MMU Entries"). Note that this field is not defined for operations that specify TLB1 (when TLBSEL = 1).

2.16.6.2 MAS Register 1 (MAS1)

MAS1, shown in Figure 2-29, is implemented as defined by the architecture. Only the low-order 8 bits of TID are implemented and page sizes 4KB through 4GB are supported for TSIZE (when using TLB1).

Writing to MAS1 requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

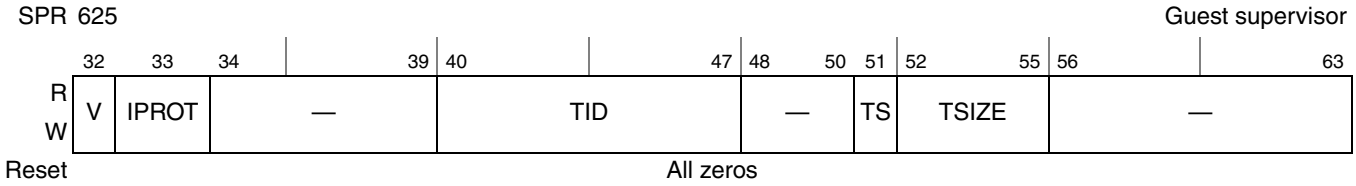


Figure 2-29. MAS Register 1 (MAS1)

The MAS1 fields are described in this table.

Table 2-31. MAS1 Field Descriptions—Descriptor Context and Configuration Control

Bits	Name	Descriptions
32	V	TLB valid bit. 0 This TLB entry is invalid. 1 This TLB entry is valid.
33	IPROT	Invalidate protect. Set to protect this TLB entry from invalidate operations from **tlbivax**, **tlbilx**, or MMUCSR0 TLB flash invalidates. Note that not all TLB arrays are necessarily protected from invalidation with IPROT. Arrays that support invalidate protection are denoted as such in the TLB configuration registers. 0 Entry is not protected from invalidation. 1 Entry is protected from invalidation.
34–39	—	Reserved
40–47	TID	Translation identity. Defines the process ID for this TLB entry. TID is compared to the process ID in the PID register during translation. A TID value of 0 defines an entry as global and matches with all process IDs.
48–50	—	Reserved
51	TS	Translation space. Compared with MSR[IS] (instruction fetch) or MSR[DS] (memory reference) to determine if this TLB entry may be used for translation.

Table 2-31. MAS1 Field Descriptions—Descriptor Context and Configuration Control (continued)

Bits	Name	Descriptions
52–55	TSIZE	Translation size. Defines the page size of the TLB entry. For TLB arrays with fixed-size TLB entries, TSIZE is ignored. For variable-size arrays, the page size is 4^{TSIZE} Kbytes. The e500mc supports the following sizes: 0001 4 Kbyte 0010 16 Kbyte 0011 64 Kbyte 0100 256 Kbyte 0101 1 Mbyte 0110 4 Mbyte 0111 16 Mbyte 1000 64 Mbyte 1001 256 Mbyte 1010 1 Gbyte 1011 4 Gbyte
56–63	—	Reserved

2.16.6.3 MAS Register 2 (MAS2)

MAS2, shown in this figure, is implemented as defined by the architecture. MAS2 is a 32-bit register. The ACM and VLE fields are not implemented.

Writing to MAS2 requires synchronization, as described in Section 3.3.3, “Synchronization Requirements.”

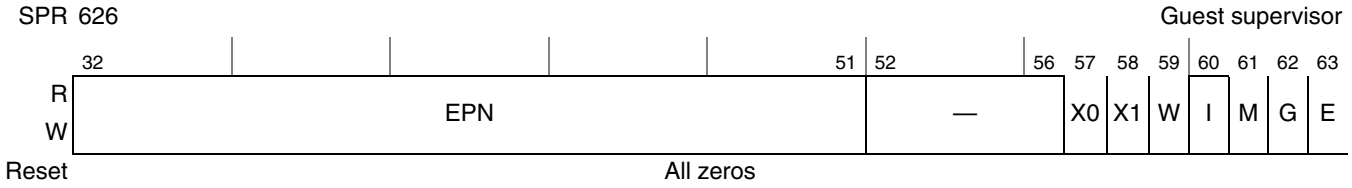


Figure 2-30. MAS Register 2 (MAS2)

The MAS2 fields are described in this table.

Table 2-32. MAS2 Field Descriptions—EPN and Page Attributes

Bits	Name	Description
32–51	EPN	Effective page number. Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero.
52–56	—	Reserved
57	X0	Implementation-dependent page attribute. Implemented as storage.
58	X1	Implementation-dependent page attribute. Implemented as storage.
59	W	Write-through 0 This page is considered write-back with respect to the caches in the system. 1 All stores performed to this page are written through the caches to main memory.

Table 2-32. MAS2 Field Descriptions—EPN and Page Attributes (continued)

Bits	Name	Description
60	I	Caching-inhibited 0 Accesses to this page are considered cacheable. 1 The page is considered caching-inhibited. All loads and stores to the page bypass the caches and are performed directly to main memory. A read or write to a caching-inhibited page affects only the memory element specified by the operation. **Note:** Cache-inhibited loads may hit in the L1 or L2 cache, but the transaction is always performed over CoreNet, ignoring the hit (although the hit may have other unarchitected side effects). Cache-inhibited loads that hit in the Data Line Fill Buffer (DLFB) are serviced out of the DLFB and are not performed over CoreNet. **Note:** Cache-inhibited (non-decorated, and non-guarded) loads execute speculatively on e500mc.
61	M	Memory coherency required 0 Memory coherency is not required. 1 Memory coherency is required. This allows loads and stores to this page to be coherent with loads and stores from other processors (and devices) in the system, assuming all such devices are participating in the coherency protocol.
62	G	Guarded 0 Accesses to this page are not guarded and can be performed before it is known if they are required by the sequential execution model. 1 All loads and stores to this page are performed without speculation (that is, they are known to be required). Guarded loads (that are not cache inhibited) execute speculatively out of the core caches, but executes non-speculatively if required to go off core to execute.
63	E	Endianness. Determines endianness for the corresponding page. Little-endian operation is true little endian, which differs from the modified little-endian byte ordering model available in some previous devices. 0 The page is accessed in big-endian byte order. 1 The page is accessed in true little-endian byte order.

2.16.6.4 MAS Register 3 (MAS3)

MAS3, shown in Figure 2-31, is implemented as defined by the architecture.

NOTE

When an operating system executing as a guest on a hypervisor uses the RPN fields of MAS3 and MAS7, the RPN should be interpreted by the hypervisor as a logical address or a guest physical address. The hypervisor re-writes the RPN field with a real physical address obtained from translating the logical address to a real physical address when emulating **tlbwe** instructions.

Writing to MAS3 requires synchronization, as described in Section 3.3.3, “Synchronization Requirements.”

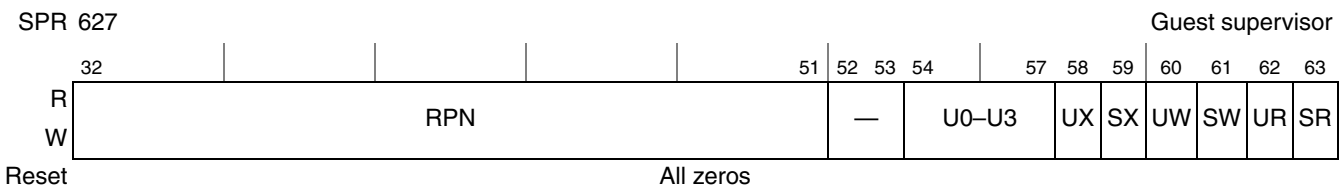


Figure 2-31. MAS Register 3 (MAS3)

The MAS3 fields are described in this table.

Table 2-33. MAS3 Field Descriptions—RPN and Access Control

Bits	Name	Description
32–51	RPN	Real page number. Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero. MAS3[RPN] contains only the low-order bits of the real page number. The high order bits of the real page number are located in MAS7. See Section 2.16.6.8, "MAS Register 7 (MAS7)," for more information.
52–53	—	Reserved
54–57	U0–U3	User attribute bits. These bits are associated with a TLB entry and can be used by system software. For example, these bits may be used to hold information useful to a page scanning algorithm or be used to mark more abstract page attributes.
58–63	UX,SX UW,SW , UR,SR	Permission bits (UX, SX, UW, SW, UR, SR). User and supervisor read, write, and execute permission bits. See the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* for more information on the page permission bits as they are defined by the architecture.

2.16.6.5 MAS Register 4 (MAS4)

MAS4, shown in Figure 2-32, is implemented as defined by the architecture. Only the low-order bit of TLBSELD is implemented and page sizes 4KB through 4GB are supported for TSIZED (when using TLB1). The ACMD and VLED fields are not implemented

Writing to MAS4 requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

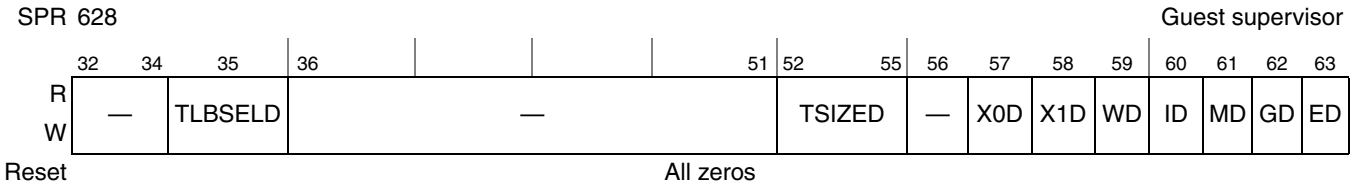


Figure 2-32. MAS Register 4 (MAS4)

The MAS4 fields are described in this table.

Table 2-34. MAS4 Field Descriptions—Hardware Replacement Assist Configuration

Bits	Name	Description
32–34	—	Reserved
35	TLBSELD	TLBSEL default value. Specifies the default value to be loaded in MAS0[TLBSEL] on a TLB miss exception.
36–51	—	Reserved
52–55	TSIZED	Default TSIZE value. Specifies the default value to be loaded into MAS1[TSIZE] on a TLB miss exception.
56	—	Reserved
57	X0D	Default X0 value. Specifies the default value to be loaded into MAS2[X0] on a TLB miss exception.
58	X1D	Default X1 value. Specifies the default value to be loaded into MAS2[X1] on a TLB miss exception.

Table 2-34. MAS4 Field Descriptions—Hardware Replacement Assist Configuration (continued)

Bits	Name	Description
59	WD	Default W value. Specifies the default value to be loaded into MAS2[W] on a TLB miss exception.
60	ID	Default I value. Specifies the default value to be loaded into MAS2[I] on a TLB miss exception.
61	MD	Default M value. Specifies the default value to be loaded into MAS2[M] on a TLB miss exception.
62	GD	Default G value. Specifies the default value to be loaded into MAS2[G] on a TLB miss exception.
63	ED	Default E value. Specifies the default value to be loaded into MAS2[E] on a TLB miss exception.

2.16.6.6 MAS Register 5 (MAS5)

MAS5, shown in Figure 2-33, is implemented as defined by the architecture. MAS5 contains hypervisor fields for specifying LPID and GS values to be used to search TLB entries with a **tlbsx** instruction and for specifying LPID values to invalidate TLB entries with a **tlbilx** instruction. Only the low-order 6 bits of SLPID are implemented.

Writing to MAS5 requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

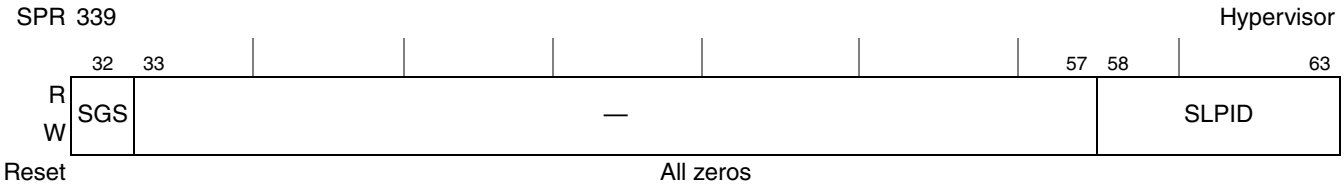


Figure 2-33. MAS Register 5 (MAS5)

The MAS5 fields are described in this table.

Table 2-35. MMU Assist Register 5 (MAS5) Register Fields

Bits	Name	Architecture Note
32	SGS	Search GS. Specifies the GS value used when searching the TLB during execution of **tlbsx**. The SGS field is compared with the Translated (TGS) field of each TLB entry to find a matching entry.
33–57	—	Reserved
58–63	SLPID	Search LPID. Specifies the LPID value used when searching the TLB during execution of **tlbsx**. The SLPID field is compared with the TLPID field of each TLB entry to find a matching entry.

2.16.6.7 MAS Register 6 (MAS6)

MAS6, shown in Figure 2-34, is implemented as defined by the architecture. Only the low-order 8 bits of the SPID field are implemented.

Note the SPID field was previously named SPID0. Both names refer to the same field.

Writing to MAS6 requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

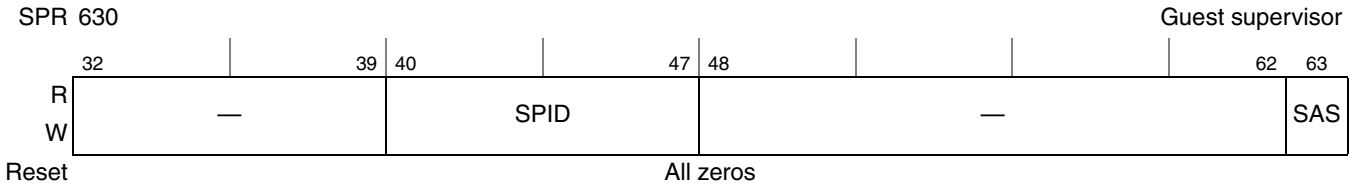


Figure 2-34. MAS Register 6 (MAS6)

The MAS6 fields are described in this table.

Table 2-36. MAS6 Field Descriptions

Bits	Name	Description
32–39	—	Reserved
40–47	SPID	Search PID. Specifies the value of PID used when searching the TLB during execution of **tlbsx**. For the e500mc, SPID contains the search PID value used when searching the TLB during execution of **tlbsx**.
48–62	—	Reserved
63	SAS	Address space (AS) value for searches. Specifies the value of AS used when searching the TLB during execution of **tlbsx**.

2.16.6.8 MAS Register 7 (MAS7)

MAS7, shown in Figure 2-35, is implemented as defined by the architecture. MAS7 contains the high-order 32-bits of the real (physical) page number. Since e500mc supports 36 bits of physical address, only the low-order 4 bits of the high-order 32-bits of the real address (RPN) are implemented.

NOTE

When an operating system executing as a guest on a hypervisor uses the RPN fields of MAS3 and MAS7, the RPN should be interpreted by the hypervisor as a logical address or a guest physical address. The hypervisor re-writes the RPN field with a real physical address obtained from translating the logical address to a real physical address when emulating **tlbwe** instructions.

Writing to MAS7 requires synchronization, as described in Section 3.3.3, “Synchronization Requirements.”

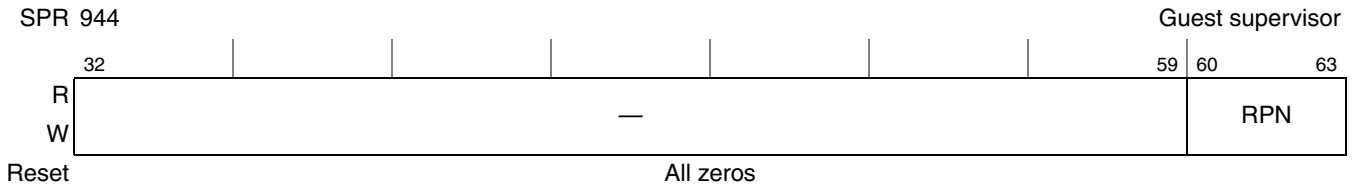


Figure 2-35. MAS Register 7 (MAS7)

The MAS7 fields are described in this table.

Table 2-37. MAS7 Field Descriptions—High-Order RPN

Bits	Name	Description
32–59	—	Reserved
60–63	RPN	Real page number, 4 high-order bits. MAS3 holds the remainder of the RPN. The byte offset within the page is provided by the EA and is not present in MAS3 or MAS7.

2.16.6.9 MAS Register 8 (MAS8)

MAS8, shown in Figure 2-36, is implemented as defined by the architecture. MAS8 contains hypervisor state fields used for selecting a TLB entry during translation. Only the low-order 6 bits of TLPID are implemented.

Writing to MAS8 requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

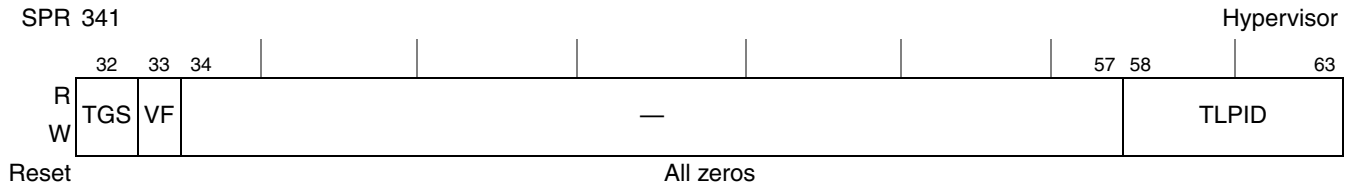


Figure 2-36. MAS Register 8 (MAS8) Format

MAS8 fields are described in this table.

Table 2-38. MMU Assist Register 8 (MAS8) Register Fields

Bits	Name	Description
32	TGS	Translation guest space. During translation, TGS is compared with MSR[GS] to select a TLB entry.
33	VF	Virtualization fault. Controls whether a DSI occurs on data accesses to the page, regardless of permission bit settings. 0 Data accesses translated by this TLB entry occur normally. 1 Data accesses translated by this TLB entry always cause a data storage interrupt directed to the hypervisor.
34–57	—	Reserved
58–63	TLPID	Translation logical partition ID. During translation, TLPID is compared with the LPIDR to select a TLB entry. A TLPID value of 0 defines an entry as global and matches all values of LPIDR.

2.16.7 External PID Registers

The e500mc implements the external PID load and store context registers (EPLC and EPSC) as they are defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

2.16.7.1 External PID Load Context Register (EPLC)

The EPLC register contains fields to provide the context for external PID load instructions. Figure 2-37 shows the format of the EPLC register. Only the low-order 6 bits of the ELPID field and the low-order 8 bits of the EPID field are implemented.

Writing to EPLC requires synchronization, as described in Section 3.3.3, "Synchronization Requirements."

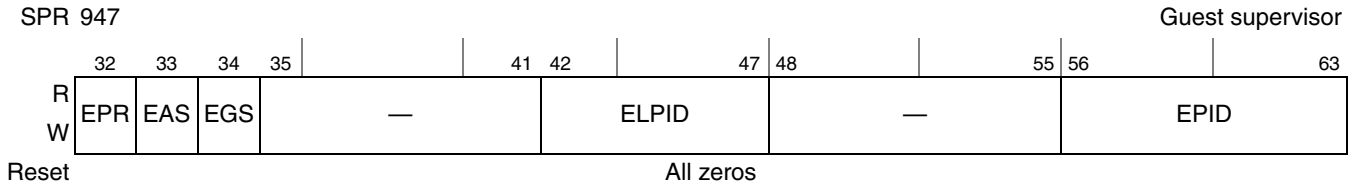


Figure 2-37. External PID Load Context (EPLC) Format

The EPLC fields are described in this table.

Table 2-39. EPLC Fields—External PID Load Context

Bits	Name	Descriptions
0–31	—	Reserved
32	EPR	External load context PR bit. Used in place of MSR[PR] for load permission checking when an external PID load instruction is executed. 0 Supervisor mode. 1 User mode.
33	EAS	External load context AS bit. Used in place of MSR[DS] for load translation when an External PID Load instruction is executed. Compared with TLB[TS] during translation. 0 Address space 0. 1 Address space 1.
34	EGS	External load context GS bit. Used in place of MSR[GS] for load translation when an External PID Load instruction is executed. Compared with TLB[TGS] during translation.This field is only writable in hypervisor state (MSR[PR] = 0 and MSR[GS] = 0). 0 Hypervisor address space. 1 Guest address space.
35–41	—	Reserved
42–47	ELPID	External load context LPID value. Used in place of LPIDR value for load translation when an external PID Load instruction is executed. Compared with TLB[TLPID] during translation. This field is only writable in hypervisor state (MSR[PR] = 0 and MSR[GS] = 0).
48–55	—	Reserved
56–63	EPID	External load context PID value. Used in place of all PID register values for load translation when an external PID Load instruction is executed. Compared with TLB[TID] during translation.

2.16.7.2 External PID Store Context (EPSC) Register

EPSC, shown in Figure 2-38, contains fields to provide the context for external PID store instructions. The field encoding is the same as EPLC. Only the low-order 6 bits of the ELPID field and the low-order 8 bits of the EPID field are implemented.

Writing to EPSC requires synchronization, as described in Section 3.3.3, “Synchronization Requirements.”

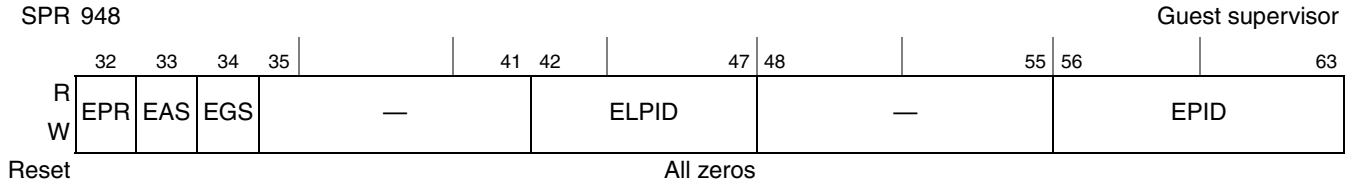


Figure 2-38. External PID Store Context (EPSC) Format

The EPSC fields are described in this table.

Table 2-40. EPSC Fields—External PID Store Context

Bits	Name	Descriptions
0–31	—	Reserved
32	EPR	External store context PR bit. Used in place of MSR[PR] for store permission checking when an External PID Store instruction is executed. 0 Supervisor mode 1 User mode.
33	EAS	External store context AS bit. Used in place of MSR[DS] for store translation when an External PID Store instruction is executed. Compared with TLB[TS] during translation. 0 Address space 0 1 Address space 1
34	EGS	External store context GS bit. Used in place of MSR[GS] for store translation when an External PID Store instruction is executed. Compared with TLB[TGS] during translation.This field is only writable in hypervisor state (MSR[PR] = 0 and MSR[GS] = 0). 0 Hypervisor address space. 1 Guest address space.
35–41	—	Reserved
42–47	ELPID	External store context LPID value. Used in place of LPIDR value for store translation when an external PID Store instruction is executed. Compared with TLB[TLPID] during translation. This field is only writable in hypervisor state (MSR[PR] = 0 and MSR[GS] = 0).
48–55	—	Reserved
56–63	EPID	External store context PID value. Used in place of all PID register values for store translation when an external PID Store instruction is executed. Compared with TLB[TID] during translation.

2.17 Internal Debug Registers

This section describes debug-related registers that are accessible to software running on the processor. These registers are intended for use by special debug tools and debug software, and not by general application or operating system code.

The e500mc implements the category Embedded.Enhanced Debug from Power ISA 2.06 which provides a separate set of save/restore registers for debug interrupts (DSRR0/DSRR1, see Section 2.9.1, “Save/Restore Registers (xSRR0/xSRR1)”), an **rfdi** instruction to return from debug interrupts, and additional debug events for Critical Interrupt Taken and Critical Interrupt Return.

The debug registers listed here generally only describe the registers and facilities that are used by software in the internal debug mode (when DBCR0[IDM] = 1). More detailed description of the debug facilities is described in Chapter 9, "Debug and Performance Monitor Facilities."

2.17.1 Unimplemented Internal Debug Registers

The e500mc does not implement the following internal debug registers defined by Power ISA 2.06 and *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*:

- DBCR3
- IAC3, IAC4
- DVC1, DVC2

2.17.2 Debug Control Register 0 (DBCR0)

DBCR0 is used to enable debug conditions, reset the processor, and control timer operation during debug events. DBCR0 is implemented on e500mc as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*, except for the following differences:

- IAC3 and IAC4 are not implemented
- DBCR0[RST] encodings are more explicitly defined for the e500mc implementation
- When in external debug mode (EDM) (DBCR0[EDM] = 1), software writes to this register while e500mc is not halted are ignored

DBCR0, shown in Figure 2-39, contains bits for enabling debug conditions.

SPR 308 — Hypervisor

	32	33	34 35	36	37	38	39	40	41	42 43	44 45	46 47	48	49 56	57	58	59 62	63
R[1]	EDM	IDM	RST	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	—	DAC1	DAC2	RET	—	CIRPT	CRET	—	FT
W[2]																		
Reset	All zeros																	

Figure 2-39. Debug Control Register 0 (DBCR0)

1 All reserved bits read as zero

2 When in EDM (DBCR0[EDM] = 1) software writes to this register are ignored while e500mc is not halted.

This table provides the bit definitions for DBCR0.

Table 2-41. DBCR0 Field Descriptions

Bits	Name	Description
32	EDM	External Debug Mode. This bit is read only by software. It reflects the status of EDBCR0[EDM]. 0 Indicates the processor is not in external debug mode. External debug events are disabled. 1 Indicates the processor is in external debug mode. A qualified debug condition generates an external debug event by updating the corresponding bit in EDBSR0 and causing the processor to halt.
33	IDM	Internal Debug Mode 0 Internal debug events are disabled. 1 Internal debug events are enabled if DBCR0[EDM] = 0. A qualified debug condition generates an internal debug event by updating the corresponding bit in the DBSR. If MSR[DE] = 1 and DBCR0[EDM] = 0, the occurrence of a debug event, or the recording of an earlier debug event in the DBSR when MSR[DE] was cleared, causes a debug interrupt.
34–35	RST	Reset. The architecture defines this field such that 00 is always no action and all other settings are implementation specific. e500mc implements these bits as follows: 0*x* Default (No action) 1*x* Core reset. Requests a core hard reset if MSR[DE] and DBCR0[IDM] are set. Always cleared on subsequent cycle.
36	ICMP	Instruction Complete Debug Condition Enable 0 ICMP debug conditions are disabled 1 ICMP debug conditions are enabled
37	BRT	Branch Taken Debug Condition Enable 0 BRT debug conditions are disabled 1 BRT debug conditions are enabled
38	IRPT	Interrupt Taken Debug Condition Enable. This bit affects only non-critical, non-debug, and non-machine check interrupts. 0 IRPT debug conditions are disabled 1 IRPT debug conditions are enabled
39	TRAP	Trap Debug Condition Enable 0 TRAP debug conditions are disabled 1 TRAP debug conditions are enabled
40	IAC1	Instruction Address Compare 1 Debug Condition Enable 0 IAC1 debug conditions are disabled 1 IAC1 debug conditions are enabled
41	IAC2	Instruction Address Compare 2 Debug Condition Enable 0 IAC2 debug conditions are disabled 1 IAC2 debug conditions are enabled
42–43	—	Reserved
44–45	DAC1	Data Address Compare 1 Debug Condition Enable 00 DAC1 debug conditions are disabled 01 DAC1 debug conditions are enabled only for store-type data storage accesses 10 DAC1 debug conditions are enabled only for load-type data storage accesses 11 DAC1 debug conditions are enabled for load-type or store-type data storage accesses

Table 2-41. DBCR0 Field Descriptions (continued)

Bits	Name	Description
46–47	DAC2	Data Address Compare 2 Debug Condition Enable 00 DAC2 debug conditions are disabled 01 DAC2 debug conditions are enabled only for store-type data storage accesses 10 DAC2 debug conditions are enabled only for load-type data storage accesses 11 DAC2 debug conditions are enabled for load-type or store-type data storage accesses
48	RET	Return Debug Condition Enable This bit affects only non-critical, non-debug, and non-machine check interrupts. 0 RET debug conditions are disabled 1 RET debug conditions are enabled
49–56	—	Reserved
57	CIRPT	Critical Interrupt Taken Debug Condition Enable 0 CIRPT debug conditions are disabled. 1 CIRPT debug conditions are enabled.
58	CRET	Return From Critical Interrupt Debug Condition Enable 0 CRET debug conditions are disabled. 1 CRET debug conditions are enabled.
59–62	—	Reserved
63	FT	Freeze Timers on Debug Event 0 Timebase counters are unaffected by DBSR bits 1 Disable clocking of TimeBase counters whenever a DBSR bit is set (excluding DBSR[MRR])

2.17.3 Debug Control Register 1 (DBCR1)

DBCR1 is implemented as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*, with the following exceptions:

- When in EDM, software writes to this register while e500mc is not halted are ignored
- IAC1 and IAC2 comparisons must be based on effective addresses. Comparisons based on real addresses are not supported
- IAC3 and IAC4 debug conditions are not implemented
- When IAC12M != 00, IAC2US and IAC2ER settings must match IAC1US and IAC1ER or results are boundedly undefined

SPR 309 Hypervisor

	32 33	34 35	36 37	38 39	40 41	42				63
R W[1]	IAC1US	IAC1ER	IAC2US	IAC2ER	IAC12M	—				
Reset	All zeros									

Figure 2-40. Debug Control Register 1 (DBCR1)

[1] When in EDM (DBCR0[EDM] = 1) software writes to this register are ignored while the e500mc is not halted.

This table provides the bit definitions for DBCR1.

Table 2-42. DBCR1 Field Descriptions

Bits	Name	Description
32–33	IAC1US	Instruction Address Compare 1 User/Supervisor Mode 00 IAC1 debug conditions unaffected by MSR[PR],MSR[GS] 01 Reserved on e500mc 10 IAC1 debug conditions can only occur if MSR[PR] = 0 (supervisor mode) 11 IAC1 debug conditions can only occur if MSR[PR] = 1 (user mode)
34–35	IAC1ER	Instruction Address Compare 1 Effective/Real Mode 00 IAC1 debug conditions are based on effective addresses 01 Reserved on e500mc 10 IAC1 debug conditions are based on effective addresses and can occur only if MSR[IS] = 0 11 IAC1 debug conditions are based on effective addresses and can occur only if MSR[IS] = 1
36–37	IAC2US	Instruction Address Compare 2 User/Supervisor Mode 00 IAC2 debug conditions unaffected by MSR[PR],MSR[GS] 01 Reserved on e500mc 10 IAC2 debug conditions can only occur if MSR[PR]=0 (supervisor mode) 11 IAC2 debug conditions can only occur if MSR[PR]=1 (user mode)
38–39	IAC2ER	Instruction Address Compare 2 Effective/Real Mode 00 IAC2 debug conditions are based on effective addresses 01 Reserved on e500mc 10 IAC2 debug conditions are based on effective addresses and can occur only if MSR[IS] = 0 11 IAC2 debug conditions are based on effective addresses and can occur only if MSR[IS] = 1
40–41	IAC12M	Instruction Address Compare 1/2 Mode. 00 Exact address compare. IAC1 debug conditions can only occur if the address of the instruction fetch is equal to the value specified in IAC1. IAC2 debug conditions can only occur if the address of the instruction fetch is equal to the value specified in IAC2. IAC1US, IAC1ER, and DBCR0[IAC1] are used for IAC1 conditions. IAC2US, IAC2ER, and DBCR0[IAC2] are used for IAC2 conditions. 01 Address bit match. IAC1 debug conditions can occur only if the address of the instruction fetch, ANDed with the contents of IAC2 is equal to the contents of IAC1, also ANDed with the contents of IAC2. IAC2 debug conditions do not occur. The DBCR0[IAC1] setting is used. The value of DBCR0[IAC2] is ignored. If IAC1US and IAC1ER do not match IAC2US and IAC2ER values, results are boundedly undefined. 10 Inclusive address range compare. IAC1 debug conditions can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC1 and less than the value specified in IAC2[1]. IAC2 debug conditions do not occur. The DBCR0[IAC1] setting is used. The value of DBCR0[IAC2] is ignored. If IAC1US and IAC1ER do not match IAC2US and IAC2ER values, results are boundedly undefined. 11 Exclusive address range compare. IAC1 debug conditions can occur only if the address of the instruction fetch is less than the value specified in IAC1 or is greater than or equal to the value specified in IAC2[2]. IAC2 debug conditions do not occur. The DBCR0[IAC1] setting is used. The value of DBCR0[IAC2] is ignored. If IAC1US and IAC1ER do not match IAC2US and IAC2ER values, results are boundedly undefined. e500mc sets both DBSR[IAC1] and DBSR[IAC2] bits if IAC12M is set to anything other than 0b00 and an instruction address compare 1 or 2 event occurs.
42–63	—	Reserved

1 If IAC1 > IAC2 or IAC1 = IAC2 a valid condition never occurs.

2 If IAC1 > IAC2 or IAC1 = IAC2 a valid condition may occur on every instruction fetch.

2.17.4 Debug Control Register 2 (DBCR2)

DBCR2 is implemented as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*, except for the following differences:

- When in EDM, software writes to this register are ignored while the e500mc is not halted.
- DAC comparisons are based on effective addresses only.
- Data Value Compare is not implemented.
- DACLINK1 and DACLINK2 are implemented.

This figure shows the debug control register 2.

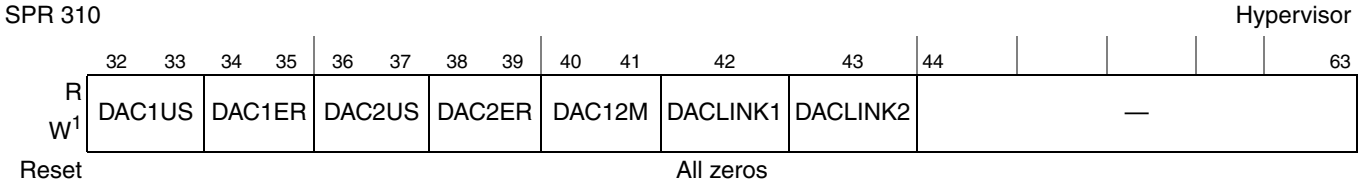


Figure 2-41. Debug Control Register 2 (DBCR2)

[1] When in EDM (DBCR0[EDM]=1) software writes to this register are ignored while the e500mc is not halted.

This table provides the bit definitions for DBCR2.

Table 2-43. DBCR2 Field Descriptions

Bits	Name	Description
32–33	DAC1US	Data Address Compare 1 User/Supervisor Mode 00 DAC1 debug conditions unaffected by MSR[PR] 01 Reserved on e500mc 10 DAC1 debug conditions can only occur if MSR[PR] = 0 (supervisor mode) 11 DAC1 debug conditions can only occur if MSR[PR] = 1 (user mode)
34–35	DAC1ER	Data Address Compare 1 Effective/Real mode 00 DAC1 debug conditions are based on effective addresses 01 Reserved on e500mc 10 DAC1 debug conditions are based on effective addresses and can occur only if MSR[DS] = 0 11 DAC1 debug conditions are based on effective addresses and can occur only if MSR[DS] = 1
36–37	DAC2US	Data Address Compare 2 User/Supervisor Mode 00 DAC2 debug conditions unaffected by MSR[PR], MSR[GS] 01 Reserved on e500mc 10 DAC2 debug conditions can only occur if MSR[PR] = 0 (supervisor mode) 11 DAC2 debug conditions can only occur if MSR[PR] = 1 (user mode)
38–39	DAC2ER	Data Address Compare 2 Effective/Real mode 00 DAC2 debug conditions are based on effective addresses 01 Reserved on 10 DAC2 debug conditions are based on effective addresses and can occur only if MSR[DS] = 0 11 DAC2 debug conditions are based on effective addresses and can occur only if MSR[DS] = 1

Table 2-43. DBCR2 Field Descriptions (continued)

Bits	Name	Description
40–41	DAC12M	Data Address Compare 1/2 Mode 00 Exact address compare. DAC1 debug conditions can only occur if the data storage address is equal to the value specified in DAC1. DAC2 debug conditions can only occur if the data storage address is equal to the value specified in DAC2. DAC1US, DAC1ER, and DBCR0[DAC1] are used for DAC1 conditions. DAC2US, DAC2ER, and DBCR0[DAC2] are used for DAC2 conditions [1] 01 Address bit match. DAC1 debug conditions can occur only if the data storage address ANDed with the contents of DAC2 is equal to the contents of DAC1 also ANDed with the contents of DAC2. DAC2 debug conditions do not occur. The DBCR0[DAC1] setting is used. The value of DBCR0[DAC2] is ignored. DAC1US and DAC1ER values are used and DAC2US and DAC2ER values are ignored. 10 Inclusive address range compare. DAC1 debug conditions can occur only if the data storage address is greater than or equal to the value specified in DAC1 and less than the value specified in DAC2[2]. DAC2 debug conditions do not occur. The DBCR0[DAC1] setting is used. The value of DBCR0[DAC2] is ignored. DAC1US and DAC1ER values are used and DAC2US and DAC2ER values are ignored. 11 Exclusive address range compare. DAC1 debug conditions can occur only if the data storage address is less than the value specified in DAC1 or is greater than or equal to the value specified in DAC2[3]. DAC2 debug conditions do not occur. The DBCR0[DAC1] setting is used. The value of DBCR0[DAC2] is ignored. DAC1US and DAC1ER values are used and DAC2US and DAC2ER values are ignored. e500mc sets both DBSR[DAC1] and DBSR[DAC2] bits if DAC12M is set to anything other than 0b00 and a data address compare 1 or 2 event occurs
42	DACLINK1	Data Address Compare 1 Link to Instruction Address Compare 1 0 No effect 1 DAC1 debug events are linked to IAC1 debug conditions. IAC1 debug conditions do not affect DBSR or EDBSR0. When linked to IAC1, the DAC1 debug event is qualified based on whether the instruction also generated an IAC1 debug condition.
43	DACLINK2	Data Address Compare 2 Link to Instruction Address Compare 2 0 No effect 1 DAC2 debug events are linked to IAC2 debug conditions. IAC2 debug conditions do not affect DBSR or EDBSR0. When linked to IAC2, the DAC2 debug event is qualified based on whether the instruction also generated an IAC2 debug condition.
44–63	—	Reserved

1 See DBCR4 for extensions to the exact address match (range defined)

2 If DAC1 > DAC2 or DAC1=DAC2 a valid condition never occurs.

3 If DAC1 > DAC2 or DAC1=DAC2 a valid condition may occur on every data storage address.

2.17.5 Debug Control Register 4 (DBCR4)

DBCR4 is used to enable debug modes and provide additional debug controls. The e500mc adds some implementation specific bits to this register, as described in this figure.

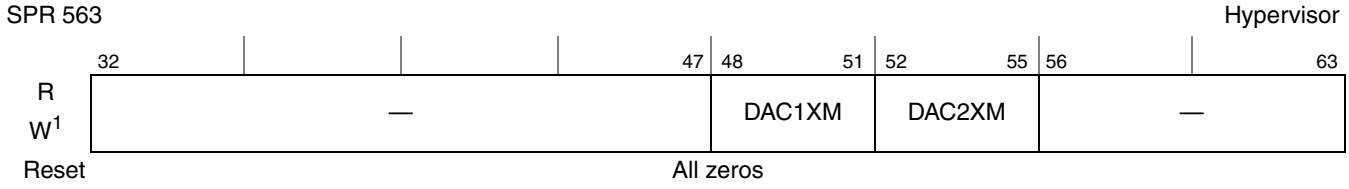


Figure 2-42. Debug Control Register 4 (DBCR4)

[1] When in EDM (DBCR0[EDM]=1) software writes to this register are ignored while the e500mc is not halted.

This table provides the bit definitions for DBCR0.

Table 2-44. DBCR4 Field Descriptions

Bits	Name	Description
32–47	—	Reserved
48-51	DAC1XM	Data Address Compare 1—Extended Mask Control 0000 No additional masking when DBCR2[DAC12M] = 00 0001–1100 Exact Match Bit Mask. Number of low order bits masked in DAC1 when comparing the storage address with the value in DAC1 for exact address compare (DBRCR2[DAC12M] = 00). The e500mc supports ranges up to 4KB. 1101–1111 Reserved
52-55	DAC2XM	Data Address Compare 2—Extended Mask Control 0000 No additional masking when DBCR2[DAC12M] = 00 0001–1100 Exact Match Bit Mask. Number of low order bits masked in DAC2 when comparing the storage address with the value in DAC2 for exact address compare (DBRCR2[DAC12M] = 00). The e500mc supports ranges up to 4KB 1101–1111 Reserved
56–63	—	Reserved

2.17.6 Debug Status Register (DBSR/DBSRWR)

DBSR provides status information for debug events when DBCR0[IDM] = 1 and DBCR0[EDM] = 0, and for the most recent processor reset.

DBSR is implemented as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*, with the following exceptions:

- When in EDM, software writes to this register are ignored while the e500mc is not halted
- When in EDM, debug events update EDBSR0 instead of DBSR
- Two additional debug events are possible: CIRPT and CRET

DBSRWR is implemented as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*, and is used to write the value of the DBSR to a specific value. DBSRWR is a write-only register.

DBSR is a write-one-to-clear register. Software should normally write DBSR with a mask specifying which bits of DBSR to clear. DBSRWR should only be used to restore a DBSR value in the case of a hypervisor partition switch.

Writing DBSRWR changes the value of the DBSR which, if nonzero, may cause later imprecise debug interrupts.

This figure shows the debug status register write register.

SPR 306 (DBSRWR) Hypervisor WO

	32	33	34 35	36	37	38	39	40	41	42 43	44	45	46	47	48	49 56	57	58	59 63
R										—						—			—
W[1]	IDE	UDE	MRR	ICMP	BRT	IRPT	TRAP	IAC1	IAC2		DAC1R	DAC1W	DAC2R	DAC2W	RET		CIRPT	CRET	
Reset	Contents can be read through DBSR only																		

Figure 2-43. Debug Status Register Write Register (DBSRWR)

[1] When in EDM (DBCR0[EDM] = 1) software writes to this register are ignored while e500mc is not halted.

This figure shows the debug status register.

SPR 304 (DBSR) Hypervisor

	32	33	34 35	36	37	38	39	40	41	42 43	44	45	46	47	48 49	56	57	58	59 63
R[1]/W[2]	IDE	UDE	MRR	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	—	DAC1R	DAC1W	DAC2R	DAC2W	RET	—	CIRPT	CRET	—
Reset	0	0	1 0	0	0	0	0	0	0	0 0	0	0	0	0	0	0 0 0 00	0	0	0..0

Figure 2-44. Debug Status Register (DBSR)

[1] All reserved bits read as zero

[2] Writing to DBSR clears any bits that set to 1 in the corresponding value being written from the source register (write-one-to-clear). When in EDM (DBCR0[EDM] = 1), software writes to this register are ignored while e500mc is not halted.

This table provides the bit definitions for DBSR and DBSRWR.

Table 2-45. DBSR/DBSRWR Field Descriptions

Bits	Name	Description
32	IDE	Imprecise Debug Event 0 No imprecise debug events have occurred 1 An imprecise debug event has occurred while MSR[DE] = 0 and DBCR0[IDM] = 1 and DBCR0[EDM] = 0
33	UDE	Unconditional Debug Event 0 No unconditional debug events have occurred 1 An unconditional debug event has occurred while DBCR0[IDM] = 1 and DBCR0[EDM] = 0. Note that unconditional debug events are not affected by EPCR[DUVD] on the e500mc. An unconditional debug event can occur when the $\overline{\text{UDE}}$ signal (level sensitive, active low) is asserted to the core. When UDE is asserted, DBSR[UDE] is set to 1 if DBCR0[IDM] = 1 and DBCR0[EDM] = 0. When DBSR[UDE] is set, DBSR[IDE] is also set.
34–35	MRR	Most Recent Reset. The e500mc implements MRR as follows: 00 No hard reset occurred since this bit was last cleared by software. 01 Reserved 10 The previous reset was a hard reset (default value on power-up). 11 Reserved
36	ICMP	Instruction Complete Debug Event 0 No instruction complete debug event has occurred 1 An instruction complete debug event has occurred while DBCR0[ICMP] = 1, DBCR0[IDM] = 1, DBCR0[EDM] = 0, and MSR[DE] = 1. See Section 9.8.9, “Instruction Complete Debug Event,” for more details.

Table 2-45. DBSR/DBSRWR Field Descriptions (continued)

Bits	Name	Description
37	BRT	Branch Taken Debug Event 0 No branch taken debug event has occurred 1 A branch taken debug event has occurred while DBCR0[BRT] = 1, DBCR0[IDM] = 1, DBCR0[EDM] = 0, and MSR[DE] = 1. See Section 9.8.8, “Branch Taken Debug Event,” for more details.
38	IRPT	Interrupt Taken Debug Event 0 No interrupt taken debug event has occurred 1 An interrupt taken debug event has occurred while DBCR0[IRPT] = 1, DBCR0[IDM] = 1, and DBCR0[EDM] = 0. See Section 9.8.10, “Interrupt Taken Debug Event,” for more details.
39	TRAP	Trap Instruction Debug Event 0 No trap instruction debug event has occurred 1 A trap instruction debug event has occurred while DBCR0[TRAP] = 1, DBCR0[IDM] = 1, and DBCR0[EDM] = 0. See Section 9.8.11, “Interrupt Return Debug Event,” for more details.
40	IAC1	Instruction Address Compare 1 Debug Event 0 No instruction address compare 1 debug event has occurred 1 An instruction address compare 1 debug event has occurred while DBCR0[IAC1] = 1, DBCR0[IDM] = 1and DBCR0[EDM] = 0. See Section 9.8.5, “Instruction Address Compare Debug Events,” for more details.
41	IAC2	Instruction Address Compare 2 Debug Event 0 No instruction address compare 2 debug event has occurred 1 An instruction address compare 2 debug event has occurred while DBCR0[IAC2] = 1, DBCR0[IDM] = 1and DBCR0[EDM] = 0. See Section 9.8.5, “Instruction Address Compare Debug Events,” for more details.
42–43	—	Reserved
44	DAC1R	Data Address Compare 1 Read Debug Event 0 No data address compare 1 debug event has occurred 1 A data address compare 1 debug event has occurred while DBCR0[DAC1] = 10 or 11, DBCR0[IDM] = 1and DBCR0[EDM] = 0. See Section 9.8.6, “Data Address Compare Debug Events,” for more details.
45	DAC1W	Data Address Compare 1 Write Debug Event 0 No data address compare 1 debug event has occurred 1 A data address compare 1 debug event has occurred while DBCR0[DAC1] = 01 or 11, DBCR0[IDM] = 1and DBCR0[EDM] = 0. See Section 9.8.6, “Data Address Compare Debug Events,” for more details.
46	DAC2R	Data Address Compare 2 Read Debug Event 0 No data address compare 2 debug event has occurred 1 A data address compare 2 debug event has occurred while DBCR0[DAC2] = 10 or 11, DBCR0[IDM] = 1and DBCR0[EDM] = 0. See Section 9.8.6, “Data Address Compare Debug Events,” for more details.
47	DAC2W	Data Address Compare 2 Write Debug Event 0 No data address compare 2 debug event has occurred 1 A data address compare 2 debug event has occurred while DBCR0[DAC2] = 01 or 11, DBCR0[IDM] = 1 and DBCR0[EDM] = 0. See Section 9.8.6, “Data Address Compare Debug Events,” for more details.
48	RET	Return Debug Event 0 No return debug event has occurred 1 A return debug event has occurred while DBCR0[RET] = 1, DBCR0[IDM] = 1and DBCR0[EDM] = 0. See Section 9.8.11, “Interrupt Return Debug Event,” for more details.

Table 2-45. DBSR/DBSRWR Field Descriptions (continued)

Bits	Name	Description
49–56	—	Reserved
57	CIRPT	Critical Interrupt Taken Debug Event. 0 No critical interrupt taken debug event has occurred 1 A critical interrupt taken debug event has occurred while DBCR0[CIRPT] = 1, DBCR0[IDM] = 1, and DBCR0[EDM] = 0.See Section 9.8.13, “Critical Interrupt Taken Debug Event,” for more details.
58	CRET	Critical Return Debug Event. 0 No critical return debug event has occurred 1 A critical return debug event has occurred while DBCR0[CRET] = 1, DBCR0[IDM] = 1, and DBCR0[EDM] = 0. See Section 9.8.14, “Critical Return Debug Event,” for more details.
59–63	—	Reserved

2.17.7 Instruction Address Compare Registers (IAC1–IAC2)

IAC1–IAC2 are implemented as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* with one exception: when in EDM, software writes to this register are ignored while the e500mc is not halted.

IAC1 and IAC2 are 32-bit registers on e500mc.

The instruction address compare registers (IAC1–IAC2) are described in Figure 2-45.

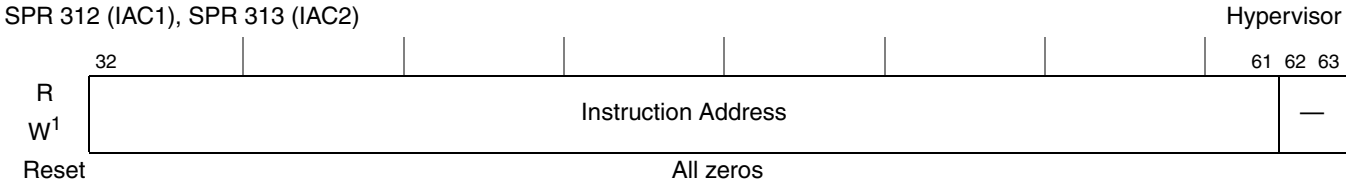


Figure 2-45. Instruction Address Compare Registers (IAC1-IAC2)

[1] When in EDM (DBCR0[EDM] = 1) software writes to this register are ignored while the e500mc is not halted.

2.17.8 Data Address Compare Registers (DAC1–DAC2)

DAC1–DAC2 are implemented as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* with one exception: when in EDM, software writes to this register are ignored while the e500mc is not halted.

DAC1 and DAC2 are 32-bit registers on e500mc.

The data address compare registers (DAC1 and DAC2) are shown in Figure 2-46.

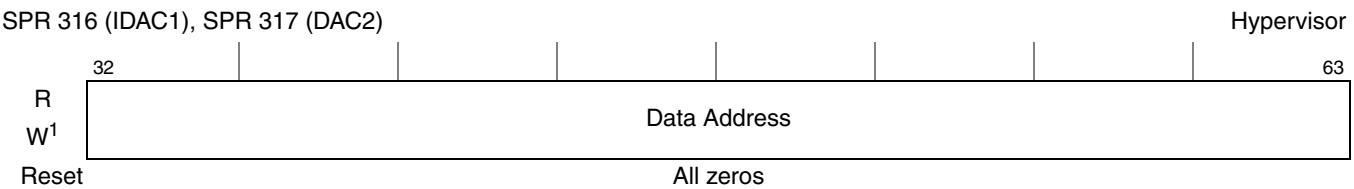


Figure 2-46. Data Address Compare Registers (DAC1–DAC2)

[1] When in EDM (DBCR0[EDM] = 1), software writes to this register are ignored while the e500mc is not halted.

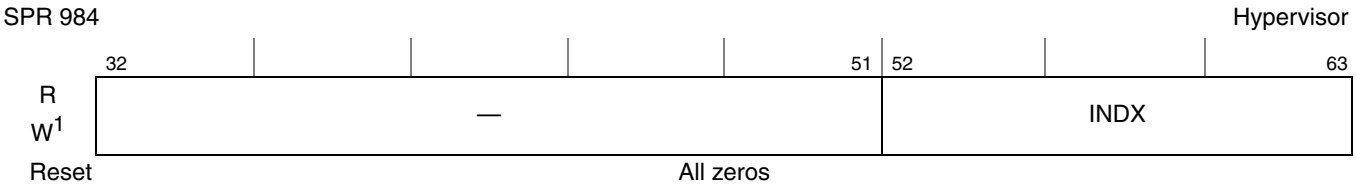
2.17.9 Nexus SPR Access Registers

The architecture defines the Nexus SPR access registers to provide access to the memory-mapped registers implemented as part of the core and described in Section 9.4, "Nexus Registers." The index offset for these registers can be specified in the Nexus SPR configuration register (NSPC) after which access to these registers can be made by using **mtspr** and **mfspr** instructions to read and write the Nexus SPR data register (NSPD).

2.17.9.1 Nexus SPR Configuration Register (NSPC)

The NSPC provides a mechanism for software to access Nexus debug resources (through SPR instructions). Refer to Section 9.9.4.2, "Special-Purpose Register Access (Nexus Only)," for details on accessing Nexus resources through the NSPC register.

This figure shows the Nexus SPR configuration register.



1. When in external debug mode (DBCR0[EDM] = 1) software writes to this register are ignored.

Figure 2-47. Nexus SPR Configuration Register (NSPC)

This table provides the bit definitions for NSPC. See Table 9-23 for the list of the Nexus registers that can be accessed.

Table 2-46. NSPC Field Descriptions

Bits	Name	Description
32–51	—	Reserved
52–63	INDX	Register index[1]

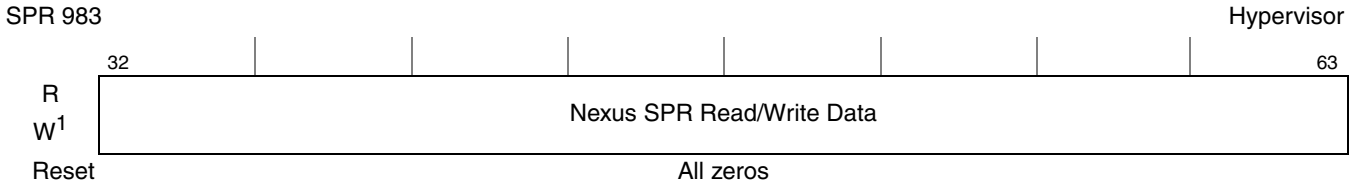
[1] Refer to Table 9-23 for appropriate index values for accessing Nexus registers

2.17.9.2 Nexus SPR Data Register (NSPD)

The NSPD provides a mechanism to transfer data to and from SPR resources. The Nexus resource to be accessed is determined by the programming of the NSPC. For write operations, the write data should be loaded into the NSPD. For read operations, the read data may be acquired from the NSPD.

Writing to the NSPD register requires an **isync** instruction immediately following the **mtspr** to NSPD to ensure that the write is completed.

This figure shows the Nexus SPR data register.



1. When in External Debug Mode (DBCR0[EDM] = 1) software writes to this register are ignored.

Figure 2-48. Nexus SPR Data Register (NSPD)

2.17.10 Debug Event Select Register (DEVENT)

DEVENT allows instrumented software to internally generate signals when an **mtspr** instruction is executed and this register is accessed. The value written to this register determines which processor output signals fire upon access. These signals are used for internal core debug resources, such as the performance monitor, as well as for SoC-level cross-triggering. See the SoC reference manual for more information on use cases.

The upper 8 DEVENT bits also provide the IDTAG used to identify channels within Data Acquisition Messages. See Section 9.10.15, "Data Acquisition," for more detail on the IDTAG.

This figure shows the debug event register.

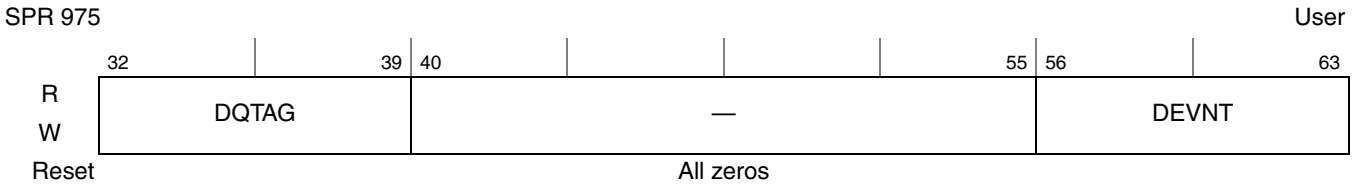


Figure 2-49. Debug Event Register (DEVENT)

This table provides the bit definitions for DEVENT.

Table 2-47. DEVENT Field Descriptions

Bits	Name	Description
32–39	DQTAG	IDTAG channel identifier used in Data Acquisition Messages
40–55	—	Reserved
56–63	DEVNT	Debug Event Signals 00000000 = No signal is asserted xxxxxxx1 = DVT0 is asserted xxxxxx1x = DVT1 is asserted xxxxx1xx = DVT2 is asserted xxxx1xxx = DVT3 is asserted xxx1xxxx = DVT4 is asserted xx1xxxxx = DVT5 is asserted x1xxxxxx = DVT6 is asserted 1xxxxxxx = DVT7 is asserted

2.17.11 Debug Data Acquisition Message Register (DDAM)

DDAM allows instrumented software to generate real-time data acquisition messages (as defined by Nexus) when an **mtspr** instruction is executed and this register is written. See Section 9.10.15, "Data Acquisition," for details.

This figure shows the debug data acquisition message register.

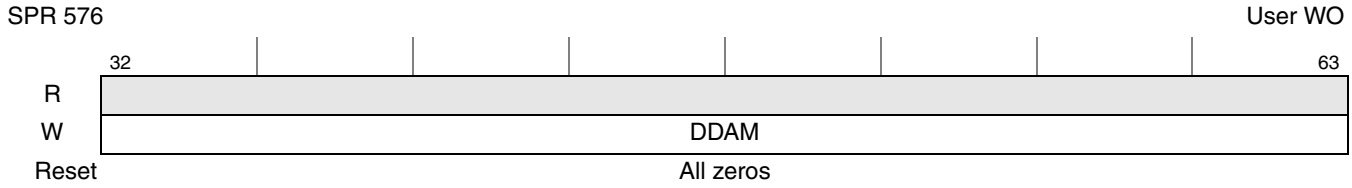


Figure 2-50. Debug Data Acquisition Message Register (DDAM)

This table describes the DDAM bit fields.

Table 2-48. DDAM Field Description

Bits	Name	Description
32–63	DDAM	Data value to be transmitted in a Data Acquisition Message (DQM)

2.17.12 Nexus Process ID Register (NPIDR)

NPIDR allows the full process ID utilized by the OS to be transmitted within Nexus Ownership Trace Messages.

Figure 2-51 shows the Nexus process ID register.

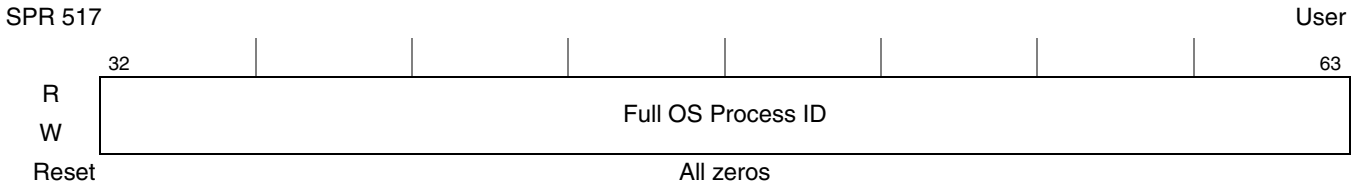


Figure 2-51. Nexus Process ID Register

NOTE

OS accesses to NPIDR must be performed in addition to writes to the PID register used to create translated addresses in the MMU for Nexus messaging.

2.18 Performance Monitor Registers (PMRs)

The performance monitor provides a set of performance monitor registers (PMRs) for defining, enabling, and counting conditions that trigger the performance monitor interrupt. PMRs are defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power*

Architecture® Processors. The performance monitor also defines IVOR35 (see Section 2.9.4, "(Guest) Interrupt Vector Offset Registers (IVORs/GIVORs)") for providing the address of the performance monitor interrupt vector. IVOR35 is described in the interrupt model chapter of the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

PMRs are similar to the SPRs and are accessed by **mtpmr** and **mfpmr**. As shown in Table 2-49, the contents of the PMRs are reflected to a read-only user-level equivalent.

Table 2-49. Performance Monitor Registers

Name	Supervisor		User		Section/Page
	Abbreviation	PMR*n*	Abbreviation	PMR*n*	
Performance monitor counter 0	PMC0	16	UPMC0	0	2.18.4/2-82
Performance monitor counter 1	PMC1	17	UPMC1	1	
Performance monitor counter 2	PMC2	18	UPMC2	2	
Performance monitor counter 3	PMC3	19	UPMC3	3	
Performance monitor local control a0	PMLCa0	144	UPMLCa0	128	2.18.2/2-77
Performance monitor local control a1	PMLCa1	145	UPMLCa1	129	
Performance monitor local control a2	PMLCa2	146	UPMLCa2	130	
Performance monitor local control a3	PMLCa3	147	UPMLCa3	131	
Performance monitor local control b0	PMLCb0	272	UPMLCb0	256	2.18.3/2-79
Performance monitor local control b1	PMLCb1	273	UPMLCb1	257	
Performance monitor local control b2	PMLCb2	274	UPMLCb2	258	
Performance monitor local control b3	PMLCb3	275	UPMLCb3	259	
Performance monitor global control 0	PMGC0	400	UPMGC0	384	2.18.1/2-76

Attempting to access a supervisor PMR from user mode (MSR[PR] = 1), results in a privileged instruction exception. Attempting to access a non-existent PMR in any privilege mode results in an illegal instruction exception.

If MSRP[PMMP] = 1, access to PMRs can cause embedded hypervisor privilege exceptions, or return a value of 0 in the target register. The behavior is described in *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

2.18.1 Global Control Register 0 (PMGC0/UPMGC0)

PMGC0, shown in Figure 2-52, controls all performance monitor counters. PMGC0 contents are reflected to UPMGC0, which is readable by user-level software. The e500mc implements these registers as they are defined by the architecture and as they are described in the *EREF: A Programmer's Reference Manual for*

Freescale Power Architecture® Processors, except for implementation of the following implementation-specific fields:

- Time base selector (TBSEL), bits 51–52. Selects the time base bit that can cause a time base transition event (the event occurs when the selected bit changes from 0 to 1).
- Time base transition event exception enable (TBEE), bit 55.

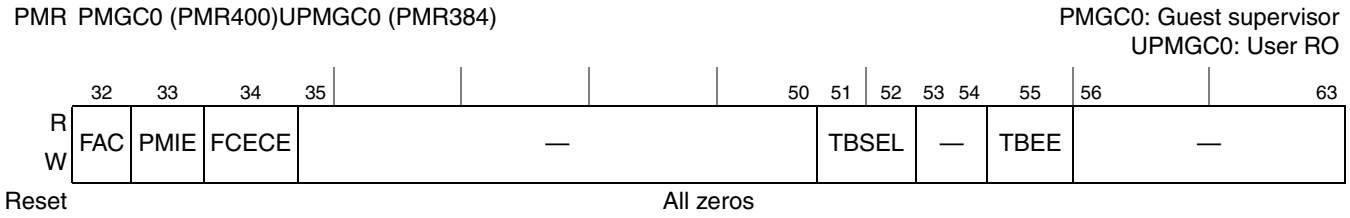


Figure 2-52. Performance Monitor Global Control Register 0 (PMGC0)/ User Performance Monitor Global Control Register 0 (UPMGC0)

PMGC0 is cleared by a hard reset. Reading this register does not change its contents. This table describes the e500mc specific PMGC0 fields.

Table 2-50. PMGC0/UPMGC0 Implementation-Specific Field Descriptions

Bits	Name	Description
51–52	TBSEL	Time base selector. Selects the time base bit that can cause a time base transition event (the event occurs when the selected bit changes from 0 to 1). 00 TB[63] (TBL[63]) 01 TB[55] (TBL[55]) 10 TB[51] (TBL[51]) 11 TB[47] (TBL[47]) Time base transition events can be used to periodically collect information about processor activity. In multiprocessor systems in which TB registers are synchronized among processors, time base transition events can be used to correlate the performance monitor data obtained by the several processors. For this use, software must specify the same TBSEL value for all processors in the system. Because the time-base frequency is implementation-dependent, software should invoke a system service program to obtain the frequency before choosing a value for TBSEL.
55	TBEE	Time base transition event exception enable. 0 Exceptions from time base transition events are disabled. 1 Exceptions from time base transition events are enabled. A time base transition is signaled to the performance monitor if the TB bit specified in PMGC0[TBSEL] changes from 0 to 1. Time base transition events can be used to freeze the counters (PMGC0[FCECE]) or signal an exception (PMGC0[PMIE]). Changing PMGC0[TBSEL] while PMGC0[TBEE] is enabled may cause a false 0 to 1 transition that signals the specified action (freeze, exception) to occur immediately. Although the interrupt signal condition may occur with MSR[EE] = 0, the interrupt cannot be taken until MSR[EE] = 1 or MSR[GS] = 1.

2.18.2 Local Control A Registers (PMLCa0–PMLCa3/UPMLCa0–UPMLCa3)

PMLCa0–PMLCa3 function as event selectors and give local control for the corresponding performance monitor counters. PMLCa*n* works with the corresponding PMLCb*n* register. PMLCa*n* contents are reflected to UPMLCa*n*. The e500mc implements these registers as they are defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* except for the following fields:

- The EVENT field only implements the low order 8 bits of the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* defined field.
- The FCGS0 and FCGS1 fields are not implemented on e500mc Rev 1.x or Rev 2.x.

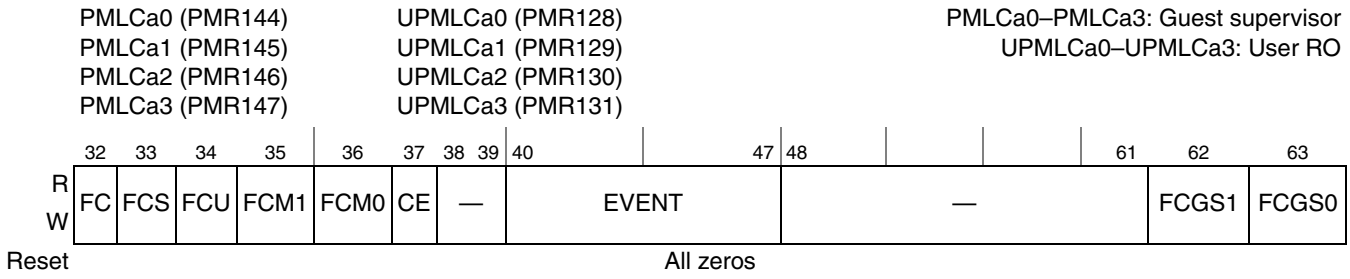


Figure 2-53. Local Control A Registers (PMLCa0–PMLCa3)/ User Local Control A Registers (UPMLCa0–UPMLCa3)

This table describes the PMLCa fields.

Table 2-51. PMLCa0–PMLCa3 Field Descriptions

Bits	Name	Description
32	FC	Freeze counter 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented.
33	FCS	Freeze counter in supervisor state 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PR] = 0.
34	FCU	Freeze counter in user state 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PR] = 1.
35	FCM1	Freeze counter while mark = 1 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PMM] = 1.
36	FCM0	Freeze counter while mark = 0 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PMM] = 0.
37	CE	Condition enable 0 PMC*x* overflow conditions cannot occur. (PMC*x* cannot cause interrupts, cannot freeze counters.) 1 Overflow conditions occur when the most-significant-bit of PMC*x* is equal to one. It is recommended that CE be cleared when counter PMC*x* is selected for chaining.
38–39	—	Reserved
40–47	EVENT	Event selector. Up to 256 events selectable.
48–61	—	Reserved

Table 2-51. PMLCa0–PMLCa3 Field Descriptions (continued)

Bits	Name	Description
62	FCGS1	Freeze counters in guest state. 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[GS] = 1.
63	FCGS0	Freeze counters in hypervisor state. 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[GS] = 0.

2.18.3 Local Control B Registers (PMLCb0–PMLCb3)

Local control B registers (PMLCb0–PMLCb3), shown in Figure 2-54, specify a threshold value and a multiple to apply to a threshold event selected for the corresponding performance monitor counter. For the e500mc, thresholding is supported only for PMC0 and PMC1. PMLCb works with the corresponding PMLCa. PMLCb*n* contents are reflected to UPMLCa*n*. The e500mc implements these registers as they are defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* except for the following e500mc-specific fields:

- TRIGONCTL and TRIGOFFCTL are available for triggering control
- PMCC and PMP are available for triggering status

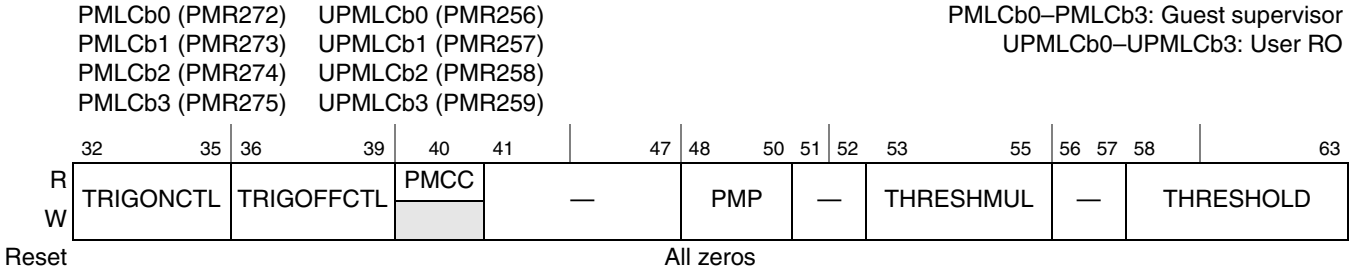


Figure 2-54. Local Control B Registers (PMLCb0–PMLCb3)/ User Local Control B Registers (UPMLCb0–UPMLCb3)

Table 2-52 describes the PMLCb fields.

The implementation specific fields TRIGONCTL and TRIGOFFCTL, provide a method for certain conditions in the processor from the debug facility or the performance monitor facility to start and stop performance monitor counting when a certain programmed condition occurs and the counter is not frozen (for the purposes of this section "frozen" means the counter is frozen by means of either PMLCa*n*[FC] or PMGC0[FAC]). The trigger state is either set to ON or OFF depending on how the controls are programmed and when the programmed conditions occur in the processor. When the trigger state is ON, events are enabled for counting in PMC*n* if counting is enabled by all other performance monitor controls. If the trigger state is OFF, counting is disabled for PMC*n*. For both controls, the following applies to how the trigger state is determined:

- When the counter is frozen by means of either PMLCa*n*[FC] or PMGC0[FAC] being set to 1, the trigger state is set to OFF. The trigger state remains off until the counter is unfrozen and a subsequent condition sets the trigger state to ON.

- If TRIGONCTL = 0b0000, the trigger state is always set to ON when the counter is not frozen. This setting is used to essentially make triggers inactive and all other performance monitor controls determine whether events are counted.
- If a condition occurs that is programmed via TRIGONCTL and the counter is not frozen, the trigger state is set to ON.
- If a condition occurs that is programmed via TRIGOFFCTL and the counter is not frozen, the trigger state is set to OFF.
- Other methods of freezing the PMC*n* from counter other than PMLCa*n*[FC] or PMGC0[FAC] have no effect on the trigger state, although such methods can prevent the counter from counting. That is, the trigger state may be ON, but the PMC*n* is not counting events because it is frozen from some other method.

Table 2-52. PMLCb0–PMLCb3 Field Descriptions

Bits	Name	Description
32–35	TRIGONCTL	Counter Trigger ON control. 0000 No ON triggering active. This means that the counter is always considered to be triggered ON when it is not frozen. 0001 Trigger ON when rise of PMC*n* Qual Pin detected 0010 Trigger ON when previous Performance Monitor Counter overflow condition (bit 32 only) 0011 Trigger ON when IAC1 match (only requires the debug condition, not the event) 0100 Trigger ON when IAC2 match (only requires the debug condition, not the event) 0101 Trigger ON when DAC1 match (only requires the debug condition, not the event) 0110 Trigger ON when DAC2 match (only requires the debug condition, not the event) 0111–1110 Trigger ON when DVT*n* asserted 1111 Reserved **Note:** DVT*n* (DVT0, DVT1, .. DVT7) are asserted by writing the DEVENT register. See Section 2.17.10, "Debug Event Select Register (DEVENT)." The counter trigger ON control uses certain conditions in the processor as a signal to start counting when those conditions occur. Triggers associated with debug events require only the debug condition to be present, and does not require that the debug event occurs. For example, an IAC1 match occurs which does not result in a debug event because DBCR0[IDM] is not set, still causes counting to begin if the appropriate trigger ON control is set. For a graphic representation of performance monitor counter controls see Figure 9-24.
36–39	TRIGOFFCTL	Counter Trigger OFF control 0000 Never trigger OFF due to a condition. 0001 Trigger OFF when fall of PMCn Qual Pin 0010 Trigger OFF when previous Performance Monitor Counter overflow condition (bit 32 only) 0011 Trigger OFF when IAC1 match (only requires the debug condition, not the event) 0100 Trigger OFF when IAC2 match (only requires the debug condition, not the event) 0101 Trigger OFF when DAC1 match (only requires the debug condition, not the event) 0110 Trigger OFF when DAC2 match (only requires the debug condition, not the event) 0111–1110 Trigger OFF when DVT*n* asserted 1111 Reserved **Note:** DVT*n* (DVT0, DVT1, .. DVT7) are asserted by writing the DEVENT register. See Section 2.17.10, "Debug Event Select Register (DEVENT)." The counter trigger OFF control uses certain conditions in the processor as a signal to stop counting when those conditions occur. Triggers associated with debug events require only the debug condition to be present, and does not require that the debug event occurs. For example, an IAC1 match occurs which does not result in a debug event because DBCR0[IDM] is not set, still causes counting to stop if the appropriate trigger OFF control is set. For a graphic representation of performance monitor counter controls see Figure 9-24.
40	PMCC	PMC*n* trigger state. 0 PMC*n* trigger state is OFF. 1 PMC*n* trigger state is ON. **Note:** This is a status bit which shows the trigger state controlled by TRIGONCTL and TRIGOFFCTL. When PMCC = 1, a PMC*n* may still not be counting if it is frozen by means other than PMLCa*n*[FC] or PMGC0[FAC].
41–47	—	Reserved

Table 2-52. PMLCb0–PMLCb3 Field Descriptions (continued)

Bits	Name	Description
48–50	PMP	Performance Monitor Overflow Periodicity Select [1] 000 Performance Monitor Watchpoint (PMW*n*) triggers on any change to counter bit 32 (period = 2^{31}) 001 Performance Monitor Watchpoint (PMW*n*) triggers on any change to counter bit 43 (period = 2^{20}) 010 Performance Monitor Watchpoint (PMW*n*) triggers on any change to counter bit 49 (period = 2^{14}) 011 Performance Monitor Watchpoint (PMW*n*) triggers on any change to counter bit 55 (period = 2^{8}) 100 Performance Monitor Watchpoint (PMW*n*) triggers on any change to counter bit 59 (period = 2^{4}) 101 Performance Monitor Watchpoint (PMW*n*) triggers on any change to counter bit 61 (period = 2^{2}) 110 Performance Monitor Watchpoint (PMW*n*) triggers on any change to counter bit 62 (period = 2^{1}) 111 Performance Monitor Watchpoint (PMW*n*) triggers on any change to counter bit 63 (period = 2^{0})
51–52	—	Reserved
53–55	THRESHMUL	Threshold multiple 000 Threshold field is multiplied by 1 (PMLCb*n*[THRESHOLD] × 1) 001 Threshold field is multiplied by 2 (PMLCb*n*[THRESHOLD] × 2) 010 Threshold field is multiplied by 4 (PMLCb*n*[THRESHOLD] × 4) 011 Threshold field is multiplied by 8 (PMLCb*n*[THRESHOLD] × 8) 100 Threshold field is multiplied by 16 (PMLCb*n*[THRESHOLD] × 16) 101 Threshold field is multiplied by 32 (PMLCb*n*[THRESHOLD] × 32) 110 Threshold field is multiplied by 64 (PMLCb*n*[THRESHOLD] × 64) 111 Threshold field is multiplied by 128 (PMLCb*n*[THRESHOLD] × 128)
56–57	—	Reserved
58–63	THRESHOLD	Threshold. Only events that exceed this value are counted. Events to which a threshold value applies are implementation-dependent as are the dimension (for example duration in cycles) and the granularity with which the threshold value is interpreted. By varying the threshold value, software can profile event characteristics. For example, if PMC1 is configured to count cache misses that last longer than the threshold value, software can obtain the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.

[1] Performance Monitor Counter overflow generates a watchpoint (PMWn) that can be used for triggering or to generate Watchpoint Messages (if enabled).

2.18.4 Performance Monitor Counter Registers (PMC0–PMC3/UPMC0–UPMC3)

The PMCs, shown in Figure 2-55, are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count 128 events. The e500mc implements these registers as they are defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

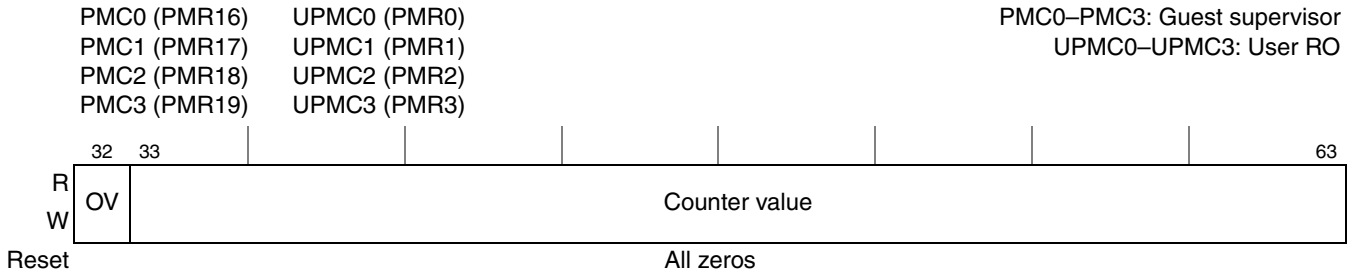


Figure 2-55. Performance Monitor Counter Registers (PMC0–PMC3)/ User Performance Monitor Counter Registers (UPMC0–UPMC3)

This table describes the PMC register fields.

Table 2-53. PMC0–PMC3 Field Descriptions

Bits	Name	Description
32	OV	Overflow. When this bit is set, it indicates this counter reaches its maximum value.
33–63	Counter Value	Indicates the number of occurrences of the specified event.

The minimum counter value is 0x0000_0000; 4,294,967,295 (0xFFFF_FFFF) is the maximum. A counter can increment by 0, 1, 2, 3, or 4 up to the maximum value and then wrap to the minimum value.

A counter enters overflow state when the high-order bit is set by entering the overflow state at the halfway point between the minimum and maximum values. A performance monitor interrupt handler can easily identify overflowed counters, even if the interrupt is masked for many cycles (during which the counters may continue incrementing). A high-order bit is set normally only when the counter increments from a value below 2,147,483,648 (0x8000_0000) to a value greater than or equal to 2,147,483,648 (0x8000_0000).

NOTE

Initializing PMCs to overflowed values is strongly discouraged. If an overflowed value is loaded into a PMC*n* that held a non-overflowed value (and PMGC0[PMIE], PMLCa*n*[CE], and (MSR[EE] or MSR[GS]) are set), an interrupt is generated before any events are counted.

The response to an overflow depends on the configuration, as follows:

- If PMLCa*n*[CE] is clear, no special actions occur on overflow: the counter continues incrementing, and no exception is signaled.
- If PMLCa*n*[CE] and PMGC0[FCECE] are set, all counters are frozen when PMC*n* overflows.
- If PMLCa*n*[CE] and PMGC0[PMIE] are set, an exception is signaled when PMC*n* reaches overflow. Interrupts are masked by when MSR[EE] and MSR[GS] are both 0. An exception may be signaled while the interrupt is masked by MSR[EE] and MSR[GS], but the interrupt is not taken until it is fully enabled and only if the overflow condition is still present and the configuration has not been changed in the meantime to disable the exception.

However, if MSR[EE] and MSR[GS] remain 0 until after the counter leaves the overflow state (msb becomes 0), or if MSR[EE] and MSR[GS] remain 0 until after PMLCa*n*[CE] or PMGC0[PMIE] are cleared, the exception is not signaled.

The following sequence is recommended for setting counter values and configurations:

1. Set PMGC0[FAC] to freeze the counters.
2. Using **mtpmr** instructions, initialize counters and configure control registers.
3. Release the counters by clearing PMGC0[FAC] with a final **mtpmr**.

Software is expected to use **mtpmr** to explicitly set PMCs to non-overflowed values. Setting an overflowed value may cause an erroneous exception. For example, if both PMGC0[PMIE] and PMLCa*n*[CE] are set and the **mtpmr** loads an overflowed value into PMC*n*, an interrupt may be generated without an event counting having taken place.

Chapter 3
Instruction Model

This chapter provides a listing and general description of instructions implemented on the e500mc processor cores grouping the instructions by general functionality. It provides the syntax and briefly describes the functionality as defined by the architecture. Full descriptions of these instructions are provided in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

3.1 Instruction Model Overview

This chapter provides information about the instruction set as implemented on the e500mc, which is an implementation of the 32-bit Power ISA.The e500mc implements extensions that define additional instructions, registers, and interrupts. The architecture defines several instructions in a general way, leaving some details of the execution up to the implementation. Any such details are described in this chapter.

3.1.1 Supported Power ISA Categories and Unsupported Instructions

The e500mc implements the following categories as defined by *Power ISA 2.06*:

- Base
- Embedded
- Alternate Time Base
- Cache Specification
- Decorated Storage
- Embedded.Enhanced Debug
- Embedded.External PID
- Embedded.Hypervisor
- Embedded.Little-Endian
- Embedded.Performance Monitor
- Embedded.Processor Control
- Embedded.Cache Locking
- External Proxy
- Floating Point and Floating Point.Record
- Memory Coherence
- Store Conditional Page Mobility
- Wait

Table 3-1 lists Power ISA 2.06 instructions defined in the above categories which are not supported on the e500mc. Attempting to execute unsupported instructions results in an illegal instruction exception-type program exception.

Table 3-1. Unsupported Power ISA 2.06 Instructions (by category)

Category	Mnemonic	Name	Notes
Base	**cmpb**	Compare Bytes	—
Base	**divwe[o][.]**	Divide Word Extended	—
Base	**divweu[o][.]**	Divide Word Extended Unsigned	—
Base	**lbarx**	Load Byte and Reserve Indexed	—
Base	**lharx**	Load Halfword and Reserve Indexed	—
Base	**popcntb**	Population Count Byte	—
Base	**popcntd**	Population Count Doubleword	—
Base	**popcntw**	Population Count Word	—
Base	**prtyw**	Parity Word	—
Base	**stbcx.**	Store Byte Conditional Indexed	—
Base	**sthcx.**	Store Halfword Conditional Indexed	—
Embedded.External PID	**evlddepx**	Vector Load Doubleword into Doubleword by External Process ID Indexed	Category SPE not supported
Embedded.External PID	**evstddepx**	Vector Store Doubleword into Doubleword by External Process ID Indexed	Category SPE not supported
Embedded.External PID	**lvepx**	Load Vector by External Process ID Indexed	Category Vector not supported
Embedded.External PID	**lvepxl**	Load Vector by External Process ID Indexed LRU	Category Vector not supported
Embedded.External PID	**stvepx**	Store Vector by External Process ID Indexed	Category Vector not supported
Embedded.External PID	**stvepxl**	Store Vector by External Process ID Indexed LRU	Category Vector not supported
Embedded.External PID	**ldepx**	Load Doubleword by External Process ID Indexed	Category 64-bit not supported
Embedded.External PID	**stdepx**	Store Doubleword by External Process ID Indexed	Category 64-bit not supported
Floating Point	**fcfid[.]**	Floating Convert From Integer Doubleword	—
Floating Point	**fcfids[.]**	Floating Convert From Integer Doubleword Single	—
Floating Point	**fcfidu[.]**	Floating Convert From Integer Doubleword Unsigned	—
Floating Point	**fcfidus[.]**	Floating Convert From Integer Doubleword Unsigned Single	—
Floating Point	**fcpsgn[.]**	Floating Copy Sign	—
Floating Point	**fctid[.]**	Floating Convert To Integer Doubleword	—
Floating Point	**fctidu[.]**	Floating Convert To Integer Doubleword Unsigned	—
Floating Point	**fctiduz[.]**	Floating Convert To Integer Doubleword Unsigned with round toward Zero	—
Floating Point	**fctidz[.]**	Floating Convert To Integer Doubleword with round toward Zero	—

Table 3-1. Unsupported Power ISA 2.06 Instructions (by category) (continued)

Category	Mnemonic	Name	Notes
Floating Point	**fctiwu[.]**	Floating Convert To Integer Word Unsigned	—
Floating Point	**fctiwuz[.]**	Floating Convert To Integer Word Unsigned with round towards Zero	—
Floating Point	**fre**	Floating Reciprocal Estimate	—
Floating Point	**frim[.]**	Floating Round to Integer Minus	—
Floating Point	**frin[.]**	Floating Round to Integer Nearest	—
Floating Point	**frip[.]**	Floating Round to Integer Plus	—
Floating Point	**friz[.]**	Floating Round to Integer Toward Zero	—
Floating Point	**frsqrtes[.]**	Floating Reciprocal Square Root Estimate Single	—
Floating Point	**fsqrt[s][.]**	Floating Square Root [Single]	—
Floating Point	**ftdiv[.]**	Floating Test for software Divide	—
Floating Point	**ftsqrt[.]**	Floating Test for software Square Root	—
Floating Point	**lfiwax**	Load Floating-Point as Integer Word Algebraic Indexed	—
Floating Point	**lfiwzx**	Load Floating-Point as Integer Word and Zero Indexed	—
Floating Point	**mtfsfi[.]** (W field)	Move to FPSCR Immediate	W field is not implemented. Always behaves as if W = 0.
Floating Point	**mtfsf[.]** (W and L fields)	Move to FPSCR	W and L fields are not implemented. Always behaves as if W = L = 0.
Wait	**wait** (WC field)	Wait	WC field is not implemented. Always behaves as Wait 0.

3.2 Computation Mode

The e500mc is a 32-bit implementation of Power ISA 2.06 and supports only 32-bit GPRs and 32-bit mode of execution.

3.3 Instruction Set Summary

The e500mc instructions are presented in the following functional categories:

- Integer instructions
  These include arithmetic and logical instructions. For more information, see Section 3.4.3.1, “Integer Instructions.”
- Floating-point instructions
  These include floating-point arithmetic and other floating-point instructions.
- Load and store instructions
  See Section 3.4.3.2, “Load and Store Instructions.”
- Flow control instructions
  These include branching instructions, CR logical instructions, trap instructions, and other

instructions that affect the instruction flow. See Section 3.4.5, “Branch and Flow Control Instructions.”

- Processor control instructions
  These instructions are used for performing various tasks associated with moving data to and from special registers, system linkage instructions, etc. See Section 3.4.6, “Processor Control Instructions.”
- Memory synchronization instructions
  These instructions are used for memory synchronizing. See Section 3.4.8, “Memory Synchronization Instructions.”
- Memory control instructions
  These instructions provide control of caches and TLBs. See Section 3.4.10, “Memory Control Instructions,” and Section 3.4.11.3, “Supervisor-Level Memory Control Instructions.”

Note that instruction groupings used here do not indicate the execution unit that processes a particular instruction or group of instructions. This information, which is useful for scheduling instructions most effectively, is provided in Chapter 10, “Execution Timing.”

Instructions are four bytes long and are word-aligned. Byte, halfword, word loads and stores occur between memory and a set of thirty-two 32-bit general-purpose registers (GPRs).

Integer instructions operate on word operands that specify GPRs as source and destination registers. Floating-point instructions operate on doubleword operands, which may contain single- or double-precision values, and use thirty-two 64-bit floating-point registers (FPRs) as source and destination registers.

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

The description of each instruction includes the mnemonic and a formatted list of operands. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the frequently used instructions (see Appendix B, “Simplified Mnemonics,” for a complete list). Programs written to be portable across the various assemblers for the Power ISA should not assume the existence of mnemonics not described in that document.

3.3.1 Instruction Decoding

Reserved fields in instructions are ignored by e500mc. If an instruction contains a defined field for which some values of that field are reserved, and that instruction is coded with those reserve values, that instruction form is considered an invalid form. Execution of an invalid form instruction is boundedly undefined.

3.3.2 Definition of Boundedly Undefined

When a boundedly undefined execution of an instruction takes place, the resulting undefined results are bounded in that a spurious change in privilege state is not allowed, and the level of privilege exercised by the program in relation to memory access and other system resources cannot be exceeded. Boundedly

undefined results for a given instruction can vary between implementations and between execution attempts in the same implementation.

3.3.3 Synchronization Requirements

This section discusses synchronization requirements for special registers, certain instructions, and TLBs. The synchronization described in this section refers to the state of the processor that is performing the synchronization.

Changing a value in certain system registers and invalidating TLB entries can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed. For example, changing MSR[IS] from 0 to 1 has the side effect of changing address space. These effects need not occur in program order (that is, the strict order in which they occur in the program) and may require explicit synchronization by software. When multiple changes are made that affect context to different values, even within the same register, those changes are not guaranteed to occur at the same time unless the instruction itself is context synchronizing. For example, changing both MSR[IS] and MSR[GS] with the same **mtmsr** instruction causes multiple changes to how fetched instructions are translated. The change to MSR[IS] may occur in a different cycle than MSR[GS], but both are guaranteed to be complete when a context synchronizing event occurs.

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed, is called a context-altering instruction. This section covers all of the context-altering instructions. The software synchronization required for each is shown in Table 3-2 and Table 3-3. Instructions that are not listed do not require explicit synchronization.

The notation “CSI” in the tables means any context-synchronizing instruction (**sc**, **isync**, **rfi**, **rfgi**, **rfci**, **rfdi**, or **rfmci**). Any interrupt can be used instead of a context-synchronizing instruction to synchronize instructions. If it is, references in this section to the synchronizing instruction should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required either before or after a context-altering instruction, the phrase ‘the synchronizing instruction before (or after) the context-altering instruction’ should be interpreted as meaning the context-altering instruction itself.

The synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. The synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

Care must be taken when altering context associated with instruction fetch and instruction address translation. Altering MSR[IS], MSR[GS], LPIDR, and PID can cause an implicit branch, where the change in translation or how instructions are fetched causes the processor to fetch instructions from a different real address than what would have resulted if the context was not changed. Implicit branches are not supported by the architecture and it is recommended that MSR[IS] and MSR[GS] context changes be performed through a return from interrupt instruction (**rfi**, **rfgi**, **rfci**, **rfdi**, or **rfmci**) which changes all the MSR context atomically and is completely context synchronizing.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

Sometimes advantage can be taken of the fact that certain instructions that occur naturally in the program, such as the **rfi** at the end of an interrupt handler, provide the required synchronization.

No software synchronization is required before altering the MSR because **mtmsr** is execution synchronizing. No software synchronization is required before most other alterations shown in Table 3-2, because all instructions before the context-altering instruction are fetched and decoded before the context-altering instruction is executed. (The processor must determine whether any of the preceding instructions are context-synchronizing.)

Table 3-2 identifies the software synchronization requirements for data access for context-altering instructions that require synchronization.

Table 3-2. Data Access Synchronization Requirements

Context Altering Instruction or Event	Required Before	Required After	Notes
mfspr (L1CSR0, L1CSR1)	**sync**	None	[1]
mtmsr (DE)	None	CSI	—
mtmsr (DS)	None	CSI	—
mtmsr (GS)	None	CSI	—
mtmsr (ME)	None	CSI	[2]
mtmsr (PR)	None	CSI	—
mtpmr (all)	None	CSI	—
mtspr (EPLC)	None	CSI	—
mtspr (EPSC)	None	CSI	—
mtspr (L1CSR0, L1CSR1)	**sync** followed by **isync**	**isync**	—
mtspr (L1CSR2)	**sync** followed by **isync**	**isync** followed by **sync**[3]	—
mtspr (L2CSR0)	**sync** followed by **isync**	**isync**	—
mtspr (L2CSR1)	**sync** followed by **isync**	**isync** followed by **sync**[3]	—
mtspr (LPIDR)	CSI	CSI	—
mtspr (PID)	CSI	CSI	—
tlbivax	CSI	**sync** followed by CSI	[4,5,6]
tlbilx	CSI	CSI	[4,5]
tlbwe	CSI	CSI	[4,5]

[1] A **sync** prior to reading L1CSR0 or L1CSR1 is required to examine any cache locking status from prior cache locking operations. The **sync** ensures that any previous cache locking operations have completed prior to reading the status.

[2] A context-synchronizing instruction is required after altering MSR[ME] to ensure that the alteration takes effect for subsequent machine check interrupts, which may not be recoverable and therefore may not be context-synchronizing.

[3] The additional **sync** after the **mtspr** is done is required if software is turning off stashing by setting the stash ID field of the register to zero. The **sync** ensures that any pending stash operations have finished.

[4] For data accesses, the context-synchronizing instruction before **tlbwe**, **tlbilx**, or **tlbivax** ensures that all memory accesses due to preceding instructions have completed to a point at which they have reported all exceptions they cause.

5 The context-synchronizing instruction after **tlbwe**, **tlbilx**, or **tlbivax** ensures that subsequent accesses (data and instruction) use the updated value in any TLB entries affected. It does not ensure that all accesses previously translated by TLB entries being updated have completed with respect to memory; if these completions must be ensured, **tlbwe**, **tlbilx**, or **tlbivax** must be followed by an **sync** and by a context-synchronizing instruction.

6 To insure that all TLB invalidations are completed and seen in all processors in the coherence domain, the global invalidation requires that a **tlbsync** be executed after the **tlbivax** as follows: **tlbivax**; **sync**; **tlbsync**; **sync**; **isync**. For the e500mc, this code should be protected by a mutual exclusion lock such that only one processor at a time is executing this sequence as multiple **tlbsync** operations on the CoreNet interface may cause the integrated device to hang.

Table 3-3 identifies the software synchronization requirements for instruction fetch and/or execution for context-altering instructions which require synchronization.

Table 3-3. Instruction Fetch and/or Execution Synchronization Requirements

Context Altering Instruction or Event	Required Before	Required After	Notes
mtmsr (DE)	None	CSI	—
mtmsr (FE0)	None	CSI	—
mtmsr (FE1)	None	CSI	—
mtmsr (FP)	None	CSI	—
mtmsr (IS)	None	CSI	—
mtmsr (GS)	None	CSI	—
mtmsr (PR)	None	CSI	—
mtpmr (all)	None	CSI	—
mtspr (L1CSR0, L1CSR1, L1CSR2)	**sync** followed by **isync**	**isync**	—
mtspr (L2CSR0, L2CSR1)	**sync** followed by **isync**	**isync**	—
mtspr (LPIDR)	None	CSI	—
mtspr (MAS*n*)	None	**isync**	1
mtspr (PID)	None	CSI	—
tlbivax	None	CSI	2,3
tlbilx	None	CSI	2
tlbwe	None	CSI	2

1 Architecturally, MAS registers changes require an **isync** before subsequent instructions that use those updated values such as a **tlbwe**, **tlbre**, **tlbilx**, **tlbsx**, and **tlbivax**. Typically software does several MAS updates and then performs a single **isync** prior to executing the TLB management instruction. Currently, e500mc does not require such synchronization because the **mtspr** and the TLB management instructions both internally use the same synchronization method. If software choses not to execute the **isync** it should be aware that the internal synchronization may change in future cores or even in a future revision of e500mc.

2 The context-synchronizing instruction after **tlbwe**, **tlbilx**, or **tlbivax** ensures that subsequent accesses (data and instruction) use the updated value in any TLB entries affected. It does not ensure that all accesses previously translated by TLB entries being updated have completed with respect to memory; if these completions must be ensured, **tlbwe**, **tlbilx**, or **tlbivax** must be followed by an **sync** and by a context-synchronizing instruction.

3 To insure that all TLB invalidations are completed and seen in all processors in the coherence domain, the global invalidation requires that a **tlbsync** be executed after the **tlbivax** as follows: **tlbivax**; **sync**; **tlbsync**; **sync**; **isync**. For the e500mc, this code should be protected by a mutual exclusion lock such that only one processor at a time is executing this sequence as multiple **tlbsync** operations on the CoreNet interface may cause the integrated device to hang.

Table 3-4 identifies the software synchronization requirements for non context-altering instructions that require synchronization.

Table 3-4. Special Synchronization Requirements

Context Altering Instruction or Event	Required Before	Required Immediately After	Notes
mtspr (BUCSR)	None	**isync**	—
mtspr (DAC*n*)	None	**isync** followed by changing MSR[DE] from 0 to 1	1
mtspr (DBCR*n*)	None	**isync** followed by changing MSR[DE] from 0 to 1	1
mtspr (DBSR)	None	**isync** followed by changing MSR[DE] from 0 to 1	1
mtspr (DBSRWR)	None	**isync** followed by changing MSR[DE] from 0 to 1	1
mtspr (EPCR[DUVD])	None	**isync** followed by changing MSR[DE] from 0 to 1	1,2
mtspr (HID*n*)	**msync** followed by **isync**	**isync**	—
mtspr (IAC*n*)	None	**isync** followed by changing MSR[DE] from 0 to 1	1
mtspr (L2ERR*)*	**msync** followed by **isync**	**isync**	—
mtspr (MMUCSR0)	None	**isync**	—
mtspr (NSPD)	None	**isync**	—

1 Synchronization requirements for changing any debug facility registers require that the changes be followed by an **isync** and a transition of MSR[DE] from 0 to 1 before the results of the changes are guaranteed to be seen. Normally changes to the debug registers occurs in the debug interrupt routine when MSR[DE] is 0 and the subsequent return from the debug routine is likely to set MSR[DE] back to 1 which accomplishes the required synchronization. Software should only make changes to the debug facility registers when MSR[DE] = 0.

2 Note that the special synchronization requirement applies only to changes to EPCR[DUVD]. If this bit is not changed, the synchronization requirements for EPCR is as described in the data or instruction execution tables above.

3.3.3.1 Synchronization with tlbwe, tlbivax, and tlbilx Instructions

The following sequence shows why, for data accesses, all memory accesses due to instructions before the **tlbwe** or **tlbivax** must complete to a point at which they have reported any exceptions. Assume valid TLB entries exist for the target memory location when the sequence starts.

1. A program issues a load or store to a page.
2. The same program executes **tlbwe**, **tlbilx**, or **tlbivax** that invalidates the corresponding TLB entry.
3. The load or store instruction finally executes, and gets a TLB miss exception.

The TLB miss exception is semantically incorrect. To prevent it, a context-synchronizing instruction must be executed between steps 1 and 2.

The **tlbilx** instruction requires the same local-processor synchronization as **tlbivax**, but not the cross-processor synchronization (that is, it does not require a **tlbsync**).

3.3.3.2 Context Synchronization

An instruction or event is context synchronizing if it satisfies the requirements listed below. Context-synchronizing operations include instructions **isync**, **sc**, **rfi**, **rfci**, **rfmci**, **rfdi**, **rfgi**, **ehpriv,** and most interrupts. The *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* describes context synchronization in detail.

1. The operation is not initiated or, in the case of **isync**, does not complete until all executing instructions complete to a point at which they have reported all exceptions they cause.
2. Instructions that precede the operation execute in the context (including such parameters as privilege level, address space, and memory protection) in which they were initiated.
3. If the operation directly causes an interrupt (for example, **sc** directly causes a system call interrupt) or is an interrupt, the operation is not initiated until no interrupt-causing exception exists having higher priority than the exception associated with the interrupt.See Section 4.12, "Exception Priorities."
4. Instructions that follow the operation are fetched and executed in the context established by the operation as required by the sequential execution model. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them speculatively may also be discarded, except as described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.)

As described in Section 3.3.3.3, "Execution Synchronization," a context-synchronizing operation is necessarily execution synchronizing. Unlike **sync** (**msync**) and **mbar**, such operations do not affect the order of memory accesses with respect to other mechanisms.

3.3.3.3 Execution Synchronization

An instruction is execution synchronizing if it satisfies items 1 and 2 of the definition of context synchronization (see Section 3.3.3.2, "Context Synchronization"). **sync** (**msync**) is treated like **isync** with respect to item 1 (that is, the conditions described in item 1 apply to completion of **sync**). Execution synchronizing instructions include **sync**, **mtmsr**, **wrtee**, and **wrteei**. All context-synchronizing instructions are execution synchronizing.

Unlike a context-synchronizing operation, an execution synchronizing instruction need not ensure that instructions following it execute in the context established by that execution synchronizing instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context-synchronizing operation.

3.3.3.4 Instruction-Related Interrupts

Interrupts are caused either directly by the execution of an instruction or by an asynchronous event. In either case, an exception may cause one of several types of interrupts to be invoked. For example, an attempt by an application program to execute a privileged instruction causes a privileged instruction exception-type program interrupt. Such exceptions and interrupts for the e500mc instructions are described in Section 4.6, "Exceptions."

3.4 Instruction Set Overview

This section provides an overview of the instructions implemented in the e500mc and highlights any special information with respect to how the e500mc implements a particular instruction.

3.4.1 Record and Overflow Forms

Note that some instructions have record and/or overflow forms that have the following features:

- CR update for integer instructions
  The dot (**.**) suffix on the mnemonic for integer computation instructions enables the update of the CR0 field. CR0 is updated based on the signed comparison of the result to 0.
- Integer overflow option
  The **o** suffix indicates that the overflow bit in the XER is enabled. In 32-bit mode, overflow (XER[OV]) is set if the carryout of bit 32 is not equal to the carryout of bit 33 in the final result of the operation. Summary overflow (XER[SOV]) is a sticky bit that is set when XER[OV] is set.
- CR update for floating-point instructions
  The dot (**.**) suffix on the mnemonic for floating-point computation instructions enables the update of the CR1 field. CR1 is updated with the exception status copied from bits FPSCR[32:35].
- CR update for store conditional instructions
  Store conditional instructions always include the dot (**.**) suffix and update CR0 based on whether the store was performed.

3.4.2 Effective Address Computation

Load and store operations (as well as **tlbivax**, **tlbilx**, cache locking, and cache management instructions) generate effective addresses used to determine the address where a storage operation is to be performed. There are several different forms of effective address generation and some instructions such as integer load and store instructions provide all such forms. The effective address calculation modes are as follows:

- Register indirect with immediate index addressing. The EA is generated by adding the sign-extended 16-bit immediate index (**d** operand) to the contents of the GPR specified by **r**A. If **r**A specifies **r0**, a value of zero is added to the index. Instruction descriptions show this option as (**r**A|0).
- Register indirect with index addressing. The EA is formed by adding the contents of two GPRs specified as operands **r**A and **r**B. A zero in place of the **r**A operand causes a zero to be added to the contents of the GPR specified in operand **r**B

Register indirect addressing. The GPR specified by the **r**B operand contains the EA.

For more information, see *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

3.4.3 User-Level Instructions

This section discusses the user-level instructions.

3.4.3.1 Integer Instructions

This section describes the integer instructions. These consist of the following:

- Integer Arithmetic Instructions
- Integer Compare Instructions
- Integer Logical Instructions
- Integer Rotate and Shift Instructions

Integer instructions use the content of the GPRs as source operands and place results into GPRs and the XER and CR fields.

3.4.3.1.1 Integer Arithmetic Instructions

This table lists the Power ISA integer arithmetic instructions implemented on the e500mc.

Table 3-5. Integer Arithmetic Instructions

Name	Mnemonic	Syntax
Add	**add** (**add. addo addo.**)	**r**D,**r**A,**r**B
Add Carrying	**addc** (**addc. addco addco.**)	**r**D,**r**A,**r**B
Add Extended	**adde** (**adde. addeo addeo.**)	**r**D,**r**A,**r**B
Add Immediate	**addi**	**r**D,**r**A,SIMM
Add Immediate Carrying	**addic**	**r**D,**r**A,SIMM
Add Immediate Carrying and Record	**addic.**	**r**D,**r**A,SIMM
Add Immediate Shifted	**addis**	**r**D,**r**A,SIMM
Add to Minus One Extended	**addme** (**addme. addmeo addmeo.**)	**r**D,**r**A
Add to Zero Extended	**addze** (**addze. addzeo addzeo.**)	**r**D,**r**A
Divide Word	**divw** (**divw. divwo divwo.**)	**r**D,**r**A,**r**B
Divide Word Unsigned	**divwu divwu. divwuo divwuo.**	**r**D,**r**A,**r**B
Multiply High Word	**mulhw** (**mulhw.**)	**r**D,**r**A,**r**B
Multiply High Word Unsigned	**mulhwu** (**mulhwu.**)	**r**D,**r**A,**r**B
Multiply Low Immediate	**mulli**	**r**D,**r**A,SIMM
Multiply Low Word	**mullw** (**mullw. mullwo mullwo.**)	**r**D,**r**A,**r**B
Negate	**neg** (**neg. nego nego.**)	**r**D,**r**A
Subtract From	**subf** (**subf. subfo subfo.**)	**r**D,**r**A,**r**B
Subtract from Carrying	**subfc** (**subfc. subfco subfco.**)	**r**D,**r**A,**r**B
Subtract from Extended	**subfe** (**subfe. subfeo subfeo.**)	**r**D,**r**A,**r**B
Subtract from Immediate Carrying	**subfic**	**r**D,**r**A,SIMM

Table 3-5. Integer Arithmetic Instructions (continued)

Name	Mnemonic	Syntax
Subtract from Minus One Extended	**subfme** (**subfme. subfmeo subfmeo.**)	**r**D,**r**A
Subtract from Zero Extended	**subfze** (**subfze. subfzeo subfzeo.**)	**r**D,**r**A

Although there is no subtract immediate instruction, its effect is achieved by negating the immediate operand of an **addi** instruction. Simplified mnemonics include this negation. Subtract instructions subtract the second operand (**r**A) from the third (**r**B). Simplified mnemonics are provided in which the third is subtracted from the second. See Appendix B, "Simplified Mnemonics."

An implementation that executes instructions with the overflow exception enable bit (OE) set or that sets the carry bit (CA) can either execute these instructions slowly or prevent execution of the next instruction until the operation completes. Chapter 10, "Execution Timing," describes how the e500mc handles such CR dependencies. The summary overflow and overflow bits XER[SO,OV] are set to reflect an overflow condition of a 32-bit result only if the instruction's OE bit is set.

3.4.3.1.2 Integer Compare Instructions

Integer compare instructions algebraically or logically compare the contents of **r**A with either the zero-extended value of the UIMM operand, the sign-extended value of the SIMM operand, or the contents of **r**B. The comparison is signed for **cmpi** and **cmp** and unsigned for **cmpli** and **cmpl**. Table 3-6 lists integer compare instructions. The L bit should always be 0.

Table 3-6. Integer Compare Instructions

Name	Mnemonic	Syntax
Compare	**cmp**	**cr**D,L,**r**A,**r**B
Compare Immediate	**cmpi**	**cr**D,L,**r**A,SIMM
Compare Logical	**cmpl**	**cr**D,L,**r**A,**r**B
Compare Logical Immediate	**cmpli**	**cr**D,L,**r**A,UIMM

The **cr**D operand can be omitted if the result of the comparison is to be placed in CR0. Otherwise the target CR field must be specified in **cr**D by using an explicit field number.

For information on simplified mnemonics, see Appendix B, "Simplified Mnemonics."

3.4.3.1.3 Integer Logical Instructions

The logical instructions, shown in Table 3-7, perform bit-parallel operations. Logical instructions do not affect XER[SO,OV,CA]. See Appendix B, "Simplified Mnemonics," for simplified mnemonic examples for integer logical operations.

Table 3-7. Integer Logical Instructions

Name	Mnemonic	Syntax	Implementation Notes
AND	**and (and.)**	**r**A,**r**S,**r**B	—
AND Immediate	**andi.**	**r**A,**r**S,UIMM	—

Table 3-7. Integer Logical Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
AND Immediate Shifted	**andis.**	**r**A,**r**S,UIMM	—
AND with Complement	**andc** (**andc.**)	**r**A,**r**S,**r**B	—
Count Leading Zeros Word	**cntlzw** (**cntlzw.**)	**r**A,**r**S	—
Equivalent	**eqv** (**eqv.**)	**r**A,**r**S,**r**B	—
Extend Sign Byte	**extsb** (**extsb.**)	**r**A,**r**S	—
Extend Sign Halfword	**extsh** (**extsh.**)	**r**A,**r**S	—
NAND	**nand** (**nand.**)	**r**A,**r**S,**r**B	—
NOR	**nor** (**nor.**)	**r**A,**r**S,**r**B	—
OR	**or** (**or.**)	**r**A,**r**S,**r**B	—
OR Immediate	**ori**	**r**A,**r**S,UIMM	**ori r0,r0,0** is the preferred form for a NOP. At dispatch it may enter the completion queue but not to an execution unit.
OR Immediate Shifted	**oris**	**r**A,**r**S,UIMM	—
OR with Complement	**orc** (**orc.**)	**r**A,**r**S,**r**B	—
XOR	**xor** (**xor.**)	**r**A,**r**S,**r**B	—
XOR Immediate	**xori**	**r**A,**r**S,UIMM	—
XOR Immediate Shifted	**xoris**	**r**A,**r**S,UIMM	—

3.4.3.1.4 Integer Rotate and Shift Instructions

Rotation operations are performed on data from a GPR, and the result, or a portion of the result, is returned to a GPR. Integer rotate instructions, summarized in Table 3-8, rotate the contents of a register. The result is either inserted into the target register under control of a mask (if a mask bit is set the associated bit of the rotated data is placed into the target register, and if the mask bit is cleared the associated bit in the target register is unchanged) or ANDed with a mask before being placed into the target register. Appendix B, "Simplified Mnemonics," lists simplified mnemonics that allow simpler coding of often-used functions such as clearing the left- or right-most bits of a register, left or right justifying an arbitrary field, and simple rotates and shifts.

Table 3-8. Integer Rotate Instructions

Name	Mnemonic	Syntax
Rotate Left Word then AND with Mask	**rlwnm** (**rlwnm.**)	**r**A,**r**S,**r**B,MB,ME
Rotate Left Word Immediate then Mask Insert	**rlwimi** (**rlwimi.**)	**r**A,**r**S,SH,MB,ME
Rotate Left Word Immediate then AND with Mask	**rlwinm** (**rlwinm.**)	**r**A,**r**S,SH,MB,ME

Integer shift instructions, listed in Table 3-9, perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Appendix B, "Simplified Mnemonics," shows how to simplify coding of such shifts. Multiple-precision shifts can be programmed as described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

Table 3-9. Integer Shift Instructions

Name	Mnemonic	Syntax
Shift Left Word	**slw** (**slw.**)	**r**A,**r**S,**r**B
Shift Right Word	**srw** (**srw.**)	**r**A,**r**S,**r**B
Shift Right Algebraic Word Immediate	**srawi** (**srawi.**)	**r**A,**r**S,SH
Shift Right Algebraic Word	**sraw** (**sraw.**)	**r**A,**r**S,**r**B

3.4.3.2 Load and Store Instructions

Although load and store instructions are issued and translated in program order, accesses can occur out of order. Memory synchronizing (barrier) instructions are provided to enforce strict ordering. e500mc load and store instructions are grouped as follows:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Floating-point load instructions
- Floating-point store instructions
- Memory synchronization instructions
- External PID load and store instructions are described in Section 3.4.11.2, “External PID Load Store Instructions”
- Decorated storage load and store instructions are described in Section 3.4.3.2.8, “Decorated Load and Store Instructions

Implementation Notes:

The following describes how the e500mc handles misalignment: The e500mc provides hardware support for misaligned memory accesses, but at the cost of performance degradation. For loads that hit in the cache, the LSU’s throughput degrades to one misaligned load every 3 cycles. Similarly, stores can be translated at a rate of one misaligned store every 3 cycles. Additionally, after translation, each misaligned store is treated as two distinct entries in the store queue, each requiring a cache access.

A word or halfword memory access requires multiple accesses if it crosses a doubleword boundary but not if it crosses a natural boundary.

Frequent use of misaligned memory accesses can greatly degrade performance.

Any load doubleword, word, or load halfword that crosses a doubleword boundary is interruptible, and therefore can restart. If the first access has been performed when the interrupt occurs, it is performed again when the instruction is restarted, even if it is to a page marked as guarded. Any load word or load halfword that crosses a translation boundary may take a translation exception on the second access. In this case, the first access may have already occurred.

Accesses that cross a translation boundary where the endianness differs cause a byte-ordering data storage interrupt.

3.4.3.2.1 Update Forms of Load and Store Instructions

Some integer load and store as well as floating-point load and store instructions contain update forms which update **r**A with the calculated EA. These instructions are specified with a '**u**' in the mnemonic.

Update forms where **r**A = 0 are considered invalid.

Update forms for loads when **r**A = **r**D are considered invalid.

3.4.3.2.2 General Integer Load Instructions

This table lists the integer load instructions.

Table 3-10. Integer Load Instructions

Name	Mnemonic	Syntax
Load Byte and Zero	**lbz**	**r**D,**d(r**A**)**
Load Byte and Zero Indexed	**lbzx**	**r**D,**r**A,**r**B
Load Byte and Zero with Update	**lbzu**	**r**D,**d(r**A**)**
Load Byte and Zero with Update Indexed	**lbzux**	**r**D,**r**A,**r**B
Load Halfword and Zero	**lhz**	**r**D,**d(r**A**)**
Load Halfword and Zero Indexed	**lhzx**	**r**D,**r**A,**r**B
Load Halfword and Zero with Update	**lhzu**	**r**D,**d(r**A**)**
Load Halfword and Zero with Update Indexed	**lhzux**	**r**D,**r**A,**r**B
Load Halfword Algebraic	**lha**	**r**D,**d(r**A**)**
Load Halfword Algebraic Indexed	**lhax**	**r**D,**r**A,**r**B
Load Halfword Algebraic with Update	**lhau**	**r**D,**d(r**A**)**
Load Halfword Algebraic with Update Indexed	**lhaux**	**r**D,**r**A,**r**B
Load Word and Zero	**lwz**	**r**D,**d(r**A**)**
Load Word and Zero Indexed	**lwzx**	**r**D,**r**A,**r**B
Load Word and Zero with Update	**lwzu**	**r**D,**d(r**A**)**
Load Word and Zero with Update Indexed	**lwzux**	**r**D,**r**A,**r**B

Some implementations execute the load algebraic (**lha**, **lhax**, **lhau**, **lhaux**) instructions with greater latency than other types of load instructions. The e500mc executes these instructions with the same latency as other load instructions.

The e500mc also contains load and store instructions for atomic memory accesses. These are described in Section 3.4.8, "Memory Synchronization Instructions."

3.4.3.2.3 Integer Store Instructions

For integer store instructions, the **r**S contents are stored into the byte, halfword, word, or doubleword in memory addressed by the EA. This table summarizes integer store instructions.

Table 3-11. Integer Store Instructions

Name	Mnemonic	Syntax
Store Byte	**stb**	**r**S,**d(r**A**)**
Store Byte Indexed	**stbx**	**r**S,**r**A,**r**B
Store Byte with Update	**stbu**	**r**S,**d(r**A**)**
Store Byte with Update Indexed	**stbux**	**r**S,**r**A,**r**B
Store Halfword	**sth**	**r**S,**d(r**A**)**
Store Halfword Indexed	**sthx**	**r**S,**r**A,**r**B
Store Halfword with Update	**sthu**	**r**S,**d(r**A**)**
Store Halfword with Update Indexed	**sthux**	**r**S,**r**A,**r**B
Store Word	**stw**	**r**S,**d(r**A**)**
Store Word Indexed	**stwx**	**r**S,**r**A,**r**B
Store Word with Update	**stwu**	**r**S,**d(r**A**)**
Store Word with Update Indexed	**stwux**	**r**S,**r**A,**r**B

3.4.3.2.4 Integer Load and Store with Byte-Reverse Instructions

This table describes integer load and store with byte-reverse instructions. The Power ISA supports true little endian on a per-page basis.

Table 3-12. Integer Load and Store with Byte-Reverse Instructions

Name	Mnemonic	Syntax
Load Halfword Byte-Reverse Indexed	**lhbrx**	**r**D,**r**A,**r**B
Load Word Byte-Reverse Indexed	**lwbrx**	**r**D,**r**A,**r**B
Store Halfword Byte-Reverse Indexed	**sthbrx**	**r**S,**r**A,**r**B
Store Word Byte-Reverse Indexed	**stwbrx**	**r**S,**r**A,**r**B

Some implementations run the load/store byte-reverse instructions with greater latency than other types of load/store instructions. The e500mc executes these instructions with the same latency as other load/store instructions.

3.4.3.2.5 Integer Load and Store Multiple Instructions

The load/store multiple instructions, listed in Table 3-13, move blocks of data to and from GPRs. If their operands require memory accesses crossing a page boundary, these instructions may require a data storage interrupt to translate the second page. Also, if one of these instructions is interrupted, it is restarted, requiring multiple memory accesses.

The architecture defines Load Multiple Word (**lmw**) with **r**A in the range of GPRs to be loaded as an invalid form. Load and store multiple accesses not word aligned cause an alignment exception.

If **r**A is in the range of registers to be loaded, what gets loaded into any register depends on whether an interrupt occurs (and at what point the interrupt occurs) requiring the instruction to be restarted. If **r**A is loaded with a new value from memory and an interrupt and subsequent return to re-execute the **lmw** instruction occurs, **r**A has a different value and forms a completely different EA, which causes the registers to be reloaded from a storage location not intended by the program.

If an interrupt does not occur, the register to be loaded starting at **r**A + 1 (for example, if **r**A is **r**10, then **r**11 is **r**A + 1) then is loaded from the new address calculated from the updated value of **r**A and the current running displacement.

Table 3-13. Integer Load and Store Multiple Instructions

Name	Mnemonic	Syntax
Load Multiple Word	**lmw**	**r**D,**d**(**r**A)
Store Multiple Word	**stmw**	**r**S,**d**(**r**A)

3.4.3.2.6 Floating-Point Load Instructions

Separate floating-point load instructions are used for single-precision and double-precision operands. Because FPRs support only double-precision format, the FPU converts single-precision data to double-precision format before loading the operands into the target FPR. This conversion is described fully in the “Floating-Point Models” appendix in the *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors*.

This table provides a list of the floating-point load instructions.

Table 3-14. Floating-Point Load Instructions

Name	Mnemonic	Operand Syntax
Load Floating-Point Double	**lfd**	**fr**D,d(**r**A)
Load Floating-Point Double Indexed	**lfdx**	**fr**D,**r**A,**r**B
Load Floating-Point Double with Update	**lfdu**	**fr**D,d(**r**A)
Load Floating-Point Double with Update Indexed	**lfdux**	**fr**D,**r**A,**r**B
Load Floating-Point Single	**lfs**	**fr**D,d(**r**A)
Load Floating-Point Single Indexed	**lfsx**	**fr**D,**r**A,**r**B
Load Floating-Point Single with Update	**lfsu**	**fr**D,d(**r**A)
Load Floating-Point Single with Update Indexed	**lfsux**	**fr**D,**r**A,**r**B

3.4.3.2.7 Floating-Point Store Instructions

There are three basic forms of the store instruction—single-precision, double-precision, and integer. The integer form is supported by the optional **stfiwx** instruction. Because the FPRs support only double-precision format for floating-point data, the FPU converts double-precision data to single-precision format before storing the operands. The conversion steps are described in “Floating-Point Store Instructions” in Appendix D, “Floating-Point Models,” in the *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors*.

This table lists the floating-point store instructions.

Table 3-15. Floating-Point Store Instructions

Name	Mnemonic	Operand Syntax
Store Floating-Point as Integer Word Indexed	**stfiwx**	**fr**S,**r**A,**r**B
Store Floating-Point Double	**stfd**	**fr**S,d(**r**A)
Store Floating-Point Double Indexed	**stfdx**	**fr**S,**r**A,**r**B
Store Floating-Point Double with Update	**stfdu**	**fr**S,d(**r**A)
Store Floating-Point Double with Update Indexed	**stfdux**	**fr**S,**r**A,**r**B
Store Floating-Point Single	**stfs**	**fr**S,d(**r**A)
Store Floating-Point Single Indexed	**stfsx**	**fr**S,**r**A,**r**B
Store Floating-Point Single with Update	**stfsu**	**fr**S,d(**r**A)
Store Floating-Point Single with Update Indexed	**stfsux**	**fr**S,**r**A,**r**B

3.4.3.2.8 Decorated Load and Store Instructions

Decorated load and store instructions allow efficient, SoC-specific operations targeted by storage address, such as packet-counting statistics. The SoC defines specific semantics understood by a SoC-customized resource that requires them. To determine the full semantic of a decorated storage operation, consult reference manual for the integrated device.

The architecture defines the decorated instructions listed in Table 3-16, which provide the EA in **r**B and the decoration in **r**A.

Table 3-16. Decorated Load and Store Instructions

Instruction	Mnemonic	Syntax	Description
Load Byte with Decoration Indexed	**lbdx**	**r**D,**r**A,**r**B	The byte, halfword, word, or floating-point doubleword addressed by EA (in **r**B) using the decoration supplied by **r**A is loaded into target GPR r**D**.
Load Halfword with Decoration Indexed	**lhdx**	**r**D,**r**A,**r**B	
Load Word with Decoration Indexed	**lwdx**	**r**D,**r**A,**r**B	
Load Floating-Point Doubleword with Decoration Indexed	**lfddx**	**fr**D,**r**A,**r**B	
Store Byte with Decoration Indexed	**stbdx**	**r**S,**r**A,**r**B	The contents of **r**S and the decoration supplied by GPR(**r**A) are stored into byte, halfword, word, or floating-point doubleword in storage addressed by EA (r**B**).
Store Halfword with Decoration Indexed	**sthdx**	**r**S,**r**A,**r**B	
Store Word with Decoration Indexed	**stwdx**	**r**S,**r**A,**r**B	
Store Floating-Point Doubleword with Decoration Indexed	**stfddx**	**fr**S,**r**A,**r**B	
Decorated Storage Notify	**dsn**	**r**A,**r**B	Address-only operation that sends a decoration without any associated load or store semantics.

Decorated load and store instructions are treated as normal cacheable loads and stores when they are to addresses that are not cache inhibited. **dsn** is treated as a 0 byte store. Decorated load and store instructions to addresses that are caching inhibited are always treated as guarded, regardless of the setting of the G bit

in the associated TLB entry. This prevents speculative decorated loads from executing, which potentially produces side effects other than the normal load semantics.

Implementation Notes:

The number of bits of decoration that are delivered along with the address for decorated load, store and notify operations is implementation dependent based on how many bits of decoration the interconnect supports. For e500mc, only the low-order 4 bits of the decoration in **r**A are implemented.

3.4.4 Floating-Point Execution Model

The core provides hardware support for all single- and double-precision floating-point operations for most value representations and all rounding modes. The PowerPC architecture provides for hardware implementation of a floating-point system as defined in ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating Point Arithmetic*. For detailed information about the floating-point execution model, refer to the "Operand Conventions" chapter in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

The IEEE 754 standard includes 64- and 32-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands.

The UISA follows these guidelines:

- Double-precision arithmetic instructions may have single-precision operands but always produce double-precision results.
- Single-precision arithmetic instructions require all operands to be single-precision and always produce single-precision results.

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

All Power ISA implementations provide the equivalent of the execution models described in this chapter to ensure that identical results are obtained. The definition of the arithmetic instructions for infinities, denormalized numbers, and NaNs follow conventions described in the following sections.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit computation of the adjusted exponent value in the following examples when the corresponding exception enable bit is one:

- Underflow during multiplication using a denormalized factor
- Overflow during division using a denormalized divisor

3.4.4.1 Floating-Point Instructions

This section describes the floating-point instructions, which include the following:

- Floating-Point Arithmetic Instructions

- Floating-Point Multiply-Add Instructions
- Floating-Point Rounding and Conversion Instructions
- Floating-Point Compare Instructions
- Floating-Point Status and Control Register Instructions
- Floating-Point Move Instructions

See Section 3.4.3.2, "Load and Store Instructions," for information about floating-point loads and stores.

The Power ISA architecture supports a floating-point system as defined in the IEEE 754 standard. All floating-point operations conform to the IEEE 754 standard, except if software sets the non-IEEE mode bit (NI) in the FPSCR. The core is in the nondenormalized mode when the NI bit is set in the FPSCR. If set the following behavioral changes occur:

- If a denormalized result is produced, a default result of zero is generated. The generated zero has the same sign as the denormalized number.
- If a denormalized value occurs on input, a zero value of the same sign as the input is used in the calculation in place of the denormalized number.

The core performs single- and double-precision floating-point operations compliant with the IEEE 754 floating-point standard.

Implementation Notes

Single-precision denormalized results require two additional processor clock cycles to round. When loading or storing a single-precision denormalized number, the load/store unit may take up to 24 processor clock cycles to convert between the internal double-precision format and the external single-precision format.

3.4.4.1.1 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are listed in this table.

Table 3-17. Floating-Point Arithmetic Instructions

Name	Mnemonic	Operand Syntax
Floating Add (Double-Precision)	**fadd** (**fadd.**)	**fr**D,**fr**A,**fr**B
Floating Add Single	**fadds** (**fadds.**)	**fr**D,**fr**A,**fr**B
Floating Divide (Double-Precision)	**fdiv** (**fdiv.**)	**fr**D,**fr**A,**fr**B
Floating Divide Single	**fdivs** (**fdivs.**)	**fr**D,**fr**A,**fr**B
Floating Multiply (Double-Precision)	**fmul** (**fmul.**)	**fr**D,**fr**A,**fr**C
Floating Multiply Single	**fmuls** (**fmuls.**)	**fr**D,**fr**A,**fr**C
Floating Reciprocal Estimate Single	**fres** (**fres.**)	**fr**D,**fr**B
Floating Reciprocal Square Root Estimate	**frsqrte** (**frsqrte.**)	**fr**D,**fr**B
Floating Select	**fsel** (**fsel.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Subtract (Double-Precision)	**fsub** (**fsub.**)	**fr**D,**fr**A,**fr**B
Floating Subtract Single	**fsubs** (**fsubs.**)	**fr**D,**fr**A,**fr**B

3.4.4.1.2 Floating-Point Multiply-Add Instructions

These instructions combine multiply and add operations without an intermediate rounding operation. The fractional part of the intermediate product is 106 bits wide, and all 106 bits take part in the add/subtract portion of the instruction.

The floating-point multiply-add instructions are listed in this table.

Table 3-18. Floating-Point Multiply-Add Instructions

Name	Mnemonic	Operand Syntax
Floating Multiply-Add (Double-Precision)	**fmadd** (**fmadd.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Add Single	**fmadds** (**fmadds.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Subtract (Double-Precision)	**fmsub** (**fmsub.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Subtract Single	**fmsubs** (**fmsubs.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Add (Double-Precision)	**fnmadd** (**fnmadd.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Add Single	**fnmadds** (**fnmadds.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Subtract (Double-Precision)	**fnmsub** (**fnmsub.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Subtract Single	**fnmsubs** (**fnmsubs.**)	**fr**D,**fr**A,**fr**C,**fr**B

Implementation Notes

Single-precision multiply-type instructions operate faster than their double-precision equivalents. See Chapter 10, “Execution Timing,” for more information.

3.4.4.1.3 Floating-Point Rounding and Conversion Instructions

The Floating Round to Single-Precision (**frsp**) instruction is used to truncate a 64-bit double-precision number to a 32-bit single-precision floating-point number. The floating-point conversion instructions convert a 64-bit double-precision floating-point number to signed integer numbers.

Examples of uses of these instructions to perform various conversions can be found in Appendix D, “Floating-Point Models,” in the *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors*. The floating-point rounding instructions are shown in this table.

Table 3-19. Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Operand Syntax
Floating Convert to Integer Word	**fctiw** (**fctiw.**)	**fr**D,**fr**B
Floating Convert to Integer Word with Round Toward Zero	**fctiwz** (**fctiwz.**)	**fr**D,**fr**B
Floating Round to Single-Precision	**frsp** (**frsp.**)	**fr**D,**fr**B

3.4.4.1.4 Floating-Point Compare Instructions

Floating-point compare instructions compare the contents of two floating-point registers. The comparison ignores the sign of zero (that is +0 = –0). The floating-point compare instructions are listed in this table.

Table 3-20. Floating-Point Compare Instructions

Name	Mnemonic	Operand Syntax
Floating Compare Ordered	**fcmpo**	**crf**D,**fr**A,**fr**B
Floating Compare Unordered	**fcmpu**	**crf**D,**fr**A,**fr**B

3.4.4.1.5 Floating-Point Status and Control Register Instructions

Every FPSCR instruction synchronizes the effects of all floating-point instructions executed by a given processor. Executing an FPSCR instruction ensures that all floating-point instructions previously initiated by the given processor have completed before the FPSCR instruction is initiated and that no subsequent floating-point instructions are initiated by the given processor until the FPSCR instruction has completed. The FPSCR instructions are listed in this table.

Table 3-21. Floating-Point Status and Control Register Instructions

Name	Mnemonic	Operand Syntax
Move from FPSCR	**mffs** (**mffs.**)	**fr**D
Move to Condition Register from FPSCR	**mcrfs**	**crf**D,**crf**S
Move to FPSCR Bit 0	**mtfsb0** (**mtfsb0.**)	**crb**D
Move to FPSCR Bit 1	**mtfsb1** (**mtfsb1.**)	**crb**D
Move to FPSCR Field Immediate	**mtfsfi** (**mtfsfi.**)	**crf**D,IMM
Move to FPSCR Fields	**mtfsf** (**mtfsf.**)	FM,**fr**B

NOTE

The architecture notes that, in some implementations, the Move to FPSCR Fields (**mtfsf***x*) instruction may perform more slowly when only a portion of the fields are updated as opposed to all of the fields. This is not the case in the e500mc.

3.4.4.1.6 Floating-Point Move Instructions

Floating-point move instructions copy data from one floating-point register to another. The floating-point move instructions do not modify the FPSCR. The CR update option in these instructions controls the placing of result status into CR1. Floating-point move instructions are listed in this table.

Table 3-22. Floating-Point Move Instructions

Name	Mnemonic	Operand Syntax
Floating Absolute Value	**fabs** (**fabs.**)	**fr**D,**fr**B
Floating Move Register	**fmr** (**fmr.**)	**fr**D,**fr**B
Floating Negate	**fneg** (**fneg.**)	**fr**D,**fr**B
Floating Negative Absolute Value	**fnabs** (**fnabs.**)	**fr**D,**fr**B

3.4.5 Branch and Flow Control Instructions

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR. Information about branch instruction address calculation is provided in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

3.4.5.1 Conditional Branch Control

For branch conditional instructions, the BO operand specifies the conditions under which the branch is taken. The first four bits of the BO operand specify how the branch is affected by or affects the condition and count registers. The fifth bit, shown in Table 3-24 as having the value *t*, is used by some implementations for branch prediction; this is not used on the e500mc.

NOTE

The e500mc ignores the BO operand for branch prediction and the BH field in the branch conditional to count register and branch conditional to link register instructions. Instead it implements dynamic branch prediction as part of the branch table buffer (BTB), described in Section 10.4.1, "Branch Unit Execution."

Table 3-23. BO Bit Descriptions

BO Bits	Description
0	Setting this bit causes the CR bit to be ignored.
1	Bit value to test against
2	Setting this causes the decrement to not be decremented.
3	Setting this bit reverses the sense of the CTR test.
4	The e500mc does not use static branch prediction and ignores this bit.

The encodings for the BO operands are shown in Table 3-24.

Table 3-24. BO Operand Encodings

BO	Description
0000*z*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.
0001*z*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*at*	Branch if the condition is FALSE.
0100*z*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is TRUE.
0101*z*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*at*	Branch if the condition is TRUE.
1*a*00*t*	Decrement the CTR, then branch if the decremented CTR ≠ 0.
1*a*01*t*	Decrement the CTR, then branch if the decremented CTR = 0.
1*z*1*zz*	Branch always.

Note:

1. In this table, *z* indicates a bit that is ignored. Note that the *z* bits should be cleared, as they may be assigned a meaning in some future version of the architecture.
2. The *a* and *t* bits provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance. e500mc always uses dynamic prediction and ignores these bits.

The 5-bit BI operand in branch conditional instructions specifies which CR bit represents the condition to test. The CR bit selected is BI +32.

If branch instructions use immediate addressing operands, target addresses can be computed ahead of the branch instruction so instructions can be fetched along the target path. If the branch instructions use LR or CTR, instructions along the path can be fetched if the LR or CTR is loaded sufficiently ahead of the branch instruction.

Branching can be conditional or unconditional, and optionally a branch return address is created by storing the EA of the instruction following the branch instruction in the LR after the branch target address has been computed. This is done regardless of whether the branch is taken.

3.4.5.2 Branch Instructions

Table 3-25 lists branch instructions. Appendix B, "Simplified Mnemonics," lists simplified mnemonics and symbols provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. The e500mc does not use the BO operand for static branch prediction.

Table 3-25. Branch Instructions

Name	Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI

3.4.5.3 Integer Select (isel)

Integer Select (**isel**), shown in Table 3-26, is a conditional register move instruction that helps eliminate branches. Programming guidelines for **isel** are given in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

Table 3-26. Integer Select Instruction

Name	Mnemonic	Syntax
Integer Select	**isel**	**r**D,**r**A,**r**B,**cr**B

3.4.5.4 Condition Register Logical Instructions

Table 3-27 shows the condition register logical instructions. Both these instructions and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Table 3-27. Condition Register Logical Instructions

Name	Mnemonic	Syntax
Condition Register AND	**crand**	**crb**D,**crb**A,**crb**B
Condition Register OR	**cror**	**crb**D,**crb**A,**crb**B
Condition Register XOR	**crxor**	**crb**D,**crb**A,**crb**B
Condition Register NAND	**crnand**	**crb**D,**crb**A,**crb**B
Condition Register NOR	**crnor**	**crb**D,**crb**A,**crb**B
Condition Register Equivalent	**creqv**	**crb**D,**crb**A,**crb**B
Condition Register AND with Complement	**crandc**	**crb**D,**crb**A,**crb**B
Condition Register OR with Complement	**crorc**	**crb**D,**crb**A,**crb**B
Move Condition Register Field	**mcrf**	**crf**D,**crf**S

Any of these instructions for which the LR update option is enabled are considered invalid.

3.4.5.5 Trap Instructions

Trap instructions, shown in Table 3-28, test for a specified set of conditions. If a condition is met, a system trap program interrupt is taken. If no conditions are met, execution continues normally. See Section 4.9.8, "Program Interrupt—IVOR6 and Appendix B, "Simplified Mnemonics," for more information.

Table 3-28. Trap Instructions

Name	Mnemonic	Syntax
Trap Word Immediate	**twi**	TO,**r**A,SIMM
Trap Word	**tw**	TO,**r**A,**r**B

3.4.5.6 System Linkage Instruction

The System Call (**sc**) instruction permits a program to call on the system to perform a service or an operating system to call on the hypervisor to perform a service; see Table 3-29 and Section 3.4.11.1, "System Linkage and MSR Access Instructions."

Table 3-29. System Linkage Instruction

Name	Mnemonic	Syntax
System Call	**sc**	LEV

Executing **sc** invokes the system call interrupt handler or the hypervisor system call interrupt handler depending on the value of the LEV field, see Section 4.9.10, "System Call/Hypervisor System Call Interrupt—IVOR8/GIVOR8/IVOR40."

An **sc** instruction without the level field is treated by the assembler as an **sc** with LEV = 0.

3.4.5.7 Hypervisor Privilege Instruction

The hypervisor facility defines the Generate Embedded Hypervisor Privilege Exception instruction, **ehpriv**, which generates a hypervisor privilege exception. See Section 4.9.19, "Hypervisor Privilege Interrupt—IVOR41." **ehpriv** is fully described in *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. Note that the OC field is not interpreted by hardware but is for the use of the hypervisor to provide specific emulation.

Table 3-30 shows the hypervisor privilege instruction.

Table 3-30. Hypervisor Privilege Instruction

Name	Mnemonic	Syntax
Hypervisor Privilege	**ehpriv**	OC

3.4.6 Processor Control Instructions

Processor control instructions read from and write to the CR, MSR, and SPRs as well as the **wait** instruction.

3.4.6.1 Move to/from Condition Register Instructions

Table 3-31 summarizes the instructions for reading from or writing to the CR.

Table 3-31. Move to/from Condition Register Instructions

Name	Mnemonic	Syntax	Implementation Note
Move to Condition Register Fields	**mtcrf**	CRM,**r**S	On some implementations, **mtcrf** may perform more slowly if only a portion of the fields are updated. This is not so for the e500mc.
Move to Condition Register from XER	**mcrxr**	**cr**D	—
Move from Condition Register	**mfcr**	**r**D	—

Table 3-31. Move to/from Condition Register Instructions (continued)

Name	Mnemonic	Syntax	Implementation Note
Move from One Condition Register Field	**mfocrf**	**r**D,FXM	See the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* for a full description of this instruction.
Move to One Condition Register Field	**mtocrf**	FXM,**r**S	See the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* for a full description of this instruction.

Implementation Notes

e500mc implements **mfocrf** the same as **mfcr** and all the contents of the CR are moved to the destination register.

e500mc implements **mtocrf** the same as **mtcrf** and all the fields of the CR specified by FXM are moved to the CR fields specified by FXM.

3.4.6.2 Move to/from Special Purpose Register Instructions

Table 3-32 lists the **mtspr** and **mfspr** instructions.

Table 3-32. Move to/from Special-Purpose Register Instructions

Name	Mnemonic	Syntax	Comments
Move to Special-Purpose Register	**mtspr**	SPR,**r**S	—
Move from Special-Purpose Register	**mfspr**	**r**D,SPR	—
Move from Time Base	**mftb**	**r**D,TBR	**mftb** behaves as if it were an **mfspr**. Although **mftb** is supported, **mfspr** is prefered, because **mftb** can only be used to read from TBL and TBU; **mfspr** can be used to read TBL, TBU, and ATB SPRs.

3.4.6.3 Wait for Interrupt Instruction

wait stops synchronous processor activity. Execution ensures that all instructions complete before **wait** completes, and that no subsequent instructions initiate until an asynchronous interrupt or a debug post-completion (for example, ICMP) event and subsequent interrupt occurs. On the e500mc, **wait** also causes any prefetched instructions to be discarded and processor instruction fetching ceases until an interrupt occurs.

Executing a **wait** instruction is a hint to the processor that no further synchronous processor activity occurs until the next asynchronous interrupt occurs. The processor may use this to reduce power consumption. The **wait** instruction completes and then waits for an interrupt. When an interrupt occurs while the processor is waiting, its associated save/restore register 0 points to the instruction following the **wait**.

Table 3-33. Wait for Interrupt Instruction

Name	Mnemonic	Syntax
Wait for Interrupt	**wait**	—

The e500mc does not implement the WC field of the wait instruction as defined in Power ISA 2.06. The WC field is ignored.

3.4.7 Performance Monitor Instructions (User Level)

The performance monitor provides read-only, application-level access to some performance monitor resources. Instructions are listed in Table 3-34.

Table 3-34. Performance Monitor Instructions

Name	Mnemonic	Syntax
Move from Performance Monitor Register	**mfpmr**	**r**D,PMRN

The user-level PMRs listed in Section Table 2-49., "Performance Monitor Registers are accessed with **mfpmr**. Attempting to write user-level PMRs in either mode causes an illegal instruction exception.

3.4.8 Memory Synchronization Instructions

Memory synchronization instructions control the order in which memory operations complete with respect to asynchronous events and the order in which memory operations are seen by other mechanisms that access memory. See the section, "Atomic Update Primitives Using **lwarx** and **stwcx.**"in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* for additional information about these instructions and about related aspects of memory synchronization. See Table 3-35 for a summary.

Table 3-35. Memory Synchronization Instructions

Name	Mnemonic	Syntax	Implementation Notes
Instruction Synchronize	**isync**	—	**isync** is refetch serializing; the e500mc waits for previous instructions (including interrupts they generate) to complete before **isync** executes. This purges all instructions from the core and refetches the next instruction. **isync** does not wait for pending stores in the store queue to complete. Any subsequent instruction sees all effects of instructions before the **isync**. Because it prevents execution of subsequent instructions until previous instructions complete, if an **isync** follows a conditional branch instruction that depends on the value returned by a preceding load, the load on which the branch depends is performed before any loads caused by instructions after the **isync** even if the effects of the dependency are independent of the value loaded (for example, the value is compared to itself and the branch tests selected, CR*n*[EQ]), and even if branch targets the next sequential instruction.

Table 3-35. Memory Synchronization Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Load Word and Reserve Indexed	**lwarx**	**r**D,**r**A,**r**B	**lwarx** with **stwcx.** can emulate semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add. Both instructions should use the same real address, the same size of operation (byte, halfword, word or doubleword), however e500mc only requires that the real addresses be in the same coherence granule and the size of operation is ignored with respect to whether the store conditional is performed or not. The address must be naturally aligned, and should be in pages that are marked as WIMGE = 001*xx*. The e500mc makes reservations on behalf of aligned 64-byte sections of address space (coherence granule). While the e500mc supports making reservations to cache inhibited memory, or to cached memory when the cache is disabled, doing so may not be supported in the future. Additionally, while e500mc supports making the reservations and store conditionals to real addresses that differ but are within the same coherence granule or with different size operations to the same granule, doing so may not be supported in the future. Executing **lwarx** and **stwcx.** to a page marked write-through (WIMGE = 10*xxx*) causes a data storage exception. If the location is not naturally aligned, an alignment exception occurs. See "Atomic Update Primitives Using **lwarx** and **stwcx.**," in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*."
Memory Barrier	**mbar**	MO	**mbar** provides a memory barrier. The behavior of **mbar** depends on the value of MO operand. Note that **mbar** uses the same opcode as **eieio**, defined by the PowerPC architecture, and with which **mbar** (MO=1) semantics are identical. MO = 0—**mbar 0** instruction provides a storage ordering function for all memory access instructions executed by the processor executing **mbar 0**. Executing **mbar 0** ensures that all data storage accesses caused by instructions preceding the **mbar 0** have completed before any data storage accesses caused by any instructions after the **mbar 0**. This order is seen by all mechanisms. The memory barrier is throughout the memory hierarchy. In the e500mc, **mbar 0** waits for proceeding data memory accesses to become visible to the entire memory hierarchy; then it is broadcast on the CoreNet interface. **mbar 0** completes only after its address tenure. MO = 1— **mbar** functions identically to **eieio**. For more information, see Section 3.4.8.1, "mbar (MO = 1)."

Table 3-35. Memory Synchronization Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Memory Synchronize	**sync** (**msync**)	**L**	**sync** (former versions of the architecture used the mnemonic **msync**) provides a memory barrier to ensure the order of affected load and store memory accesses. sync provides 2 types of memory barriers specified by the L field: • L = 0 ("heavyweight sync" - **hwsync**), The memory barrier is throughout the memory hierarchy. In the e500mc, **sync 0** waits for proceeding data memory accesses to become visible to the entire memory hierarchy; then it is broadcast on the CoreNet interface. **sync 0** completes only after its address tenure. Subsequent instructions can execute out of order but complete only after the **sync 0** completes. The simplified mnemonics **hwsync**, **sync**, and **msync** are equivalent to **sync 0**. • L = 1 ("lightweight sync" - **lwsync**). The memory barrier provides an ordering function for the storage accesses caused by load, store, and **dcbz** instructions executed by the processor executing the **sync** instruction and for which the specified storage location is neither write through required nor caching inhibited. The applicable pairs are all pairs a_i,b_j of such accesses except those in which a_i is an access caused by a store or **dcbz** instruction and b_j is an access caused by a load instruction. The **sync 1** instruction memory barrier orders accesses described by the applicable pairs above to the local caches of the processor such that a_i is performed in all caches local to the processor prior to any b_j access. The simplified mnemonic **lwsync** is equivalent to **sync 1**. **sync** latency depends on the processor state when it is dispatched and on various system-level conditions. Frequent use of **sync 0** degrades performance and **sync 1** should be used where possible. In multiprocessing code that performs locking operations to lock shared data structures: • **sync** is used to ensure that all stores into a data structure caused by store instructions executed in a critical section of a program are performed with respect to another processor before the store that releases the lock is performed with respect to that processor. **sync 1** (**lwsync**) or **mbar 1** is preferable in many cases. • Unlike a context-synchronizing operation, **sync 0** does not discard prefetched instructions. The section, "Lock Acquisition and Import Barriers," in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* describes how the **sync** and **mbar** instructions can be used to control memory access ordering when memory is shared between programs.
Store Word Conditional Indexed	**stwcx.**	**r**S,**r**A,**r**B	See **lwarx** above for a description of how load and reserve and store conditional instructions are used in pairs. For **stwcx.** e500mc takes a data storage exception if the page is marked write-through (WIMGE = 10*xxx*) and takes an alignment exception if the access is not naturally aligned.

3.4.8.1 mbar (MO = 1)

As defined by the architecture, **mbar 1** (MO = 1) functions like **eieio**, as it is defined by the PowerPC architecture. It provides ordering for the effects of certain classes of load and store instructions. These instructions consist of two sets, which are ordered separately. Memory accesses caused by a **dcbz** or a **dcba** are ordered like a store. The two sets follow:

- Caching-inhibited, guarded loads and stores to memory and write-through-required stores to memory. **mbar 1** controls the order in which accesses are performed in main memory. It ensures that all applicable memory accesses caused by instructions preceding the **mbar 1** have completed with respect to main memory before any such accesses caused by instructions following **mbar 1** access main memory. It acts like a barrier that flows through the memory queues and to main

memory, preventing the reordering of memory accesses across the barrier. No ordering is performed for **dcbz** if the instruction causes the system alignment error handler to be invoked.

All accesses in this set are ordered as one set; there is not one order for guarded, caching-inhibited loads and stores and another for write-through-required stores.

- Stores to memory that are caching-allowed, write-through not required, and memory-coherency required. **mbar 1** controls the order in which accesses are performed with respect to coherent memory. It ensures that, with respect to coherent memory, applicable stores caused by instructions before the **mbar 1** complete before any applicable stores caused by instructions after it.

Except for **dcbz** and **dcba**, **mbar 1** does not affect the order of cache operations (whether caused explicitly by a cache management instruction or implicitly by the cache coherency mechanism). Also, **mbar 1** does not affect the order of accesses in one set with respect to accesses in the other.

mbar 1 may complete before memory accesses caused by instructions preceding it have been performed with respect to main memory or coherent memory as appropriate. **mbar 1** is intended for use in managing shared data structures, in accessing memory-mapped I/O, and in preventing load/store combining operations in main memory. For the first use, the shared data structure and the lock that protects it must be altered only by stores in the same set (for both cases described above). For the second use, **mbar 1** can be thought of as placing a barrier into the stream of memory accesses issued by a core, such that any given access appears to be on the same side of the barrier to both the core and the I/O device.

Like **mbar 0**, **mbar 1** is broadcast on the CoreNet interface, however, unlike **mbar 0**, **mbar 1** does not wait for it address tenure before completing execution.

Because the core performs store operations in order to memory that is designated as both caching-inhibited and guarded, **mbar 1** is needed for such memory only when loads must be ordered with respect to stores or with respect to other loads.

The section, “Lock Acquisition and Import Barriers,” in the *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors* describes how **sync** and **mbar** control memory access ordering when programs share memory.

3.4.9 Reservations

The ability to emulate an atomic operation using load with reservation and store conditional instructions is based on the conditional behavior of **stwcx.**, the reservation set by **lwarx**, and the clearing of that reservation if the target location is modified by another processor or mechanism before the store conditional instruction performs its store. Behavior of these instructions is described in the *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors*. On the e500mc, a reservation may be lost for any of the following reasons:

- Execution of a **stwcx.** by the processor
- Some other processor successfully modifies a location in the reservation granule and the address containing the reservation is marked as Memory Coherence Required (M = 1)
- Execution of another load with reservation instruction, which removes the old reservation and establishes a reservation at the address specified in the load with reservation instruction

- Some other processor successfully executes a **dcbtst**, **dcbtstep**, **dcbtstls**, **dcbal**, or **dcba** to a location in the reservation granule and the address containing the reservation is marked as Memory Coherence Required (M = 1)

On the e500mc, a reservation also may be lost for any of the following reasons:

- Any asynchronous interrupt is taken on the processor holding the reservation

Software should be written to not assume that the reservation is lost as the result of any interrupt. System software should always perform a store conditional instruction to a scratch location when performing a context switch or a partition switch to ensure that any held reservation is lost prior to initiating the new context.

3.4.10 Memory Control Instructions

Memory control instructions can be classified as follows:

- User- and supervisor-level cache management instructions.
- Supervisor-level-only translation lookaside buffer management instructions

This section describes the user-level cache management instructions. See Section 3.4.11.3, "Supervisor-Level Memory Control Instructions," for information about supervisor-level cache and translation lookaside buffer management instructions. Cache-locking instructions are described in Section 3.4.10.2, "Cache Locking Instructions."

3.4.10.1 User-Level Cache Instructions

The instructions listed in Table 3-36 help user-level programs manage on-chip caches if they are implemented. See Chapter 5, "Core Caches and Memory Subsystem," for more information about cache topics. The following sections describe how these operations are treated with respect to the e500mc's caches.

3.4.10.1.1 CT Field Values

The e500mc supports the following CT values:

- CT = 0 indicates the L1 cache.
- CT = 2 indicates the L2 cache.
- CT = 1 indicates the platform cache, if one is implemented on the integrated device.

  Additional values may be defined by the integrated device.
- The CT values 1, 3, 5, and 7 are not supported and produce undefined results when used with an address that is mapped to PCI address space on the integrated device.

As with other memory-related instructions, the effects of cache management instructions on memory are weakly-ordered. If the programmer must ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, a **sync** must be placed after those instructions.

Section 3.4.10.2, "Cache Locking Instructions," describes cache-locking instructions.

Table 3-36. User-Level Cache Instructions

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Allocate	**dcba**	**r**A,**r**B	If L1CSR0[DCBZ32] = 0 d**cba** operates on all bytes in cache line (cache-line operation) If L1CSR0[DCBZ32] = 1 **dcba** operates on 32 bytes (32-byte operation) **dcba** performs the same address translation and protection as a store and is treated as a store for debug events. The **dcba** instruction is treated as a 32 or cache line number of bytes store of zeros operation. The store operation is always size aligned to a 32 byte granule for a 32 byte operation and a cache line granule for a cache line operation by truncating the EA as necessary to achieve the appropriate granule. Using **dcba** with 32-byte operation may perform inferior to using cache-line operation and should be avoided when possible. The **dcba** is treated as a NOP if any of the following occur: • The page is marked write-through. • The page is marked caching inhibited. • A DTLB miss exception or protection violation occurs. • An L2 MMU multi-way hit is detected. • The targeted cache is disabled. When **dcba** is treated as a NOP, executing the **dcba** can result in IAC debug events, but does not cause DAC debug events. When using **dcba** in 32-bye operation on e500mc, if the line is not already valid in the cache, the line is read from main storage prior to performing the **dcba** operation.
Data Cache Block Allocate by Line	**dcbal**	**r**A,**r**B	This instruction behaves the same as **dcba** except it always operates on all bytes in the cache line regardless of the setting of L1CSR0[DCBZ32].
Data Cache Block Flush	**dcbf**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations: • For cache hits with the tag marked modified, the cache block is written back to memory and the cache entry is invalidated. • For cache hits with the tag marked not modified, the entry is invalidated. • For cache misses, no further action is taken. A **dcbf** is broadcast if WIMGE = xx1xx (coherency enforced).**dcbf** acts like a load with respect to address translation and memory protection. It executes in the LSU regardless of whether the cache is disabled or locked.
Data Cache Block Set to Zero	**dcbz**	**r**A,**r**B	If L1CSR0[DCBZ32] = 0 d**cbz** operates on all bytes in cache line (cache-line operation) If L1CSR0[DCBZ32] = 1 **dcbz** operates on 32 bytes (32-byte operation) **dcbz** performs the same address translation and protection as a store and is treated as a store for debug events. The **dcbz** instruction is treated as a 32 or cache line number of bytes store of zeros operation. The store operation is always size aligned to a 32 byte granule for a 32 byte operation and a cache line granule for a cache line operation by truncating the EA as necessary to achieve the appropriate granule. Using **dcbz** with 32-byte operation may perform inferior to using cache-line operation and should be avoided when possible. **dcbz** takes an alignment exception if any of the following occur: • The page is marked write-through. • The page is marked caching inhibited. When using **dcbz** in 32-bye operation on e500mc, if the line is not already valid in the cache, the line is read from main storage prior to performing the **dcbz** operation.
Data Cache Block Set to Zero by Line	**dcbzl**	**r**A,**r**B	This instruction behaves the same as **dcbz** except it always operates on all bytes in the cache line regardless of the setting of L1CSR0[DCBZ32].
Data Cache Block Store	**dcbst**	**r**A,**r**B	**dcbst** is implemented identically to **dcbf**.

Table 3-36. User-Level Cache Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Touch	**dcbt**	TH,**r**A,**r**B [1]	When **dcbt** executes, the e500mc checks for protection violations (as for a load instruction). **dcbt** is treated as a NOP in the following cases on e500mc: • The access would cause a DSI or DTLB Miss exception. • The page is marked Caching Inhibited. • The page is marked Guarded. • The targeted cache is disabled. • An L2 MMU multi-way hit is detected. • A **dcbf** (or **dcbst**, **dcbstep**, **dcbfep**) was previously executed and has not yet performed its flush and the **dcbt** and **dcbf** (or **dcbst**, **dcbstep**, **dcbfep**) specify the same cache line, but specify a different byte address within the cache line. • HID0[NOPTI] = 1 Otherwise, if no data is in the cache location, then a cache line fill is requested. When **dcbt** is treated as a NOP, executing the **dcbt** can result in IAC debug events, but does not cause DAC debug events.
Data Cache Block Touch for Store	**dcbtst**	TH,**r**A,**r**B [1]	**dcbtst** is treated as a **dcbt** except that the line is allocated and an attempt is made to mark it as exclusive in the specified cache.
Instruction Cache Block Invalidate	**icbi**	**r**A,**r**B	**icbi** is broadcast on the CoreNet interface. It should always be followed by a **sync** and an **isync** to make sure its effects are seen by instruction fetches and instruction execution following the **icbi** itself.
Instruction Cache Block Touch	**icbt**	CT,**r**A,**r**B	When **icbt** executes, the e500mc checks for protection violations (as for a load instruction). **icbt** is treated as a NOP in the following cases on e500mc: • The access would cause a DSI or TLB Miss exception. • The page is marked Caching Inhibited. • The page is marked Guarded. • The targeted cache is disabled. • An L2 MMU multi-way hit is detected. • HID0[NOPTI] = 1 Otherwise, if no data is in the cache location, then a cache line fill is requested. When **icbt** is treated as a NOP, executing the **icbt** can result in IAC debug events, but does not cause DAC debug events. Note that the primary instruction cache (CT=0) on e500mc does not perform **icbt** instructions and they are treated as a NOP.

[1] These instructions formerly used CT as the first operand, however, Power ISA has renamed the field as TH to accommodate the capability of performing streaming prefetches. For e500mc, the TH field can be treated as a CT value.

3.4.10.2 Cache Locking Instructions

Table 3-37 describes the implementation of the cache locking instructions, which are fully described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

The **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, and **icblc** cache-locking instructions require hypervisor state privilege to execute when MSRP[UCLEP] is set. Execution of these instructions in the guest supervisor state when MSRP[UCLEP] is set causes a hypervisor privilege exception. User mode execution of these instructions is unaffected and is still controlled by MSR[UCLE].

The CT field designates the specified cache in the cache hierarchy.

In these instructions, unless otherwise stated, the behavior applies to all.

Table 3-37. Cache Locking Instructions

Name	Mnemonic	Syntax	Implementation Details
Data Cache Block Lock Clear	**dcblc**	CT,rA,rB	The line in the specified cache is unlocked, making it eligible for replacement.
Data Cache Block Touch and Lock Set	**dcbtls**	CT,rA,rB	The line is loaded and locked into the specified cache.
Data Cache Block Touch for Store and Lock Set	**dcbtstls**	CT,rA,rB	The line is loaded and locked into the specified cache. The line is marked as modified.
Instruction Cache Block Lock Clear	**icblc**	CT,rA,rB	The line in the specified cache is unlocked, making it eligible for replacement.
Instruction Cache Block Touch and Lock Set	**icbtls**	CT,rA,rB	The line is loaded and locked into the specified cache.

Full descriptions of these instructions are in the "Instruction Set" chapter of the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. Note the following behavior for e500mc:

- Unable to lock conditions occur if the locking instruction has no exceptions and the line cannot be locked when CT = 0 or CT = 2. When an unable to lock condition occurs the line is not loaded or locked and L1CSR0[CUL] (or L1CSR1[ICUL] if an **icbtls** executed) is set to 1 regardless of whether the L2 cache or the primary cache was specified. An unable to lock condition occurs when:
  - The targeted cache is not enabled.
  - The target address is marked Caching Inhibited (WIMGE = 0bx1xx)
  - The instruction is an **icbtls**, the L2 cache is specified, and L2CSR0[L2DO] = 1.
  - The instruction is an **dcbtls** or **dcbtstls**, the L2 cache is specified, and L2CSR0[L2IO] = 1.
  - An error loading the line occurred either on the CoreNet interface or from the L2 cache.
- An overlocking condition occurs if the locking instruction has no exceptions and if all available ways in the specified cache are locked.
  - If an overlocking condition occurs in the primary cache (CT=0), the line is not loaded or locked and L1CSR0[CLO] (L1CSR0[ICLO] if an **icbtls** executed) is set to 1. L1CSR0[CUL] and L1CSR1[ICUL] are not set.
  - If an overlocking condition occurs in the L2 cache (CT=2) L2CSR0[L2LO] is set to 1. L1CSR0[CUL] and L1CSR1[ICUL] are not set. If L2CSR0[L2CLOA] = 1, the line is loaded and locked replacing and unlocking a line in the set that would have normally been selected for replacement if no lines in the set were locked. If L2CSR0[L2CLOA] = 0, the line is not loaded or locked.
- Note that setting L1CSR0[CLFC] flash invalidates all primary data cache lock bits and setting L1CSR0[ICLFC] flash invalidates all primary instruction cache lock bits, allowing system software to clear all cache locking in the L1 cache without knowing the addresses of the lines locked. Setting L2CSR0[L2LFC] flash invalidates all L2 cache lock bits allowing system software to clear all cache locking in the L2 cache without knowing the addresses of the lines locked.

Because L1 cache locking is not persistent, setting L1CSR0[CFI] or L1CSR1[ICFI] clears the locks in the respective caches because the lines containing the locks are invalidated.

- Touch and lock set instructions (**icbtls**, **dcbtls**, and **dcbtstls**) are always executed and are not treated as hints.
- Since e500mc implements cache locking for the L1 cache as non-persistent, when combining CT=2 cache operations with CT=0 data cache locking operations to the same line without any synchronization, the final state of the CT=0 lock operations is unknown (that is, the line may or may not be locked into the L1 data cache).

Cache locking clear instructions (**dcblc** and **icblc**) are NOPed if the specified cache is the L1 or L2 cache and the cache is not enabled.

Consult the SoC documentation to determined behavior for the platform cache (CT = 1).

To precisely detect an overlock or unable to lock condition in the primary data cache, system software must perform the following code sequence:

```
dcbtls
sync
mfspr (L1CSR0)
(check L1CSR0[CUL] for data cache index unable-to-lock condition)
(check L1CSR0[CLO] for data cache index overlock condition)
```

The following code sequence precisely detects an overlock in the primary instruction cache:

```
icbtls
sync
mfspr (L1CSR1)
check L1CSR1[ICUL] for instruction cache index unable-to-lock condition
check L1CSR1[ICLO] for instruction cache index overlock condition
```

3.4.11 Hypervisor- and Supervisor-Level Instructions

The architecture includes the structure of the memory management model, supervisor-level registers, and the interrupt model. This section describes the hypervisor- and supervisor-level instructions implemented on the e500mc. Instructions described here have an associated privilege and actions as described in Table 3-38.

Table 3-38. Instruction Execution Based on Privilege Level

Privilege Level of Instruction	User Mode (MSR[GS,PR]=0bx1)	Guest Supervisor Mode (MSR[GS,PR]=0b10)	Hypervisor Mode (MSR[GS,PR]=0b00)
User	execute normally	execute normally	execute normally
Guest Supervisor	privileged instruction exception	execute normally	execute normally
Hypervisor	privileged instruction exception	embedded hypervisor privilege exception	execute normally

3.4.11.1 System Linkage and MSR Access Instructions

Table 3-39 describes system linkage instructions as they are implemented on the e500mc. The user-level **sc** (LEV = 0) instruction lets a user program call on the system to perform a service and causes the processor to take a system call interrupt. The **sc** (LEV = 1) instruction is also used for the supervisor to

involve the hypervisor to perform a service and causes the processor to take an embedded hypervisor system call interrupt. The supervisor-level **rfi** and **rfgi** instructions are used for returning from an interrupt handler. The hypervisor level **rfci** instruction is used for critical interrupts; **rfdi** is used for debug interrupts; **rfmci** is used for machine check interrupts.

Guest supervisor software should use **rfi**, **rfci**, **rfdi**, and **rfmci** when returning from their associated interrupts. When a guest operating system executes **rfi**, the processor maps the instruction to **rfgi** assuring that the appropriate guest save/restore registers are used for the return. For **rfci**, **rfdi**, and **rfmci**, the hypervisor should emulate these instructions as it emulates the taking of these interrupts in guest supervisor state.

Privileges are as follows:

- **sc** is user privileged.
- **rfi** (**rfgi**), **mfmsr**, **mtmsr**, **wrtee**, **wrteei** are guest–supervisor privileged.
- **rfci**, **rfdi**, **rfmci** are hypervisor privileged.

Table 3-39. System Linkage Instructions—Supervisor-Level

Name	Mnemonic	Syntax	Implementation Notes
Return from Interrupt	**rfi**	—	These instructions are context-synchronizing, which for the e500mc means it works its way to the final execute stage, updates architected registers, and redirects instruction flow. In guest supervisor state (MSR[GS,PR]=0b10), **rfi (rfgi)** cannot alter MSR[GS] or any bits protected by MSRP. Guest supervisor state maps **rfi** to **rfgi**. Guest supervisor state cannot execute **rfci**, **rfdi**, or **rfmci** as they are hypervisor privileged and are emulated by the hypervisor.
Return from Guest Interrupt	**rfgi**	—	
Return from Critical Interrupt	**rfci**	—	
Return from Debug Interrupt	**rfdi**	—	
Return from Machine Check Interrupt	**rfmci**	—	
System Call	**sc**	LEV	

This table lists instructions for accessing the MSR.

Table 3-40. Move to/from Machine State Register Instructions

Name	Mnemonic	Syntax	Notes
Move from Machine State Register	**mfmsr**	rD	—
Move to Machine State Register	**mtmsr**	rS	In guest supervisor state (MSR[GS,PR]=0b10) **mtmsr** cannot alter MSR[GS] or any bits protected by MSRP.
Write MSR External Enable	**wrtee**	rS	—
Write MSR External Enable Immediate	**wrteei**	E	—

Certain encodings of the SPR field of **mtspr** and **mfspr** instructions (shown in Table 3-32) provide access to supervisor-level SPRs. Encodings for SPRs are listed in Table 2-2. Simplified mnemonics are provided for **mtspr** and **mfspr**. See Section 3.3.3, "Synchronization Requirements," and the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* for more information on context synchronization requirements when altering certain SPRs.

3.4.11.2 External PID Load Store Instructions

External PID load and store instructions are used by the operating system and hypervisor to perform load, store, and cache management instruction to a separate address space while still fetching and executing in the normal supervisor or hypervisor context. The operating system or hypervisor selects the address space to target by altering the contents of the EPLC and EPSC registers for loads and stores respectively. When the effective address specified by the external PID load or store instruction is translated, the translation mechanism uses ELPID, EPID, EAS, EPR, and EGS values from the EPLC or EPSC register instead of LPIDR, PID, MSR[DS], MSR[PR], and MSR[GS] values. Such instructions are useful for an operating system to access and manipulate virtual memory using the context and credentials of a process.

The external PID instructions are implemented as described in *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. Any implementation specific behaviors for external PID instructions is the same as the non external PID analogous instruction for e500mc (except that the translation mechanism is changed as described). See the appropriate description of the analogous instruction for any implementation specific details. All external PID instructions are guest supervisor privileged.

This table lists external PID load and store instructions.

Table 3-41. External PID Load Store Instructions

Instruction	Mnemonic	Syntax	Non External PID Analogous Instruction
Load Byte by External PID Indexed	**lbepx**	**r**D,**r**A,**r**B	**lbzx**
Load Floating-Point Doubleword by External PID Indexed	**lfdepx**	**fr**D,**r**A,**r**B	**lfdx**
Load Halfword by External PID Indexed	**lhepx**	**r**D,**r**A,**r**B	**lhzx**
Load Word by External PID Indexed	**lwepx**	**r**D,**r**A,**r**B	**lwzx**
Store Byte by External PID Indexed	**stbepx**	**r**S,**r**A,**r**B	**stbx**
Store Floating-Point Doubleword by External PID Indexed	**stfdepx**	**fr**S,**r**A,**r**B	**stfdx**
Store Halfword by External PID Indexed	**sthepx**	**r**S,**r**A,**r**B	**sthx**
Store Word by External PID Indexed	**stwepx**	**r**S,**r**A,**r**B	**stwx**
Data Cache Block Flush by External PID Indexed	**dcbfep**	**r**A,**r**B	**dcbf**
Data Cache Block Store by External PID Indexed	**dcbstep**	**r**A,**r**B	**dcbst**
Data Cache Block Touch by External PID Indexed	**dcbtep**	TH,**r**A,**r**B	**dcbt**
Data Cache Block Touch for Store by External PID Indexed	**dcbtstep**	TH,**r**A,**r**B	**dcbtst**
Data Cache Block Zero by External PID Indexed	**dcbzep**	**r**A,**r**B	**dcbz**
Data Cache Block Zero Long by External PID Indexed	**dcbzlep**	**r**A,**r**B	**dcbzl**
Instruction Cache Block Invalidate by External PID Indexed	**icbiep**	**r**A,**r**B	**icbi**

3.4.11.3 Supervisor-Level Memory Control Instructions

Memory control instructions include the following:

- Cache management instructions (supervisor-level and user-level)
- Translation lookaside buffer management instructions

This section describes supervisor-level memory control instructions. Section 3.4.10, "Memory Control Instructions," describes user-level memory control instructions.

3.4.11.3.1 Supervisor-Level Cache Instruction

Table 3-42 lists the supervisor-level cache management instructions except for cache management instructions which are part of the External PID instructions.

Table 3-42. Supervisor-Level Cache Management Instruction

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Invalidate	**dcbi**	**r**A,**r**B	**dcbi** executes as defined in the Power ISA but has implementation dependent behaviors. When the address to be invalidated is marked Memory Coherence Required (WIMGE = 0bx01xx), a **dcbf** is performed which first flushes the line if modified prior to invalidation. If the address is not marked as Memory Coherence Required (WIMGE=0bx00xx), the line is not flushed and is invalidated. In this case if the line was modified, the modified data is lost. In the e500mc, **dcbi** cannot generate a cache-locking exception**.**

dcbi is guest supervisor privileged.

See Section 3.4.10.1, "User-Level Cache Instructions," for cache instructions that provide user-level programs the ability to manage on-chip caches.

3.4.11.3.2 Supervisor-Level TLB Management Instructions

The address translation mechanism is defined in terms of TLBs and page table entries (PTEs) used to locate the logical-to-physical address mapping for an access. Chapter 6, "Memory Management Units (MMUs)," describes TLB operations. TLB management instructions are implemented as defined in *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

This table summarizes the operation of the TLB instructions in the e500mc.

Table 3-43. TLB Management Instructions

Name	Mnemonic	Syntax	Implementation Notes
TLB Invalidate Local	**tlbilx**	T,**r**A, **r**B	Invalidates TLB entries in the processor which executes the **tlbilx** instruction. TLB entries which are protected by the IPROT attribute ($entry_{IPROT}$=1) are not invalidated.**tlbilx** can be used to invalidate all entries corresponding to a LPID value, all entries corresponding to a PID value, or a single entry." **tlbilx** is guest supervisor privileged, however it causes an embedded hypervisor privilege exception if EPCR[DGTMI] is set. **Note: tlbilx** is the preferred way of performing TLB invalidations, especially for operating systems running as a guest to the hypervisor because invalidations are partitioned and do not require hypervisor privilege. **Note: tlbilx** requires the same local-processor synchronization as **tlbivax**, but not the cross-processor synchronization (that is, it does not require **tlbsync**).
TLB Invalidate Virtual Address Indexed	**tlbivax**	**r**A, **r**B	A TLB invalidate operation is performed whenever **tlbivax** is executed. **tlbivax** invalidates any TLB entry in the targeted TLB array that corresponds to the virtual address calculated by this instruction as long as IPROT is not set; this includes invalidating TLB entries contained in TLBs on other processors and devices in addition to the processor executing **tlbivax**. Thus, an invalidate operation is broadcast throughout the coherent domain of the processor executing **tlbivax**. For more information see Section 6.3, "Translation Lookaside Buffers (TLBs)." • **tlbivax** is hypervisor privileged.
TLB Read Entry	**tlbre**	—	**tlbre** causes the contents of a single TLB entry to be extracted from the MMU and be placed in the corresponding fields of the MAS registers. The entry extracted is specified by the TLBSEL, ESEL, and EPN fields of MAS0 and MAS2. The contents extracted from the MMU are placed in MAS0–MAS3, MAS7, and MAS8. If HID0[EN_MAS7_UPDATE] = 1, MAS7 is updated with the four highest-order bits of physical address for the TLB entry. See Section 6.3, "Translation Lookaside Buffers (TLBs)." **tlbre** is hypervisor privileged.
TLB Search Indexed	**tlbsx**	**r**A, **r**B	**tlbsx** searches the MMU for a particular entry based on the computed EA and the search values in MAS5 and MAS6.If a match is found, MAS1[V] is set and the found entry is read into the MAS0–MAS3, MAS7, and MAS8. If HID0[EN_MAS7_UPDATE] = 1, MAS7 is updated with the four highest-order bits of physical address for the TLB entry. If the entry is not found MAS1[V] is set to 0.See Section 6.3, "Translation Lookaside Buffers (TLBs)." **tlbsx** is hypervisor privileged. Note that **r**A=0 is a preferred form for **tlbsx** and that some Freescale implementations, including the e500mc, take an illegal instruction exception if **r**A != 0.
TLB Synchronize	**tlbsync**	—	Causes a TLBSYNC transaction on CoreNet interface. See Section 6.3, "Translation Lookaside Buffers (TLBs)." **tlbsync** is hypervisor privileged. Note that only one **tlbsync** can be in process at any given time on all processors of a coherence domain. The hypervisor or operating system should ensure this by doing the appropriate mutual exclusion. If e500mc detects multiple **tlbsync** operations at the same time, a machine check can occur.
TLB Write Entry	**tlbwe**	—	**tlbwe** causes the contents of certain fields of MAS0–MAS3, MAS7, and MAS8 to be written into a single TLB entry in the MMU. The entry written is specified by the TLBSEL, ESEL, and EPN fields of MAS0 and MAS2. Execution of **tlbwe** on the e500mc also causes the upper 4 bits of the RPN that reside in MAS7 to be written to the selected TLB entry. See Section 6.3, "Translation Lookaside Buffers (TLBs)." **tlbwe** is hypervisor privileged regardless of the setting of EPCR[DGTMI] as e500mc does not provide a Logical to Real Translation (LRAT) mechanism.

Implementation Notes

If an attempt is made to write a TLB1 entry and MAS1[TSIZE] specifies an invalid size (that is, 0 or 11 to 15), the entry is treated as if it is 4KB.

The TLB management instructions from Power ISA 2.06 contain a significant amount of optional capabilities. Although these capabilities are described in configuration registers, Freescale implementations only utilize a portion of the these capabilities. To minimize compatibility problems, system software should incorporate uses of these instructions into subroutines.

Executing **tlbsx** with **r**A != 0 causes an illegal instruction exception on e500mc. Software should always use **tlbsx** with **r**A = 0.

3.4.11.4 Message Clear and Message Send Instructions

The e500mc can generate messages to other processors and devices in the system. Messages are generated by using the Message Send (**msgsnd**) instruction. When a processor executes a **msgsnd** instruction that message is sent to all other processors in the coherence domain. Depending on the message type and the payload of the message (specified by **r**B), other processors that receive this message may take one of several types of doorbell interrupts. The e500mc accepts message types which generate the following doorbell interrupts:

- Processor doorbell
- Processor doorbell critical
- Guest processor doorbell
- Guest processor doorbell critical
- Guest processor doorbell machine check

See Section Table 4-30., “Message Types and Section 4.9.18.1, “Doorbell Interrupt Definitions. Messages that have already been accepted by a processor but have not caused one of the associated interrupts because the interrupt is masked may be cleared by the Message Clear (**msgclr**) instruction.

Both **msgsnd** and **msgclr** are implemented as described in the architecture and *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

More information about the doorbell interrupt types can be found in Section 4.9.18.1, “Doorbell Interrupt Definitions.”

Table 3-44. Message Clear and Message Send Instructions

Name	Mnemonic	Syntax
Message Clear	**msgclr**	**r**B
Message Send	**msgsnd**	**r**B

msgsnd and **msgclr** are hypervisor privileged.

3.4.11.5 Performance Monitor Instructions (Supervisor Level)

Software communicates with the performance monitor through performance monitor registers (PMRs) with the instructions listed in Table 3-45.

Table 3-45. Supervisor Performance Monitor Instructions

Name	Mnemonic	Syntax
Move from Performance Monitor Register	**mfpmr**	**r**D,PMRN
Move to Performance Monitor Register	**mtpmr**	PMRN,**r**S

Writing to a performance monitor register (**mtpmr**) requires guest supervisor privilege when MSRP[PMMP] = 0. If MSRP[PMMP] = 1, performance monitor registers are only accessible to the hypervisor. User level access is limited to read only access to certain registers through aliases designed to be accessed by user level software. Supervisor software can access these as well as all other defined performance monitor registers. Attempting to access an undefined performance monitor register causes an illegal instruction exception. PMRs are listed in Section 2.18, "Performance Monitor Registers (PMRs)."

3.4.12 Recommended Simplified Mnemonics

The description of each instruction includes the mnemonic and a formatted list of operands. Compliant assemblers support the mnemonics and operand listed. Simplified mnemonics and symbols is provided for frequently used instructions; refer to Appendix B, "Simplified Mnemonics," for a complete list. Programs written to be portable across the various assemblers should not assume the existence of mnemonics not described in this document.

3.4.13 Context Synchronization

Context synchronization is achieved by post- and presynchronizing instructions. An instruction is presynchronized by completing all instructions before dispatching the presynchronized instruction. Post-synchronizing is implemented by not dispatching any later instructions until the post-synchronized instruction is completely finished.

3.5 Debug Instruction Model

The Debugger Notify Halt instruction (**dnh**) is implemented as defined in Power ISA and *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. **dnh** can be used to halt the processor when an external debugger is attached and has enabled halting by setting EDBCR0[DNH_EN]. When the processor is halted, the DUI field is passed directly to the debugger as information describing the reason for the halt. The DUIS field can be extracted by the debugger if required for more detailed information. If an external debugger is not attached or has not enabled halting, **dnh** takes an illegal instruction exception.

Table 3-46. dnh Debug Instruction

Name	Mnemonic	Syntax	Implementation Details
Debugger Notify Halt	**dnh**	DUI,DUIS	—

3.6 Instruction Listing

This table lists the instructions implemented on the e500mc.

Table 3-47. e500mc Instruction Set

Mnemonic	Syntax	Classification	Cross-Reference
add	**r**D,**r**A,**r**B	Integer	Table 3-5
add.	**r**D,**r**A,**r**B	Integer	Table 3-5
addc	**r**D,**r**A,**r**B	Integer	Table 3-5
addc.	**r**D,**r**A,**r**B	Integer	Table 3-5
addco	**r**D,**r**A,**r**B	Integer	Table 3-5
addco.	**r**D,**r**A,**r**B	Integer	Table 3-5
adde	**r**D,**r**A,**r**B	Integer	Table 3-5
adde.	**r**D,**r**A,**r**B	Integer	Table 3-5
addeo	**r**D,**r**A,**r**B	Integer	Table 3-5
addeo.	**r**D,**r**A,**r**B	Integer	Table 3-5
addi	**r**D,**r**A,SIMM	Integer	Table 3-5
addic	**r**D,**r**A,SIMM	Integer	Table 3-5
addic.	**r**D,**r**A,SIMM	Integer	Table 3-5
addis	**r**D,**r**A,SIMM	Integer	Table 3-5
addme	**r**D,**r**A	Integer	Table 3-5
addme.	**r**D,**r**A	Integer	Table 3-5
addmeo	**r**D,**r**A	Integer	Table 3-5
addmeo.	**r**D,**r**A	Integer	Table 3-5
addo	**r**D,**r**A,**r**B	Integer	Table 3-5
addo.	**r**D,**r**A,**r**B	Integer	Table 3-5
addze	**r**D,**r**A	Integer	Table 3-5
addze.	**r**D,**r**A	Integer	Table 3-5
addzeo	**r**D,**r**A	Integer	Table 3-5
addzeo.	**r**D,**r**A	Integer	Table 3-5
and	**r**A,**r**S,**r**B	Integer logical	Table 3-7
and.	**r**A,**r**S,**r**B	Integer logical	Table 3-7
andc	**r**A,**r**S,**r**B	Integer logical	Table 3-7
andc.	**r**A,**r**S,**r**B	Integer logical	Table 3-7
andi.	**r**A,**r**S,UIMM	Integer logical	Table 3-7
andis.	**r**A,**r**S,UIMM	Integer logical	Table 3-7

Table 3-47. e500mc Instruction Set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
b	LI	Branch	Table 3-25
ba	LI	Branch	Table 3-25
bc	BO,BI,BD	Branch	Table 3-25
bca	BO,BI,BD	Branch	Table 3-25
bcctr	BO,BI	Branch	Table 3-25
bcctrl	BO,BI	Branch	Table 3-25
bcl	BO,BI,BD	Branch	Table 3-25
bcla	BO,BI,BD	Branch	Table 3-25
bclr	BO,BI	Branch	Table 3-25
bclrl	BO,BI	Branch	Table 3-25
bl	LI	Branch	Table 3-25
bla	LI	Branch	Table 3-25
cmp	**crf**D,L,**r**A,**r**B	Compare	Table 3-6
cmpi	**crf**D,L,**r**A,SIMM	Compare	Table 3-6
cmpl	**crf**D,L,**r**A,**r**B	Compare	Table 3-6
cmpli	**crf**D,L,**r**A,UIMM	Compare	Table 3-6
cntlzw	**r**A,**r**S	Integer logical	Table 3-7
cntlzw.	**r**A,**r**S	Integer logical	Table 3-7
crand	**crb**D,**crb**A,**crb**B	Condition register logical	Table 3-6
crandc	**crb**D,**crb**A,**crb**B	Condition register logical	Table 3-6
creqv	**crb**D,**crb**A,**crb**B	Condition register logical	Table 3-6
crnand	**crb**D,**crb**A,**crb**B	Condition register logical	Table 3-6
crnor	**crb**D,**crb**A,**crb**B	Condition register logical	Table 3-6
cror	**crb**D,**crb**A,**crb**B	Condition register logical	Table 3-6
crorc	**crb**D,**crb**A,**crb**B	Condition register logical	Table 3-6
crxor	**crb**D,**crb**A,**crb**B	Condition register logical	Table 3-6
dcba	**r**A,**r**B	Cache control	Table 3-36
dcbal	**r**A,**r**B	Extended cache line/cache control	Table 3-36
dcbf	**r**A,**r**B	Cache control	Table 3-36
dcbfep	**r**A,**r**B	External PID load/store	Table 3-41
dcbi	**r**A,**r**B	Cache control	Table 3-36
dcblc	CT,**r**A,**r**B	Cache locking	Table 3-37
dcbst	**r**A,**r**B	Cache control	Table 3-36

Table 3-47. e500mc Instruction Set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
dcbstep	**r**A,**r**B	External PID load/store	Table 3-41
dcbt	TH,**r**A,**r**B	Cache control	Table 3-36
dcbtep	TH,**r**A,**r**B	External PID load/store	Table 3-41
dcbtls	CT,**r**A,**r**B	Cache locking	Table 3-37
dcbtst	CT,**r**A,**r**B	Cache control	Table 3-36
dcbtstep	TH,**r**A,**r**B	External PID load/store	Table 3-41
dcbtstls	CT,**r**A,**r**B	Cache locking	Table 3-37
dcbz	**r**A,**r**B	Cache control	Table 3-36
dcbzep	**r**A,**r**B	External PID load/store	Table 3-41
dcbzl	**r**A,**r**B	Extended cache line/cache control	Table 3-36
dcbzlep	**r**A,**r**B	External PID load/store	Table 3-41
divw	**r**D,**r**A,**r**B	Integer	Table 3-5
divw.	**r**D,**r**A,**r**B	Integer	Table 3-5
divwo	**r**D,**r**A,**r**B	Integer	Table 3-5
divwo.	**r**D,**r**A,**r**B	Integer	Table 3-5
divwu	**r**D,**r**A,**r**B	Integer	Table 3-5
divwu.	**r**D,**r**A,**r**B	Integer	Table 3-5
divwuo	**r**D,**r**A,**r**B	Integer	Table 3-5
divwuo.	**r**D,**r**A,**r**B	Integer	Table 3-5
dnh	DUI,DUIS	Debug	Table 3-46
dsn	**r**A,**r**B	Decorated load/store	Table 3-16
ehpriv	OC	Hypervisor	Table 3-30.”
eqv	**r**A,**r**S,**r**B	Integer logical	Table 3-7
eqv.	**r**A,**r**S,**r**B	Integer logical	Table 3-7
extsb	**r**A,**r**S	Integer logical	Table 3-7
extsb.	**r**A,**r**S	Integer logical	Table 3-7
extsh	**r**A,**r**S	Integer logical	Table 3-7
extsh.	**r**A,**r**S	Integer logical	Table 3-7
fabs	**fr**D,**fr**B	Floating-point	Table 3-17
fabs.	**fr**D,**fr**B	Floating-point	Table 3-17
fadd	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-17
fadd.	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-17
fadds	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-17

Table 3-47. e500mc Instruction Set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
fadds.	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-17
fcmpo	**crf**D,**fr**A,**fr**B	Floating-point	Table 3-17
fcmpu	**crf**D,**fr**A,**fr**B	Floating-point	Table 3-17
fctiw	**fr**D,**fr**B	Floating-point	Table 3-17
fctiw.	**fr**D,**fr**B	Floating-point	Table 3-17
fctiwz	**fr**D,**fr**B	Floating-point	Table 3-17
fctiwz.	**fr**D,**fr**B	Floating-point	Table 3-17
fdiv	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-17
fdiv.	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-17
fdivs	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-17
fdivs.	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-17
fmadd	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fmadd.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fmadds	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fmadds.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fmr	**fr**D,**fr**B	Floating-point	Table 3-22
fmr.	**fr**D,**fr**B	Floating-point	Table 3-22
fmsub	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-17
fmsub.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-17
fmsubs	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-17
fmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-17
fmul	**fr**D,**fr**A,**fr**C	Floating-point	Table 3-17
fmul.	**fr**D,**fr**A,**fr**C	Floating-point	Table 3-17
fmuls	**fr**D,**fr**A,**fr**C	Floating-point	Table 3-17
fmuls.	**fr**D,**fr**A,**fr**C	Floating-point	Table 3-17
fnabs	**fr**D,**fr**B	Floating-point	Table 3-17
fnabs.	**fr**D,**fr**B	Floating-point	Table 3-17
fneg	**fr**D,**fr**B	Floating-point	Table 3-17
fneg.	**fr**D,**fr**B	Floating-point	Table 3-17
fnmadd	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fnmadd.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fnmadds	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fnmadds.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18

Table 3-47. e500mc Instruction Set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
fnmsub	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fnmsub.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fnmsubs	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fnmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fres	**fr**D,**fr**B	Floating-point	Table 3-18
fres.	**fr**D,**fr**B	Floating-point	Table 3-18
frsp	**fr**D,**fr**B	Floating-point	Table 3-18
frsp.	**fr**D,**fr**B	Floating-point	Table 3-18
frsqrte	**fr**D,**fr**B	Floating-point	Table 3-18
frsqrte.	**fr**D,**fr**B	Floating-point	Table 3-18
fsel	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fsel.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-18
fsub	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-18
fsub.	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-18
fsubs	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-18
fsubs.	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-18
icbi	**fr**A,**fr**B	Cache control	Table 3-36
icbiep	**r**A,**r**B	External PID load/store	Table 3-41
icblc	CT,**r**A,**r**B	Cache locking	Table 3-37
icbt	CT,**r**A,**r**B	Cache control	Table 3-36
icbtls	CT,**r**A,**r**B	Cache locking	Table 3-37
isel	**r**D,**r**A,**r**B,**crb**C	Integer select	Table 3-26
isync	—	Synchronization	Table 3-35
lbdx	**r**D,**r**A,**r**B	Decorated load/store	Table 3-16
lbepx	**r**D,**r**A,**r**B	External PID load/store	Table 3-41
lbz	**r**D,**d(r**A**)**	Integer load	Table 3-10
lbzu	**r**D,**d(r**A**)**	Integer load	Table 3-10
lbzux	**r**D,**r**A,**r**B	Integer load	Table 3-10
lbzx	**r**D,**r**A,**r**B	Integer load	Table 3-10
lfd	**fr**D,**d(r**A**)**	Floating-point load/store	Table 3-14
lfddx	**fr**D,**r**A,**r**B	Decorated load/store	Table 3-16
lfdepx	**fr**D,**r**A,**r**B	External PID load/store	Table 3-41
lfdu	**fr**D,**d(r**A**)**	Floating-point load/store	Table 3-14

Table 3-47. e500mc Instruction Set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
lfdux	**fr**D,**r**A,**r**B	Floating-point load/store	Table 3-14
lfdx	**fr**D,**r**A,**r**B	Floating-point load/store	Table 3-14
lfs	**fr**D,**d(r**A**)**	Floating-point load/store	Table 3-14
lfsu	**fr**D,**d(r**A**)**	Floating-point load/store	Table 3-14
lfsux	**fr**D,**r**A,**r**B	Floating-point load/store	Table 3-14
lfsx	**fr**D,**r**A,**r**B	Floating-point load/store	Table 3-14
lha	**r**D,**d(r**A**)**	Integer load	Table 3-10
lhau	**r**D,**d(r**A**)**	Integer load	Table 3-10
lhaux	**r**D,**r**A,**r**B	Integer load	Table 3-10
lhax	**r**D,**r**A,**r**B	Integer load	Table 3-10
lhbrx	**r**D,**r**A,**r**B	Integer load/store w/byte reverse	Table 3-12
lhdx	**r**D,**r**A,**r**B	Decorated load/store	Table 3-16
lhepx	**r**D,**r**A,**r**B	External PID load/store	Table 3-41
lhz	**r**D,**d(r**A**)**	Integer load	Table 3-10
lhzu	**r**D,**d(r**A**)**	Integer load	Table 3-10
lhzux	**r**D,**r**A,**r**B	Integer load	Table 3-10
lhzx	**r**D,**r**A,**r**B	Integer load	Table 3-10
lmw	**r**D,**d(r**A**)**	Integer load	Table 3-10
lwarx	**r**D,**r**A,**r**B	Synchronization	Table 3-35
lwbrx	**r**D,**r**A,**r**B	Integer load/store w/byte reverse	Table 3-12
lwdx	**r**D,**r**A,**r**B	Decorated load/store	Table 3-16
lwepx	**r**D,**r**A,**r**B	External PID load/store	Table 3-41
lwz	**r**D,**d(r**A**)**	Integer load	Table 3-10
lwzu	**r**D,**d(r**A**)**	Integer load	Table 3-10
lwzux	**r**D,**r**A,**r**B	Integer load	Table 3-10
lwzx	**r**D,**r**A,**r**B	Integer load	Table 3-10
mbar	—	Synchronization	Table 3-35
mcrf	**crf**D,**crf**S	Condition register logical	Table 3-6
mcrfs	**crf**D,**crf**S_FP	Condition register logical	Table 3-6
mcrxr	**crf**D	Condition register logical	Table 3-31
mfcr	**r**D	Condition register logical	Table 3-31
mffs	**fr**D	FPSCR	Table 3-21
mffs.	**fr**D	FPSCR	Table 3-21

Table 3-47. e500mc Instruction Set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
mfmsr	**r**D	MSR	Table 3-40
mfocrf	**r**D,C**R**M	CR logical	Table 3-31
mfpmr	**r**D,PM**R**N	Move from PMR	Table 3-45
mfspr	**r**D,SP**R**	SPR	Table 3-32
mftb	**r**D,SP**R**	Move from time base	Table 3-32
msgclr	**r**B	Doorbell	Table 3-44
msgsnd	**r**B	Doorbell	Table 3-44
mtcrf	C**R**M,**r**S	Condition register logical	Table 3-31
mtfsb0	**crb**D_FP	FPSCR	Table 3-21
mtfsb0.	**crb**D_FP	FPSCR	Table 3-21
mtfsb1	**crb**D_FP	FPSCR	Table 3-21
mtfsb1.	**crb**D_FP	FPSCR	Table 3-21
mtfsf	FM,fB	FPSCR	Table 3-21
mtfsf.	FM,fB	FPSCR	Table 3-21
mtfsfi	**crf**D_FP,FP_IMM	FPSCR	Table 3-21
mtfsfi.	**crf**D_FP,FP_IMM	FPSCR	Table 3-21
mtmsr	**r**S	MSR	Table 3-40
mtocrf	C**R**M,**r**S	CR logical	Table 3-31
mtpmr	PM**R**N,**r**S	Move to PMR	Table 3-34
mtspr	SP**R**,**r**S	SPR	Table 3-32
mulhw	**r**D,**r**A,**r**B	Integer	Table 3-5
mulhw.	**r**D,**r**A,**r**B	Integer	Table 3-5
mulhwu	**r**D,**r**A,**r**B	Integer	Table 3-5
mulhwu.	**r**D,**r**A,**r**B	Integer	Table 3-5
mulli	**r**D,**r**A,SIMM	Integer	Table 3-5
mullw	**r**D,**r**A,**r**B	Integer	Table 3-5
mullw.	**r**D,**r**A,**r**B	Integer	Table 3-5
mullwo	**r**D,**r**A,**r**B	Integer	Table 3-5
mullwo.	**r**D,**r**A,**r**B	Integer	Table 3-5
nand	**r**A,**r**S,**r**B	Integer logical	Table 3-7
nand.	**r**A,**r**S,**r**B	Integer logical	Table 3-7
neg	**r**D,**r**A	Integer	Table 3-5
neg.	**r**D,**r**A	Integer	Table 3-5

Table 3-47. e500mc Instruction Set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
nego	**r**D,**r**A	Integer	Table 3-5
nego.	**r**D,**r**A	Integer	Table 3-5
nor	**r**A,**r**S,**r**B	Integer logical	Table 3-7
nor.	**r**A,**r**S,**r**B	Integer logical	Table 3-7
or	**r**A,**r**S,**r**B	Integer logical	Table 3-7
or.	**r**A,**r**S,**r**B	Integer logical	Table 3-7
orc	**r**A,**r**S,**r**B	Integer logical	Table 3-7
orc.	**r**A,**r**S,**r**B	Integer logical	Table 3-7
ori	**r**A,**r**S,UIMM	Integer logical	Table 3-7
oris	**r**A,**r**S,UIMM	Integer logical	Table 3-7
rfci	—	System Linkage	Table 3-39
rfdi	—	System Linkage	Table 3-39
rfgi	—	System Linkage	Table 3-39
rfi	—	System Linkage	Table 3-39
rfmci	—	System Linkage	Table 3-39
rlwimi	**r**A,**r**S,SH,MB,ME	Integer rotate	Table 3-8
rlwimi.	**r**A,**r**S,SH,MB,ME	Integer rotate	Table 3-8
rlwinm	**r**A,**r**S,SH,MB,ME	Integer rotate	Table 3-8
rlwinm.	**r**A,**r**S,SH,MB,ME	Integer rotate	Table 3-8
rlwnm	**r**A,**r**S,**r**B,MB,ME	Integer rotate	Table 3-8
rlwnm.	**r**A,**r**S,**r**B,MB,ME	Integer rotate	Table 3-8
sc	LEV	System call	Table 3-8
slw	**r**A,**r**S,**r**B	Integer shift	Table 3-9
slw.	**r**A,**r**S,**r**B	Integer shift	Table 3-9
sraw	**r**A,**r**S,**r**B	Integer shift	Table 3-9
sraw.	**r**A,**r**S,**r**B	Integer shift	Table 3-9
srawi	**r**A,**r**S,SH	Integer shift	Table 3-9
srawi.	**r**A,**r**S,SH	Integer shift	Table 3-9
srw	**r**A,**r**S,**r**B	Integer shift	Table 3-9
srw.	**r**A,**r**S,**r**B	Integer shift	Table 3-9
stb	**r**S,**d(r**A**)**	Integer store	Table 3-11
stbdx	**r**S,**r**A,**r**B	Decorated load/store	Table 3-16
stbepx	**r**S,**r**A,**r**B	External PID load/store	Table 3-41

Table 3-47. e500mc Instruction Set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
stbu	**r**S,**d(r**A**)**	Integer store	Table 3-11
stbux	**r**S,**r**A,**r**B	Integer store	Table 3-11
stbx	**r**S,**r**A,**r**B	Integer store	Table 3-11
stfd	**fr**S,**d(r**A**)**	Floating-point store	Table 3-15
stfddx	**fr**S,**r**A,**r**B	Decorated load/store	Table 3-16
stfdepx	**fr**S,**r**A,**r**B	External PID load/store	Table 3-41
stfdu	**fr**S,**d(r**A**)**	Floating-point store	Table 3-15
stfdux	**fr**S,**r**A,**r**B	Floating-point store	Table 3-15
stfdx	**fr**S,**r**A,**r**B	Floating-point store	Table 3-15
stfiwx	**fr**S,**r**A,**r**B	Floating-point store	Table 3-15
stfs	**fr**S,**d(r**A**)**	Floating-point store	Table 3-15
stfsu	**fr**S,**d(r**A**)**	Floating-point store	Table 3-15
stfsux	**fr**S,**r**A,**r**B	Floating-point store	Table 3-15
stfsx	**fr**S,**r**A,**r**B	Floating-point store	Table 3-15
sth	**r**S,**d(r**A**)**	Integer store	Table 3-11
sthbrx	**r**S,**r**A,**r**B	Integer load/store w/byte reverse	Table 3-12
sthdx	**r**S,**r**A,**r**B	Decorated load/store	Table 3-16
sthepx	**r**S,**r**A,**r**B	External PID load/store	Table 3-41
sthu	**r**S,**d(r**A**)**	Integer store	Table 3-11
sthux	**r**S,**r**A,**r**B	Integer store	Table 3-11
sthx	**r**S,**r**A,**r**B	Integer store	Table 3-11
stmw	**r**S,**d(r**A**)**	Integer store	Table 3-11
stw	**r**S,**d(r**A**)**	Integer store	Table 3-11
stwbrx	**r**S,**r**A,**r**B	Integer load/store w/byte reverse	Table 3-12
stwcx.	**r**S,**r**A,**r**B	Synchronization	Table 3-35
stwdx	**r**S,**r**A,**r**B	Decorated load/store	Table 3-16
stwepx	**r**S,**r**A,**r**B	External PID load/store	Table 3-41
stwu	**r**S,**d(r**A**)**	Integer store	Table 3-11
stwux	**r**S,**r**A,**r**B	Integer store	Table 3-11
stwx	**r**S,**r**A,**r**B	Integer store	Table 3-11
subf	**r**D,**r**A,**r**B	Integer	Table 3-5
subf.	**r**D,**r**A,**r**B	Integer	Table 3-5
subfc	**r**D,**r**A,**r**B	Integer	Table 3-5

Table 3-47. e500mc Instruction Set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
subfc.	**r**D,**r**A,**r**B	Integer	Table 3-5
subfco	**r**D,**r**A,**r**B	Integer	Table 3-5
subfco.	**r**D,**r**A,**r**B	Integer	Table 3-5
subfe	**r**D,**r**A,**r**B	Integer	Table 3-5
subfe.	**r**D,**r**A,**r**B	Integer	Table 3-5
subfeo	**r**D,**r**A,**r**B	Integer	Table 3-5
subfeo.	**r**D,**r**A,**r**B	Integer	Table 3-5
subfic	**r**D,**r**A,SIMM	Integer	Table 3-5
subfme	**r**D,**r**A	Integer	Table 3-5
subfme.	**r**D,**r**A	Integer	Table 3-5
subfmeo	**r**D,**r**A	Integer	Table 3-5
subfmeo.	**r**D,**r**A	Integer	Table 3-5
subfo	**r**D,**r**A,**r**B	Integer	Table 3-5
subfo.	**r**D,**r**A,**r**B	Integer	Table 3-5
subfze	**r**D,**r**A	Integer	Table 3-5
subfze.	**r**D,**r**A	Integer	Table 3-5
subfzeo	**r**D,**r**A	Integer	Table 3-5
subfzeo.	**r**D,**r**A	Integer	Table 3-5
sync (msync)	L	Synchronization	Table 3-35
tlbilx	T,**r**A,**r**B	TLB management	Table 3-43
tlbivax	**r**A,**r**B	TLB management	Table 3-43
tlbre	—	TLB management	Table 3-43
tlbsx	**r**A,**r**B	TLB management	Table 3-43
tlbsync	—	TLB management	Table 3-43
tlbwe	—	TLB management	Table 3-43
tw	TO,**r**A,**r**B	Trap	Table 3-28
twi	TO,**r**A,SIMM	Trap	Table 3-28
wait	—	Wait	Table 3-35
wrtee	**r**S	MSR	Table 3-40
wrteei	E	MSR	Table 3-40
xor	**r**A,**r**S,**r**B	Integer logical	Table 3-7
xor.	**r**A,**r**S,**r**B	Integer logical	Table 3-7

Table 3-47. e500mc Instruction Set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
xori	**r**A,**r**S,UIMM	Integer logical	Table 3-7
xoris	**r**A,**r**S,UIMM	Integer logical	Table 3-7

Chapter 4
Interrupts and Exceptions

This chapter provides a general description of the interrupt and exception model as it is implemented in the e500mc core. It identifies and describes the portions of the interrupt model that are defined by the architecture and those that are specific to the e500mc.

4.1 Interrupts and Exceptions Overview

The terms “interrupt” and “exception” are used as follows:

Interrupt — An interrupt is the action in which the processor saves its context (typically the machine state register (MSR) and next instruction address) and begins execution at a predetermined interrupt handler address with a modified MSR.

Exception — An exception is the event that, if enabled, may cause the processor to take an interrupt. Multiple exceptions may occur during the execution of an instruction and the exception priority mechanism determines which of the exceptions cause an associated interrupt. In some cases when an asynchronous exception has occurred, but the associated interrupt is not enabled, other actions within the processor may clear the exception condition prior to it being enabled, which prevents the associated interrupt from occurring. The architecture describes exceptions as being generated by instructions, the internal timer facility, debug events, error conditions, and signals from internal and external peripherals.

There are four categories of interrupts, which are described in the following sections:

- Standard Interrupts
- Critical Interrupts
- Debug Interrupts
- Machine Check Interrupts

4.1.1 Standard Interrupts

Standard interrupts are first-level interrupts that allow the processor to change program flow to handle conditions generated by external signals, errors, or conditions arising from program execution or from programmable timer-related events. These interrupts are largely identical to those defined originally by the PowerPC OEA. They use save and restore registers (SRR0/SRR1) to save state when they are taken and they use the **rfi** instruction to restore state. Asynchronous, noncritical interrupts can be masked by the external interrupt enable bit, MSR[EE] (when not in guest state).

Guest interrupts are standard interrupts that are handled by guest-supervisor software. They use guest save and restore registers (GSRR0/GSRR1) to save state when they are taken and they use the **rfgi** instruction to restore state. Guest interrupts are listed in Table 2-5.

Section 2.3, "Register Mapping in Guest–Supervisor State," describes how accesses to non-guest registers are changed by the processor to their guest register equivalents when MSR[PR] = 0 and MSR[GS] = 1.

4.1.2 Critical Interrupts

Critical interrupts are logically higher priority than standard interrupts (critical input, processor doorbell critical, guest processor doorbell critical, and watchdog timer interrupts) and can be taken during regular program flow or during a standard interrupt. They use the critical save and restore registers (CSRR0/CSRR1) and the **rfci** instruction.

Critical interrupts can be masked by the critical enable bit, MSR[CE] (when not in guest state).

4.1.3 Debug Interrupts

Debug interrupts are logically a higher priority than critical interrupts and can be taken during regular program flow, during a standard interrupt, or during a critical interrupt. They use the debug save and restore registers (DSRR0/DSRR1) and the **rfdi** instruction. See Section 4.9.16, "Debug Interrupt—IVOR15." Debug interrupts can be masked by the debug enable bit, MSR[DE] (when not in guest state).

4.1.4 Machine Check Interrupts

Machine check interrupts are logically a higher priority than critical interrupts and can be taken during regular program flow, during a standard interrupt, during a critical interrupt, or during a debug interrupt. They use the machine check save and restore registers (MCSRR0/MCSRR1) and the **rfmci** instruction. See Section 4.9.3, "Machine Check Interrupt—IVOR1." Machine check interrupts can be masked by the machine check enable bit, MSR[ME] (when not in guest state).

The e500mc also implements precise synchronous machine check error report interrupts as well as an asynchronous non-maskable interrupt (NMI) which are not masked by MSR[ME]. For e500mc details, see Section 4.9.3, "Machine Check Interrupt—IVOR1."

4.1.5 Special Considerations for Interrupts and Exceptions

The masking of asynchronous interrupts using the MSR bits EE, CE, DE, or ME only prevents the associated asynchronous interrupts in the current state (guest or hypervisor). With the exception of guest processor doorbell, guest processor doorbell critical, and guest processor doorbell machine check, such masking depends on which state the interrupt is directed to as described in Section 4.3, "Directed Interrupts." This means that an asynchronous interrupt that is masked while executing in the guest state (its associated masking bit is 0) can still be taken if that interrupt is directed to the hypervisor state. Guest processor doorbell, guest processor doorbell critical, and guest processor doorbell machine check require that the processor be executing in the guest state and the appropriate interrupt enable bit is set even though these interrupts are always directed to the hypervisor state.

All asynchronous interrupts except the NMI interrupt are ordered because each type of interrupt has its own set of save/restore registers. Only one interrupt of each category is reported (standard, critical, debug, machine check, and guest), and when it is processed (taken) no program state is lost. Program state may be lost if synchronous exceptions occur within the interrupt handler for those same synchronous exceptions before software has successfully saved the state of the save/restore registers. For example, executing an illegal instruction as the first instruction of the program interrupt handler causes another program interrupt changing the state of the SRR0/SRR1 registers before software can save them thus destroying the return path to the original interrupt. (See Section 4.6.1, "Interrupt Ordering and Masking.")

All interrupts except the machine check interrupt are context synchronizing, as defined in the instruction model chapter of the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. A machine check interrupt acts like a context-synchronizing operation with respect to subsequent instructions.

4.2 e500mc Implementation of Interrupt Architecture

This section describes the architecture-defined interrupt model as implemented on the e500mc. Specific details are also provided throughout this chapter. The e500mc implements all the interrupts defined by the embedded category and the e500mc implements the following interrupts that are defined by, but not required by, optional parts of the embedded architecture:

- In general, the e500mc implements the machine check interrupt as it is defined by Power ISA 2.06, but extends the definition to include synchronous error reports and a non-maskable interrupt (NMI).
  The e500mc implements three types of machine check interrupts: asynchronous, error report, and NMI. Asynchronous machine check events are logged directly into the MCSR. If such an event is logged in the MCSR and MSR[ME] = 1 or MSR[GS] = 1, a machine check interrupt is taken. But if some of the MCSR's asynchronous bits have been set and MSR[ME] = 0 and MSR[GS] = 0, the asynchronous machine check exception is pending and if these bits are still set when MSR[ME] or MSR[GS] is changed to 1, the asynchronous machine check interrupt occurs. The e500mc does not take a checkstop, as is the case with previous e500 cores.
  In addition. the e500mc implements error report machine check exceptions (which are recorded in certain defined MCSR bits). Error report machine check interrupts are not gated by MSR[ME] (or MSR[GS]), and are synchronous and precise: They occur only if there is an error condition on an instruction that would otherwise complete execution, and not for instructions that have not completed and deallocated. For example, the core does not take an error report on an instruction in a mispredicted branch path or on an instruction that gets flushed by some other interrupt (such as an asynchronous machine check interrupt).
- The e500mc implements debug interrupts as described by the embedded enhanced debug category which provides a separate set of debug save/restore registers (DSRR0 and DSRR1).
- The e500mc implements the performance monitor interrupt from the embedded performance monitor category.
- The e500mc implements the enabled floating-point exception (program interrupt) and the floating-point unavailable interrupt from the floating-point category.
- The e500mc implements the following interrupts defined by the embedded processor control category:

  - Processor doorbell
  - Processor doorbell critical
- The e500mc implements the following interrupts defined by the embedded hypervisor category:
  - Hypervisor privilege
  - Hypervisor system call
  - Guest processor doorbell interrupt
  - Guest processor doorbell critical interrupt
  - Guest processor doorbell machine check interrupt
- The e500mc does not implement the unimplemented operation exception of the program interrupt. All unimplemented instructions take an illegal instruction exception.
- Interrupt priorities differ from those specified in the architecture as described in Section 4.11, "Interrupt Priorities."

4.3 Directed Interrupts

Interrupts on e500mc are directed to either the guest state or the hypervisor state. The state to which interrupts are directed determines which SPRs are used to form the vector address, which save/restore registers are used to capture the processor state at the time of the interrupt, and which ESR is used to post exception status. Interrupts directed to the guest state use the GIVPR to determine the upper 48 bits of the vector address and use GIVORs to provide the lower 16 bits. Interrupts directed to the hypervisor state use the IVPR and the IVORs. Interrupts that are directed to the guest state use GSRR0/GSRR1 registers to save the context at interrupt time. Interrupts directed to the hypervisor state use SRR0/SRR1, CSRR0/CSRR1, DSRR0/DSRR1, and MCSRR0/MCSRR1 for standard, critical, debug, and machine check interrupts respectively, with the exception of guest processor doorbell interrupts which use GSRR0/GSRR1.

In general, all interrupts are directed to the hypervisor state except for the following cases:

- The system call interrupt is directed to the state from which the interrupt was taken. If an **sc 0** instruction is executed in guest state, the interrupt is directed to the guest state. If an **sc 0** instruction is executed in hypervisor state, the interrupt is directed to the hypervisor state. Note that **sc 1** is always directed to the hypervisor state and produces a hypervisor system call interrupt.
- One of the following interrupts occurs while the processor is in the guest state, and the associated control bit in the EPCR is set to configure the interrupt to be directed to the guest state:
  - External input (EPCR[EXTGS] = 1)
  - Data TLB error (EPCR[DTLBGS] = 1)
  - Instruction TLB error (EPCR[ITLBGS] = 1)
  - Data storage (EPCR[DSIGS] = 1 and TLB[VF] = 0 [virtualization fault])
  - Instruction storage (EPCR[ISIGS] = 1)

NOTE

A data storage interrupt caused by a virtualization fault exception is always taken in the hypervisor state.

In no case is an interrupt directed to the guest when the processor is executing in the hypervisor state.

For more specific information about how interrupts are directed, see *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* or Power ISA 2.06.

4.4 Recoverability and MSR[RI]

MSR[RI] is an MSR (and save/restore register) storage bit for compatibility with pre-Book E PowerPC processors. When an interrupt occurs, the recoverable interrupt bit, MSR[RI] is unchanged by the interrupt mechanism when a new MSR is established; however, when a machine check, error report or NMI interrupt occurs, MSR[RI] is cleared.

If used properly, RI determines whether an interrupt that is taken at the machine check interrupt vector can be safely returned from (that is, that architected state set by the interrupt mechanism has been safely stored by software). RI should be set by software when all MSR values are first established. When an interrupt occurs that is taken at the machine check interrupt vector, software should set RI when it has safely stored MCSRR0 and MCSRR1. The associated MCSRR1 bit should be checked to determine whether the interrupt occurred when another machine check interrupt was being processed and before state was successfully saved. If MCSRR1[RI] is set, it is safe to return when processing is complete.

4.5 Interrupt Registers

Table 4-1 summarizes registers used for interrupt handling. The *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* provides full descriptions.

NOTE

In this manual, references to *x*SRR0 and *x*SRR1 apply to the respective (standard, critical, machine check, and guest) save restore 0 and save restore 1 registers. References to (G)*register* refer to *register* if the interrupt is taken in hypervisor state, or G*register* if the interrupt is taken in guest state (for example (G)DEAR refers to DEAR and GDEAR registers).

Whether the interrupt is directed to hypervisor or guest–supervisor software is described in see Section 4.3, "Directed Interrupts," for more details.

Table 4-1. Interrupt Registers

Register	Description
Save/restore register 0 (SRR0, CSRR0, DSRR0, GSRR0, MCSRR0)	On an interrupt, *x*SRR0 holds the EA at which execution continues when the corresponding return from interrupt instruction executes. Typically, this is the EA of the instruction that caused the interrupt or the subsequent instruction.
Save/restore register 1 (SRR1, CSRR1, DSRR1, GSRR1, MCSRR1)	When an interrupt is taken, MSR contents are placed into *x*SRR1. When the return from interrupt (**rfi**, **rfgi**, **rfci**, **rfdi**, **rfmci**) instruction executes, the values are restored to the MSR from *x*SRR1. *x*SRR1 bits that correspond to reserved MSR bits are also reserved. Note that an MSR bit that is reserved may be altered by a return from interrupt instruction.

Table 4-1. Interrupt Registers (continued)

Register	Description
Data exception address register (DEAR/GDEAR)	Contains the address referenced by a load, store, or cache management instruction that caused an alignment, data TLB miss, or data storage interrupt. When executing in the guest state (MSR[GS] = 1), accesses to the DEAR are mapped to GDEAR upon executing **mtspr** or **mfspr**. DEAR and GDEAR are described in Section 2.9.2, "(Guest) Data Exception Address Register (DEAR/GDEAR)."
Exception proxy register (EPR/GEPR)	Defined by the external interrupt proxy facility, which is described in Section 4.9.6.3, "External Proxy." EPR is used to convey the peripheral-specific interrupt vector associated with the external input interrupt triggered by the programmable interrupt controller (PIC) in the integrated device. When executing in the guest state (MSR[GS] = 1), accesses to the EPR are mapped to GEPR upon executing **mfspr**. EPR and GEPR are described in Section 2.9.5, "(Guest) External Proxy Register (EPR/GEPR)."
Interrupt vector prefix register (IVPR/GIVPR)	(G)IVPR[32-47] provides the high-order 16 bits of the address of the interrupt handling routine for each interrupt type. The 16-bit vector offsets are concatenated to the right of (G)IVPR to form the address of the interrupt handling routine.
Exception syndrome register (ESR/GESR)	Identifies a syndrome for differentiating exception conditions that can generate the same interrupt. When such an exception occurs, corresponding (G)ESR bits are set and all others are cleared. Other interrupt types do not affect the (G)ESR. (G)ESR does not need to be cleared by software. When executing in the guest state (MSR[GS] = 1), accesses to the ESR are mapped to GESR upon executing **mtspr** or **mfspr**. See Section 2.9.6, "(Guest) Exception Syndrome Register (ESR/GESR)."
Interrupt vector offset registers (IVORs/GIVORs)	Holds the quad-word index from the base address provided by the (G)IVPR for each interrupt type. Table 4-2 lists the (G)IVOR assignments for the e500mc core. Supported (G)IVORs and the associated interrupts are listed in Table 4-2.
Machine check address register (MCAR/MCARU)	On a machine check interrupt, MCAR/MCARU is updated with the address of the data associated with the machine check if applicable. See Section 2.9.8, "Machine Check Address Register (MCAR/MCARU)."
Machine check syndrome register (MCSR)	On a machine check interrupt, MCSR is updated with a syndrome to indicate exceptions, listed in Table 2-8 and fully described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. Section 2.9.9, "Machine Check Syndrome Register (MCSR)," describes MCSR bit fields as they are defined for the e500mc.

NOTE

System software may need to identify the type of instruction that caused the interrupt and examine the TLB entry and ESR to fully identify the exception or exceptions. For example, because both protection violation and byte-ordering exception conditions may be present, and either causes a data storage interrupt, system software would have to look beyond ESR[BO], such as the state of MSR[PR] in SRR1 and the TLB entry page protection bits, to determine if a protection violation also occurred.

4.6 Exceptions

Exceptions are caused directly by instruction execution or by an asynchronous event. In either case, the exception may cause one of several types of interrupts to be invoked.

The following examples are of exceptions caused directly by instruction execution:

- An attempt to execute a reserved-illegal instruction (illegal instruction exception-program interrupt)
- An attempt by an application program to execute a privileged instruction or to access a privileged SPR (privileged instruction exception-program interrupt)
- An attempt to access a nonexistent SPR (illegal-operation program exception-type program interrupt on all accesses to undefined SPRs, regardless of MSR[GS,PR]
- An attempt to access a location that is either unavailable (TLB miss exception-instruction or data TLB error interrupt) or not permitted (access control exception-instruction or data storage interrupt)
- An attempt to access a location with an effective address alignment not supported by the implementation (alignment exception-alignment interrupt)
- Execution of a System Call (**sc**) instruction (system call/hypervisor system call-system call/hypervisor system call interrupt). Whether a system call interrupt occurs or a hypervisor system call interrupt occurs depends on the value of the LEV operand.
- Execution of a trap instruction whose trap condition is met (trap exception-program interrupt)
- Execution of an unimplemented, defined instruction (illegal instruction exception-program interrupt)

Invocation of an interrupt is precise. Power Architecture allows for floating-point enabled exceptions to be imprecise, however e500mc implements them as precise.

4.6.1 Interrupt Ordering and Masking

Multiple exceptions that can each generate an interrupt can exist simultaneously. However, the architecture does not provide for reporting multiple simultaneous interrupts of the same class. therefore, the architecture defines that interrupts must be ordered with one another and provides a way to mask certain persistent interrupt types, as described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

4.7 Interrupt Classification

All interrupts except machine check are grouped by three independent characteristics:

- The set of resources assigned to the interrupt.
  - Standard interrupts use SRR0/SRR1 and the **rfi** instruction. Guest supervisor versions of these interrupts use GSRR0/GSRR1 and the **rfgi** instruction (note that SRR0, SRR1, and **rfi** accesses are mapped to GSRR0, GSRR1, and **rfgi** by the processor when in guest supervisor state).
  - Critical interrupts use CSRR0/CSRR1 and the **rfci** instruction.
  - Debug interrupts use DSRR0/DSRR1, and the **rfdi** instruction.
  - Machine check interrupts use MCSRR0/MCSRR1, and the **rfmci** instruction.
- Whether the interrupt is synchronous or asynchronous. Asynchronous interrupts are caused by events external to instruction execution; synchronous interrupts are caused by instruction execution. Some synchronous interrupts can be imprecise with respect to the instructions that

caused the exception. The *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* describes asynchronous and synchronous interrupts.

4.8 Interrupt Processing

Each interrupt has a vector, the address of the initial instruction that is executed when an interrupt occurs. When an interrupt is taken, the following steps are performed:

1. *x*SRR0 is loaded with an instruction address at which processing resumes when the corresponding return from interrupt instruction executes.
2. The (G)ESR or MCSR may be loaded with exception-specific information. See descriptions of individual descriptions in Section 4.9, "Interrupt Definitions."
3. *x*SRR1 is loaded with a copy of the MSR contents.
4. New MSR values take effect beginning with the first instruction of the interrupt handler. These settings vary somewhat for certain interrupts, as described in Section 4.9, "Interrupt Definitions."

   MSR fields are described in Section 2.7.1, "Machine State Register (MSR)."
5. Instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt type ([G]IVPR[32–47] || (G)IVOR*n*[48–59] || 0b0000)

   The (G)IVOR*n* for the interrupt type is described in Table 4-2. (G)IVPR and (G)IVOR contents are indeterminate upon reset and must be initialized by system software.

At the end of an interrupt handling routine, executing the appropriate return from interrupt instruction causes the MSR to be restored from *x*SRR1 and instruction execution to resume at the address contained in *x*SRR0.

NOTE

On e500mc, any asynchronous interrupt that is taken clears any reservation established from a **lwarx** instruction. However, this behavior is not required by the architecture and software should assume that the reservation is not cleared as subsequent future cores may not clear it.

NOTE

At process switch, due to possible data availability requirements and process interlocks, the operating system should consider executing the following:

- **stwcx.**—Clear outstanding reservations to prevent pairing a **lwarx** in the old process with a **stwcx.** in the new one
- **sync**—Ensure that memory operations of an interrupted process complete with respect to other processors before that process begins executing on another processor
- Return from interrupt instructions—Ensure that instructions in the new process execute in the new context. Normally an operating system must use such an instruction to atomically begin executing in the new process context at the appropriate privilege level.

4.9 Interrupt Definitions

Table 4-2 summarizes each interrupt type, exceptions that may cause that interrupt, the interrupt classification, which (G)ESR bits can be set, which MSR bits can mask the interrupt type, and which IVOR is used to specify the vector address.

Table 4-2. Interrupt Summary by (G)IVOR

IVOR	Interrupt	Exception	Directing State at Exception	(G)ESR[1]	Enabled by	Type[2]	Save and Restore Registers	Page
IVOR0	Critical input		—	—	MSR[CE] or MSR[GS]	A	CSRRs	4-14
IVOR1	Machine check		—	—	MSR[ME] or MSR[GS]	A	MCSRRs	4-14
	Error report		—	—	—	SP	MCSRRs	4-14
IVOR2	Data storage (DSI)	Access or virtualization fault	MSR[GS] = 0 or EPCR[DSIGS] = 0 or TLB[VF] = 1	[ST], [FP,AP], [EPID]	—	SP	SRRs	4-21
		Load reserve or store conditional to write-through required location (W = 1)		[ST]	—	SP		
		Cache locking		[DLK,ILK],[ST]	—	SP		
		Byte ordering		[ST],[FP],BO, [EPID]	—	SP		
GIVOR2	Data storage (DSI)	Access	MSR[GS] = 1 EPCR[DSIGS] = 1	[ST], [FP,AP], [EPID]	—	SP	GSRRs	4-21
		Load reserve or store conditional to write- through required location (W = 1)		[ST]	—	SP		
		Cache locking		[DLK,ILK],[ST]	—	SP		
		Byte ordering		[ST],[FP],BO, [EPID]	—	SP		
IVOR3	Instruction storage (ISI)	Access	MSR[GS] = 0 or EPCR[ISIGS] = 0	—	—	SP	SRRs	4-23
GIVOR3	Instruction storage (ISI)	Access	MSR[GS] = 1 and EPCR[ISIGS] = 1	—	—	SP	GSRRs	4-23
IVOR4	External input [3]		EPCR[EXTGS] = 0	—	MSR[EE] or MSR[GS]	A	SRRs	4-24
GIVOR4	External input [3]		EPCR[EXTGS] = 1	—	MSR[EE] and MSR[GS]	A	GSRRs	4-24
IVOR5	Alignment		—	[ST],[FP,AP], [EPID]	—	SP	SRRs	4-26

Table 4-2. Interrupt Summary by (G)IVOR (continued)

IVOR	Interrupt	Exception	Directing State at Exception	(G)ESR[1]	Enabled by	Type[2]	Save and Restore Registers	Page
IVOR6	Program	Illegal	—	PIL	—	SP	SRRs	4-28
		Privileged		PPR	—	SP		
		Trap		PTR	—	SP		
		Floating-point enabled		FP,[PIE]	MSR[FE0] MSR[FE1]	SP SP* SI SI*		
		Unimplemented opcode		PUO[4]	—	SP		
IVOR7	Floating-point unavailable		—	—	—	SP	SRRs	4-29
IVOR8	System call		MSR[GS] = 0	—	—	SP*	SRRs	4-29
GIVOR8	System call		MSR[GS] = 1	—	—	SP*	GSRRs	4-29
IVOR10	Decrementer		—	—	(MSR[EE] or MSR[GS]) and TCR[DIE]	A	SRRs	4-30
IVOR11	Fixed interval timer		—	—	(MSR[EE] or MSR[GS]) and TCR[FIE]	A	SRRs	4-31
IVOR12	Watchdog		—	—	(MSR[CE] or MSR[GS]) and TCR[WIE]	A	CSRRs	4-32
IVOR13	Data TLB error	Data TLB miss	MSR[GS] = 0 or EPCR[DTLBGS] = 0	[ST],[FP,AP], [EPID]	—	SP	SRRs	4-32
GIVOR13	Data TLB error	Data TLB miss	MSR[GS] = 1 and EPCR[DTLBGS] = 1	[ST],[FP,AP], [EPID]	—	SP	GSRRs	4-32
IVOR14	Instruction TLB error	Instruction TLB miss	MSR[GS] = 0 or EPCR[ITLBGS] = 0	—	—	SP	SRRs	4-33
GIVOR14	Instruction TLB error	Instruction TLB miss	MSR[GS] = 1 and EPCR[ITLBGS] = 1	—	—	SP	GSRRs	4-33

Table 4-2. Interrupt Summary by (G)IVOR (continued)

IVOR	Interrupt	Exception	Directing State at Exception	(G)ESR[1]	Enabled by	Type[2]	Save and Restore Registers	Page
IVOR15	Debug	Trap (synchronous)	—	—	MSR[DE] and DBCR0[IDM] In guest state, if EPCR[DUVD] = 1 and MSR[GS] = 0, debug events (except for unconditional debug events) are not posted in the DBSR. See Section 4.9.1 6.1, "Suppressing Debug Events in Hypervisor Mode."	SP[5]	DSRRs	4-34
		Instruction address compare (synchronous)						
		Data address compare (synchronous)						
		Instruction complete						
		Branch taken						
		Return from interrupt						
		Return from critical interrupt						
		Interrupt taken						
		Critical interrupt taken						
		Unconditional debug event				A		
IVOR35	Performance monitor		—	—	MSR[EE] or MSR[GS]	A	SRRs	4-36
IVOR36	Processor doorbell		—	—	MSR[EE] or MSR[GS]	A	SRRs	4-38
IVOR37	Processor doorbell critical		—	—	MSR[CE] or MSR[GS]	A	CSRRs	4-38
IVOR38	Guest processor doorbell		—	—	MSR[EE] and MSR[GS]	A	GSRRs	4-38
IVOR39	Guest processor doorbell critical		—	—	MSR[CE] and MSR[GS]	A	CSRRs	4-39
	Guest processor doorbell machine check		—	—	MSR[ME] and MSR[GS]	A	CSRRs	4-39
IVOR40	Hypervisor system call		—	—	—	SP*	SRRs	4-29
IVOR41	Hypervisor privilege		—	—	—	SP	SRRs	4-40

[1] In general, when an interrupt affects an (G)ESR as indicated in the table, it also causes all other (G)ESR bits to be cleared. Special rules may apply for implementation-specific (G)ESR bits.

Legend:
xxx (no brackets) means (G)ESR[*xxx*] is set.
[xxx] means (G)ESR[*xxx*] could be set.
[xxx,yyy] means either (G)ESR[*xxx*] or (G)ESR[*yyy*] may be set, but not both.
{xxx,yyy} means either (G)ESR[*xxx*] or (G)ESR[*yyy*] and possibly both may be set.

[2] Interrupt types:
SP = synchronous and precise
SI = synchronous and imprecise
A = asynchronous
* = post completion interrupt. *x*SRR0 registers point after the instruction causing the exception.

[3] Section 4.9.6.3, "External Proxy," describes how the e500mc interacts with a programmable interrupt controller (PIC) defined by the integrated device.

[4] PUO does not occur on e500mc.

[5] This debug interrupt may be made pending if MSR[DE] = 0 at the time of the exception.

4.9.1 Partially Executed Instructions

In general, the architecture permits load and store instructions to be partially executed, interrupted, and then restarted from the beginning upon return from the interrupt. To guarantee that a particular load or store instruction completes without being interrupted and restarted, software must mark the memory as guarded and use an elementary (nonmultiple) load or store aligned on an operand-sized boundary.

4.9.1.1 Restarting Instructions After Partial Execution

To guarantee that load and store instructions can, in general, be restarted and completed correctly without software intervention, the following rules apply when an execution is partially executed and then interrupted:

- For an elementary load, no part of a target register **r**D is altered.
- For update forms of load or store, the update register, **r**A, is not altered.

The following effects are permissible when certain instructions are partially executed and then restarted:

- For any store, bytes at the target location may have been altered (if write access to that page in which bytes were altered is permitted by the access control mechanism). In addition, for store conditional instructions, CR0 has been set to an undefined value, and it is undefined whether the reservation has been cleared.
- For any load, bytes at the addressed location may have been accessed (if read access to that page in which bytes were accessed is permitted by the access control mechanism).
- For load multiple some registers in the range to be loaded may have been altered. It is a programming error to include the addressing registers **r**A, and possibly **r**B, in the range to be loaded, and thus the rules for partial execution do not protect these registers against overwriting.

NOTE

In no case is access control violated.

For e500mc the following rules apply:

1. The only instructions that are “interruptible” and for which the changes to registers (on loads) or the changes to memory (on stores) are visible are load and store multiple instructions.
2. If a load or store multiple gets interrupted, some number of words may already be processed and the GPRs or memory reflects that. When the interrupt returns the multiple instruction is restarted and performs all the previous actions as well as finishing the actions (assuming its not interrupted again).
3. Any aligned, elementary (byte, halfword, word, doubleword) load or store is not interruptible and always performs atomically (meaning that the bytes are all read or written at once and no other agent can modify the bytes during the time the bytes are sampled or written.
4. Store conditionals are always atomic. The CR value is always set based on whether the store was actually performed or not.
5. Misaligned loads and stores are not guaranteed to be atomic. Therefore another agent may modify the data in between the sampling of all the bytes. However, it's unlikely you'd even see this happen if the data is cacheable.
6. A load that is not guarded may speculatively access memory and later be canceled if it turns out the load was cancelled (because we took an interrupt or had a mispredicted branch). In this case, the load data is never made visible to the architected state (i.e. the GPR that is the target of the load is not changed).

NOTE

In the same sense, a misaligned load is also possibly sample data non-atomically.
In the case of misaligned guarded loads, the guarded load may sample data and then be cancelled due to an interrupt, but does not change any visible architected state.

4.9.1.2 Interruptions After Partial Execution

As previously stated, elementary, aligned, guarded loads and stores are the only access instructions guaranteed not to 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:

- Any load or store (except elementary, aligned, and guarded):
  - Any asynchronous interrupt
  - Machine check
  - Decrementer
  - Fixed-interval timer
  - Watchdog timer
  - Debug (unconditional debug event)
- Misaligned elementary load or store, or any multiple:

  All of the above listed under item •, plus the following:

— Alignment
— Data storage (if the access crosses a page boundary and protections on the 2 pages differ)
— Data TLB (if the access crosses a page boundary and one of the pages is not in the TLB)
— Debug (data address compare)

4.9.2 Critical Input Interrupt—IVOR0

A critical input interrupt occurs when no higher priority interrupt exists, a critical input exception is presented to the interrupt mechanism, and MSR[CE] = 1 or MSR[GS] = 1. The reference manual for the integrated device describes how this exception is signaled (typically the signal is described as *cint*.).

As defined by the architecture, CSRR0, CSRR1, and MSR are updated as shown in this table.

Table 4-3. Critical Input Interrupt Register Settings

Register	Setting
CSRR0	Set to the effective address of the next instruction to be executed
CSRR1	Set to the MSR contents at the time of the interrupt
MSR	• ME and DE are unchanged. • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR0[48–59] || 0b0000.

NOTE

To avoid redundant critical input interrupts, software must take any actions required by the implementation to clear any critical input exception status before reenabling MSR[CE] or setting MSR[GS].

4.9.3 Machine Check Interrupt—IVOR1

The Machine Check Interrupt consists of three different, but related, types of exception conditions that all use the same interrupt vector and same interrupt registers. The three different interrupts are:

- Asynchronous machine check exceptions which are the result of error conditions directly detected by the processor or as a result of the assertion of the machine check signal pin (typically described in the integrated device reference manual as the *mcp* signal) as described by Section 4.9.3.4, “Asynchronous Machine Check Exceptions.”
- Synchronous error report exceptions which are the result of an instruction encountering an error condition, but execution cannot continue without propagating data derived from the error condition as described in Section 4.9.3.3, “Machine Check Error Report Synchronous Exceptions.”
- Non-maskable (NMI) interrupts which are non-maskable, non-recoverable interrupts that are signaled from the SoC as described by Section 4.9.3.2, “NMI Exceptions.”

For all of these interrupts, the following occur:

- MCSRR0 and MCSRR1 save the return address and MSR.

- An address related to the machine check may be stored in MCAR (and MCARU), according to Table 4-4.
- The machine check syndrome register, MCSR, is used to log information about the error condition. The MCSR is described in Section 2.9.9, "Machine Check Syndrome Register (MCSR)."
- At the end of the machine check interrupt software handler, a Return from Machine Check Interrupt (**rfmci**) may be used to return to the state saved in MCSRR0 and MCSRR1.

Machine check exceptions are typically caused by a hardware failure or by software performing actions for which the hardware has not been designed to handle, or cannot provide a suitable result. They may be caused indirectly by execution of an instruction, but may not be recognized or reported until long after the processor has executed that instruction.

4.9.3.1 General Machine Check, Error Report, and NMI Mechanism

Asynchronous machine check, error report machine check, and NMI exceptions are independent of each other, even though they share the same interrupt vector.

4.9.3.1.1 Error Detection and Reporting Overview

The general flow of error detection and reporting occurs as follows:

- When the processor detects an error directly (that is, the error occurs within the processor) or the machine check signal pin (*mcp*) is asserted, the error is posted to the MCSR by setting an error status bit corresponding to the error that was detected. If the error bit set in the MCSR is one of the asynchronous machine check error conditions, an asynchronous machine check occurs when MSR[ME] = 1 or MSR[GS] = 1. Note that an asynchronous machine check interrupt always occurs when the asynchronous machine check interrupt is enabled and any of the asynchronous error bits in the MCSR are non-zero.
- If an instruction is a consumer of data associated with the error, the instruction has an error report exception associated with the instruction ensuring that if the instruction reaches the point of completion, the instruction takes an error report machine check interrupt to prevent the erroneous data from propagating.
- It is possible a single error within the processor sets both an asynchronous machine check error condition in the MCSR, and associates an error report with the instruction that consumed data associated with the error. The asynchronous error bit is always set, and if this triggers an asynchronous machine check interrupt before the instruction that has the error report exception completes, the asynchronous machine check interrupt flushes the instruction with the error report, and the error report does not occur. Likewise, if the instruction with the error report exception attempts to complete before the asynchronous error bit is set in MCSR, the error report machine check interrupt is taken. In this case, the processor still sets the MCSR asynchronous error bit, probably well before software has read the MCSR. When software reads the MCSR, it appears that both an asynchronous machine check exception and a synchronous error report occurred, because the error report has caused the error report bits to be set, and the processor also has set an asynchronous machine check error bit. This can easily happen if the error occurs when MSR[ME] = 0 and MSR[GS] = 0 because the asynchronous machine check interrupt is not enabled.

- It is also possible that an error report machine check interrupt occurs without an associated asynchronous machine check error bit being set in the MCSR. This can occur when the processor is the consumer of some data for which the error was detected by some agent other than the processor. For example, an error in DRAM may occur and if the processor executed a load instruction which accessed that DRAM where the error occurred, the load instruction would take an error report machine check interrupt if it attempted to complete execution.
- A non-maskable interrupt (NMI) occurs when the integrated device asserts the NMI signal to the e500mc. The MCSR[NMI] bit is set when the interrupt occurs. The NMI signal is non-maskable and occurs regardless of the state of MSR[ME] or MSR[GS].

NOTE

The taking of an asynchronous machine check interrupt always occurs when any of the asynchronous machine check error bits is not zero and the asynchronous machine check interrupt is enabled (MSR[ME] = 1 or MSR[GS] = 1). The condition persists until software clears the asynchronous machine check error bits in MCSR.
To avoid multiple asynchronous machine check interrupts, software should always read the contents of the MCSR within the asynchronous machine check interrupt handler and clear any set bits in the MCSR prior to re-enabling machine check interrupts by setting MSR[ME] or MSR[GS]. Note that the processor may set asynchronous machine check error bits in MCSR at any time as errors are detected, including when the processor is in the asynchronous machine check interrupt handler and MSR[ME] = 0.

An asynchronous machine check, error report, or NMI interrupt occurs when no higher priority interrupt exists and an asynchronous machine check, error report, or NMI exception is presented to the interrupt mechanism.

The following general rules apply:

- The instruction whose address is recorded in MCSRR0 has not completed, but may have attempted to execute.
- No instruction after the one whose address is recorded in MCSRR0 has completed execution.
- Instructions in the architected instruction stream prior to this instruction have all completed successfully.

4.9.3.1.2 Machine Check Interrupt Settings

When a machine check interrupt is taken, registers are updated as shown in this table.

Table 4-4. Machine Check Interrupt Settings

Register	Setting
MCSRR0	The core sets this to an EA of an instruction executing or about to execute when the exception occurred.
MCSRR1	Set to the contents of the MSR at the time of the exception.
MSR	• RI is cleared. • All other defined MSR bits are cleared.

Table 4-4. Machine Check Interrupt Settings (continued)

Register	Setting
MCAR (MCARU)	MCAR is updated with the address of the data associated with the machine check. See Section 2.9.8, "Machine Check Address Register (MCAR/MCARU)."
MCSR	Set according to the machine check condition. See Table 2-8.

Instruction execution resumes at address IVPR[32–47] || IVOR1[48–59] || 0b0000.

NOTE

For implementations on which a machine check interrupt is caused by referring to an invalid physical address, executing **dcbz**, **dcbzl**, **dcbzep**, **dcbzlep**, **dcba**, or **dcbal** can ultimately cause a machine check interrupt long after the instruction executed by establishing a data cache block associated with an invalid physical address. The interrupt can occur later on an attempt to write that block to main memory, for example, as the result of executing an instruction that causes a cache miss for which the block is the target for replacement or as the result of executing **dcbst** or **dcbf**.

4.9.3.1.3 Machine Check Exception Sources

The e500mc machine check exception sources are specified in this table.

Table 4-5. Machine Check Exception Sources

Source	Additional Enable Bits[1]
Machine check input signal asserted. Set immediately on recognition of assertion of the *mcp* input. This input comes from the SoC and is a level sensitive signal. This usually occurs as the result of an error detected by the SoC.	HID0[EMCP]
Instruction cache tag or data array parity error	L1CSR1[ICPE] and L1CSR1[ICE]
Data cache data parity or tag parity error due to a load or store	L1CSR0[CECE] and L1CSR0[CE]
L2 MMU multi-way hit. Multi-way hit in the L2 MMU. Indicates that a lookup in the L2 MMU yielded multiple hits. This signifies overlapping TLB entries. The overlap may be between multiple ways in the 4K array, between multiple entries in the CAM, or between entries in the 4K and CAM. These errors are detected when the L2 MMU is accessed. It is possible for an overlap condition to exist for some time before it is detected. As long as translations are satisfied by the L1 MMU, no L2 MMU lookup is required, and the overlap condition is not detected. If an L2 MMU simultaneous hit occurs during the execution of **dcba**, **dcbal**, **dcbt**, **dcbtep**, **dcbtst**, **dcbtstep**, or **icbt**, no error report machine check occurs on the instruction and no access off the core is performed.	HID0[EN_L2MMU_MHD]
Nonmaskable interrupt.	None
Simultaneous **tlbsync** operations detected. The system should never have two outstanding **tlbsync** operations on CoreNet.	None
L2 cache error	L2CSR0[L2E] and L2ERRDIS [2]

[1] "Additional Enable Bits" indicates any other state that, if not enabled, inhibits the recognition this particular error condition.
[2] For a description of L2ERRDIS, see Section 2.15.4.1, "L2 Cache Error Disable Register (L2ERRDIS)."

4.9.3.2 NMI Exceptions

Non-maskable interrupt exceptions cause an interrupt on the machine check vector. A non-maskable interrupt occurs when the integrated device asserts the *nmi* signal to the e500mc. The *nmi* signal is non-maskable and occurs regardless of the state of MSR[ME] or MSR[GS]. Software should clear the NMI bit in MCSR after the NMI interrupt has been taken before setting MSR[ME] or MSR[GS].

NMI interrupts are by definition non-recoverable since the interrupt occurs asynchronously and the interrupt cannot be masked by software. Unrecoverability can occur if the NMI occurs while the processor is in the early part of an asynchronous machine check, error report machine check, or another NMI interrupt handler and the return state in MCSRR0 and MCSRR1 have not yet been saved by software. It is possible for software to use MSR[RI] to determine whether software believes it is safe to return, but the system designer must allow for the case for which MCSRR0 and MCSRR1 have not been saved.

4.9.3.3 Machine Check Error Report Synchronous Exceptions

Error report machine checks are intended to limit the propagation of bad data. For example, if a cache parity error is detected on a load, the load instruction is not allowed to *complete*, a synchronous error report machine check is generated, and the MCSRR0 holds the address of the load instruction with which the parity error is associated. (For a discussion of instruction completion, see Chapter 10, "Execution Timing.")

Preventing the load instruction from completing prevents the bad data from reaching the GPRs and prevents any subsequent instructions dependent on that data from executing. Error reports do not indicate the source of the problem (such as the cache parity error in the current example); the source is indicated by an asynchronous machine check. When an error report type of machine check occurs, the MCSR indicates the operation that incurred the error as shown in this table.

Table 4-6. Error Report Definitions

Error Report	Definition
Instruction fetch error report (MCSR[IF])	An error occurred while attempting to fetch the instruction corresponding to the address contained in MCSRR0.
Load instruction error report (MCSR[LD])	An error occurred while attempting to execute the load instruction corresponding to the address contained in MCSRR0.

Table 4-6. Error Report Definitions (continued)

Error Report	Definition
Guarded load instruction error report (MCSR[LDG])	If LD is set and the load was a guarded load (that is, has the guarded storage attribute), this bit may be set. Note that some implementations may have specific conditions that govern when this bit is set.
Store instruction error report (MCSR[ST])	An error occurred while attempting to perform address translation on the instruction corresponding to the address contained in MCSRR0. Since stores may complete with respect to the processor pipeline before their effects are seen in all memory subsystem areas, only translation errors are reported as error reports with stores. Note that some instructions which are considered load instructions with respect to permission checking and debug events are reported as store error reports (MCSR[ST] is set). See Section 2.9.9, "Machine Check Syndrome Register (MCSR)" for which instructions set MCSR[LD] or MCSR[ST].

Table 4-7 describes which error sources generate which error report status bits in the MCSR.

Note that there is no MCSR error status bit for CoreNet data errors. If a CoreNet data error occurs on a load or instruction fetch and the instruction reaches the bottom of the completion buffer, an error report occurs. But, because there is no MCSR error status bit for data errors, the core does not generate an asynchronous machine check. The device that detects the error is expected to report it. For example, assume that the core attempts to perform a load from a PCI device that encounters an error. The PCI device would signal a "PCI Master Abort" and would signal the error to the programmable interrupt controller (PIC).

The core's memory transaction should be completed with a data error so that the core is not hung awaiting the transaction. Eventually, the PIC should interrupt the core (the PIC should be programmed to direct such an error to take a machine check interrupt).

Error reports are intended to be a mechanism to stop the propagation of bad data; the asynchronous machine check is intended to allow software to attempt to recover from errors gracefully.

In a multicore system, the PIC is likely to steer all PCI error interrupts to one processor. For the PCI Master Abort example, assume that Processor B performs a load that gets a PCI Master Abort, and the PIC steers the PCI's error signal to Processor A's machine check input signal. Here, the error report in Processor B prevents the propagation of bad data; Processor A gets the task of attempting a graceful recovery. Some interprocessor communication is likely necessary.

Table 4-7. Synchronous Machine Check Error Reports

Synchronous Machine Check Source	Error Type	MCSR Update[1]	Precise[2]
Instruction fetch	Instruction cache data array parity error	IF	Within fetch group[3]
	Instruction cache tag array parity error		
	L2MMU multi-way hit		
	CoreNet bad data		
	L2 cache error		

Table 4-7. Synchronous Machine Check Error Reports (continued)

Synchronous Machine Check Source	Error Type	MCSR Update[1]	Precise[2]
Load (or touch) instruction	Data cache tag parity error	LD, [LDG][4]	Yes
	Data cache data array parity error		
	L2MMU multi-way hit		
	L2 cache tag parity or data error (uncorrectable ECC error)[5]		
	CoreNet Bad Data		
Store or cache operation instruction	L2MMU multi-way hit	ST	—

1 The MCSR update column indicates which MCSR bits are updated when the machine check interrupt is taken.

2 The Precise column either indicates 'yes' or 'within fetch group'. If "yes." the error type causes a machine check in which the MCSRR0 points to the instruction that encountered the error, provided that MSR[ME] or MSR[GS] were set when the instruction was executed.

3 Error report machine check interrupts caused by instruction fetches (denoted by MCSR[IF]) are associated with all instructions within a given fetch group. If any instruction within the fetch group encountered an error of any type, then all instructions within the fetch group are marked with an instruction fetch error report exception. therefore, if the error report exception later causes a machine check interrupt, MCSRR0 points to the oldest instruction from that fetch group.

4 LDG is set if the load was a guarded load (WIMGE=xxx1x).

5 If L2 error detection is not enabled, an error report exception is not reported and the corrupted instruction may be executed and may modify architected state.

An error report occurs only if the instruction that encountered the error reaches the bottom of the completion buffer (that is, it becomes the oldest instruction currently in execution) and the instruction would have completed otherwise. If the instruction is flushed (possibly due to a mispredicted branch or asynchronous interrupt, including an asynchronous machine check) before reaching the bottom of the completion buffer, the error report does not occur.

4.9.3.4 Asynchronous Machine Check Exceptions

An asynchronous machine check occurs only when MSR[ME] = 1 or MSR[GS] = 1 and an MCSR asynchronous error bit is set. Because MSR[ME] and MSR[GS] are cleared whenever a machine check interrupt occurs, a synchronous error report interrupt may clear MSR[ME] and MSR[GS] before the MCSR error bit is posted. If the error report handler clears the MCSR error bit before setting MSR[ME] or MSR[GS], no asynchronous machine check interrupt occurs.

This table describes asynchronous machine check and NMI exceptions.

Table 4-8. Asynchronous Machine Check and NMI Exceptions

Error Source	Error Type	Transaction Source	MCSR Update[1]		MCAR Update[2]
External	Machine check input (*mcp*) pin[3]	n/a	MCP		—
	NMI Pin	n/a	NMI		—
Instruction cache	Data array parity error	Instruction fetch	MAV	ICPERR	EA
	Tag array parity error				RA

Table 4-8. Asynchronous Machine Check and NMI Exceptions (continued)

Error Source	Error Type	Transaction Source	MCSR Update[1]		MCAR Update[2]
Data cache	Tag parity error	load, touch, stores, cache operations, or snoops	MAV	DCERR	RA
	Data array parity error	load, castout, or snoop			
L2 cache	All types[4]	All types	BSL2_ERR		—
L2 MMU	Multi-way hit	**tlbsx**, instruction fetch, load, touch, store, cache op (all types)	MAV	L2MMU_MHIT	EA[5]
	Multiple simultaneous **tlbsync** operations detected	TLBSYNC snoop	TLBSYNC		none

1 The MCSR update column indicates which MCSR bits are updated when the exception is logged.

2 The MCAR update column indicates whether the error type provides either a real or effective address (RA or EA), or no address which is associative with the error.

3 The machine check input pin is used by the SoC to indicate all types of machine check type error which are detected by the SoC. Software must query error logging information within the SoC to determine the specific error condition and source.

4 The L2 cache has a separate set of error reporting and capture registers.

5 The lower 12 bits of the EA are cleared.

4.9.4 Data Storage Interrupt (DSI)—IVOR2/GIVOR2

A DSI occurs when no higher priority interrupt exists and a data storage exception is presented to the interrupt mechanism. The interrupt is directed to the hypervisor unless the following conditions exist: determined as follows:

- The exception is not a virtualization fault (TLB[VF] = 0).
- The state in which the exception occurred is the guest state (MSR[GS] = 1).
- The interrupt is programmed to be directed to the guest state (EPCR[DSIGS] = 1).

If all the above conditions are met, the DSI is directed to the guest supervisor state.

This table (taken from Table 4-2) summarizes exception conditions and behavior for the data storage and guest data storage interrupts.

Table 4-9. Data Storage interrupt

IVOR	Interrupt	Exception	Directing State at Exception	(G)ESR[1]	Save/Restore Registers
IVOR2	Data storage (DSI)	Access or virtualization fault	MSR[GS] = 0 or EPCR[DSIGS] = 0 or TLB[VF] = 1	[ST], [FP,AP], [EPID]	SRRs
		Load reserve or store conditional to write-through required location (W = 1)		[ST]	
		Cache locking		[DLK,ILK],[ST]	
		Byte ordering		[ST],[FP],BO, [EPID]	

Table 4-9. Data Storage interrupt (continued)

IVOR	Interrupt	Exception	Directing State at Exception	(G)ESR[1]	Save/Restore Registers
GIVOR2	Guest data storage (DSI)	Access	MSR[GS] = 1 ¶ EPCR[DSIGS] = 1	[ST], [FP,AP], [EPID]	GSRRs
		Load reserve or store conditional to write- through required location (W = 1)		[ST]	
		Cache locking		[DLK,ILK],[ST]	
		Byte ordering		[ST],[FP],BO, [EPID]	

[1] In general, when an interrupt affects an (G)ESR as indicated in the table, it also causes all other (G)ESR bits to be cleared. Special rules may apply for implementation-specific (G)ESR bits.

Legend:
xxx (no brackets) means (G)ESR[*xxx*] is set.
[xxx] means (G)ESR[*xxx*] could be set.
[xxx,yyy] means either (G)ESR[*xxx*] or (G)ESR[*yyy*] may be set, but not both.
{xxx,yyy} means either (G)ESR[*xxx*] or (G)ESR[*yyy*] and possibly both may be set.

This table describes exceptions as defined by the architecture, noting any e500mc-specific behavior.

Table 4-10. Data Storage Interrupt Exception Conditions

Exception	Cause
Read access control exception	Occurs when either of the following conditions exists: • In user mode (MSR[PR] = 1), a load or load-class cache management instruction attempts to access a memory location that is not user-mode read enabled (page access control bit UR = 0). • In supervisor mode (MSR[PR] = 0), a load or load-class cache management instruction attempts to access a location that is not supervisor-mode read enabled (page access control bit SR = 0).
Virtualization fault	Loads and stores translated by TLB entries with the TLB[VF] = 1 always take a data storage interrupt directed to hypervisor state.
Write access control exception	Occurs when either of the following conditions exists: • In user mode (MSR[PR] = 1), a store or store-class cache management instruction attempts to access a location that is not user-mode write enabled (page access control bit UW = 0). • In supervisor mode (MSR[PR] = 0), a store or store-class cache management instruction attempts to access a location that is not supervisor-mode write enabled (page access control bit SW = 0).
Byte-ordering exception	Data cannot be accessed in the byte order specified by the page's endian attribute. **Note:** This exception is provided to assist implementations that cannot support dynamically switching byte ordering between consecutive accesses, the byte order for a class of accesses, or misaligned accesses using a specific byte order. On the e500mc, load/store accesses that cross a page boundary such that endianness changes cause a byte-ordering exception.
Cache locking exception	The locked state of one or more cache lines may potentially be altered. Occurs with the execution of **icbtls**, **icblc**, **dcbtls**, **dcbtstls**, or **dcblc** when (MSR[PR] = 1) and (MSR[UCLE] = 0). ESR is set as follows: • For **icbtls** and **icblc**, ESR[ILK] is set. • For **dcbtls**, **dcbtstls**, or **dcblc**, ESR[DLK] is set. The architecture refers to this as a cache-locking exception.
Storage synchronization exception	Occurs when a **lwarx** or **stwcx.** attempts to access a location marked write-through required. Note that if the EA associated with a store conditional instruction would have caused a write access control exception, were the instruction not a store conditional, even if the store would not be performed (because the reservation is not held), a DSI write access control exception occurs. See "Atomic Update Primitives Using **lwarx** and **stwcx.**," in the "Instruction Model" chapter of the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

Regardless of the EA, **icbt**, **dcbt**, **dcbtst**, **dcba** and **dcbal** cannot cause a data storage interrupt.

NOTE

icbi, **icbt**, **icblc**, and **icbtls** are treated as loads from the addressed byte with respect to translation and protection. Both use MSR[DS], not MSR[IS], to determine translation for their operands. Instruction storage and TLB error interrupts are associated with instruction fetching and not execution. Data storage and TLB error interrupts are associated with execution of instruction cache management instructions.

When the interrupt occurs, the processor suppresses execution of the instruction that caused it. Registers are updated as follows:

Table 4-11. Data Storage Interrupt Register Settings

Register	Setting
(G)SRR0	Set to the EA of the instruction causing the interrupt
(G)SRR1	Set to the MSR contents at the time of the interrupt
(G)ESR	ST Set if the instruction causing the interrupt is a store or store-class cache management instruction DLK Set when a DSI occurs because **dcbtls**, **dcbtstls**, or **dcblc** is executed in user mode and MSR[UCLE] = 0. ILK Set when a DSI occurs because **icbtls** or **icblc** is executed in user mode and MSR[UCLE] = 0. BO Set if the instruction caused a byte-ordering exception. All other defined ESR bits are cleared.
MSR	• ME, CE, and DE are unchanged. • GS, UCLE, and PMM are cleared if the interrupt is directed to the hypervisor state. • UCLE and PMM are cleared If the interrupt is directed to the guest state and the associated bits of MSRP are 0. • RI is not cleared. • All other defined MSR bits are cleared.
(G)DEAR	Set to the EA of a byte that lies both within the range of bytes being accessed by the access or cache management instruction and within the page whose access caused the exception,

Instruction execution resumes at address (G)IVPR[32–47] || (G)IVOR2[48–59] || 0b0000.

4.9.5 Instruction Storage Interrupt (ISI)—IVOR3/GIVOR3

An ISI occurs when no higher priority interrupt exists and an instruction storage interrupt is presented to the interrupt mechanism.

The interrupt is directed to the hypervisor unless the following conditions exist:

- The state in which the exception occurred is the guest state (MSR[GS] = 1).
- The interrupt is programmed to be directed to the guest state (EPCR[ISIGS] = 1).

If all the above conditions are met, the ISI is directed to the guest supervisor state.

This table describes exception conditions.

Table 4-12. Instruction Storage Interrupt Exception Conditions

Exception	Cause
Execute access control exception	In user mode, an instruction fetch attempts to access memory that is not user mode execute enabled (page access control bit UX = 0). In supervisor mode, an instruction fetch attempts to access a memory that is not supervisor mode execute enabled (page access control bit SX = 0).

When an ISI occurs, the processor suppresses execution of the instruction causing the interrupt.

Registers are updated as shown in this table.

Table 4-13. Instruction Storage Interrupt Register Settings

Register	Setting
(G)SRR0	Set to the EA of the instruction causing the interrupt
(G)SRR1	Set to the MSR contents at the time of the interrupt
MSR	• ME, CE, and DE are unchanged. • GS, UCLE, and PMM are cleared if the interrupt is directed to the hypervisor state. • UCLE and PMM are cleared If the interrupt is directed to the guest state and the associated bits of MSRP are 0. • RI is not cleared. • All other defined MSR bits are cleared.
(G)ESR	All defined ESR bits are cleared.

Instruction execution resumes at address (G)IVPR[32–47] || (G)IVOR3[48–59] || 0b0000.

4.9.6 External Input Interrupt—IVOR4/GIVOR4

An external input interrupt occurs when no higher priority interrupt exists, an external input interrupt (typically described in the integrated reference manual as the *int* signal) is presented to the interrupt mechanism, and MSR[EE] = 1. The interrupt is directed to the hypervisor unless the following conditions exist: determined as follows:

- The state in which the exception occurred is the guest state (MSR[GS] = 1).
- The interrupt is programmed to be directed to the guest state (EPCR[EXTGS] = 1).

If all the above conditions are met, the external input interrupt is directed to the guest supervisor state. The interrupt is enabled by the MSR[EE], MSR[GS], and EPCR[EXTGS] bits as follows:

- If EPCR[EXTGS] = 0, the interrupt is enabled if MSR[EE] = 1 or MSR[GS] = 1.
- If EPCR[EXTGS] = 1, the interrupt is enabled if MSR[EE] = 1 and MSR[GS] = 1.

4.9.6.1 Receiving External Input Interrupts

In an integrated device, external interrupts are typically signaled to the core from a programmable interrupt controller (PIC), which manages and prioritizes interrupt requests from integrated peripheral devices such that the highest priority request is guaranteed to be presented to the core as quickly as possible.

The e500mc provides two methods of receiving external input interrupts, which is controlled through a register field in the PIC:

- In one method, the legacy method, the core takes an external input interrupt when the *int* signal from the PIC is asserted and the external input interrupt is enabled. The input is level sensitive and if *int* is deasserted before the interrupt is enabled, no interrupt occurs. If the interrupt is enabled and occurs, software reads the memory-mapped Interrupt Acknowledge (IACK) register which contains the specific vector of the interrupt. This causes the PIC to deassert *int* until another interrupt is requested and management of the interrupt is software's responsibility (it is *in-service*) until it performs an associated End of Interrupt (EOI) memory-mapped register write to the PIC.
- In the alternate method known as External Proxy, a signaling protocol occurs between the core and the PIC. Instead of just signaling *int*, the PIC also provides the specific vector for the interrupt. When the interrupt is enabled and the PIC is asserting *int*, the interrupt occurs and the core communicates to the PIC that the interrupt has been taken and provides the vector from the PIC in the (G)EPR register which software then can read. As part of the communication with the PIC, the PIC puts the specific interrupt *in-service* as if software had read the IACK register in the legacy method. This method is further described in Section 4.9.6.3, "External Proxy."

4.9.6.2 External Input Interrupt Register Settings

Registers are updated as shown in this table.

Table 4-14. External Input Interrupt Register Settings

Register	Setting
(G)SRR0	Set to the effective address of the next instruction to be executed
(G)SRR1	Set to the MSR contents at the time of the interrupt
MSR	• ME, CE, and DE are unchanged. • GS, UCLE, and PMM are cleared if the interrupt is directed to the hypervisor state. • UCLE and PMM are cleared If the interrupt is directed to the guest state and the associated bits of MSRP are 0. • RI is not cleared. • All other defined MSR bits are cleared.
(G)EPR	If external proxy is used, (G)EPR holds the vector offset that identifies the source that generated the interrupt triggered from the PIC. For external interrupts not generated using interrupt proxy, (G)EPR is updated to all zeros.

Instruction execution resumes at address (G)IVPR[32–47] || (G)IVOR4[48–59] || 0b0000.

NOTE

To avoid redundant external input interrupts, software must take any actions required to clear any external input exception status before reenabling MSR[EE].

4.9.6.3 External Proxy

The external proxy facility defines an interface for using a core-to-interrupt controller hardware interface for acknowledging external interrupts from a programmable interrupt controller (PIC) implemented as part

of the integrated device. This functionality is enabled through a register field defined by the PIC and documented in the reference manual for the integrated device.

Using this interface reduces the latency required to read and acknowledge the interrupt that normally requires a cache-inhibited guarded load to the memory controller.

In previous integrated devices, when the core received a signal from the PIC indicating that the external interrupt was necessary to handle a condition typically presented by an integrated peripheral device, the interrupt handler responded by reading a memory-mapped register (interrupt acknowledge, or IACK) defined by the Open PIC standard. In addition to providing an additional vector offset specific to the peripheral device, this read negated the internal signal and changed the status of the interrupt request from *pending* to *in-service* in which state it would remain until the completion of the interrupt handling.

The external proxy eliminates the need to read the IACK register by presenting the vector to the external proxy register (EPR), or guest external proxy register (GEPR), described in Section 2.9.5, "(Guest) External Proxy Register (EPR/GEPR)."

Instead of just signaling *int*, the PIC also provides the specific vector for the interrupt. When the interrupt is enabled and the PIC is asserting *int*, the interrupt occurs and the core communicates to the PIC that the interrupt has been taken and provides the vector from the PIC in the (G)EPR register which software then can read. As part of the communication with the PIC, the PIC puts the specific interrupt *in-service* as if software had read the IACK register in the legacy method. The PIC always asserts the highest priority pending interrupt to the core and the interrupt that is put in-service is determined by when the core takes the interrupt based on the appropriate enabling conditions.From a system software perspective, the core does not acknowledge the interrupt until the external input interrupt is taken.

Software in the external input interrupt handler would then read (G)EPR to determine the vector for the interrupt. The value of the vector in (G)EPR does not change until the next external input interrupt occurs and therefore software must read (G)EPR before re-enabling the interrupt.

When using external proxy (and even with the legacy method), software must ensure that end-of-interrupt (EOI) processing is synchronized with taking of external input interrupts such that the EOI indicator is received so that the interrupt controller can properly pair it with the source. For example, writing the EOI register for the PIC would require that the following sequence occur:

```
block interrupts;     // turn EE off for external interrupts
write EOI register;   // signal end of interrupt
read EOI register;    // ensure write has completed
unblock interrupts;   // allow interrupts
```

4.9.7 Alignment Interrupt—IVOR5

An alignment interrupt occurs when no higher priority exception exists and an alignment exception is presented to the interrupt mechanism. On the e500mc, these exceptions are as follows:

- The following accesses are not word aligned:
  - Floating-point loads and stores
  - Load multiple or store multiple instruction (**lmw** and **stmw**).
  - A **lwarx** or **stwcx**. instruction.

NOTE

The architecture does not support use of a misaligned EA by load and reserve or store conditional instructions. If a misaligned EA is specified, the alignment interrupt handler must treat the instruction as a programming error and not attempt to emulate the instruction.

- A **dcbz, dcbzep**, **dcbzepl**, or **dcbzl** is attempted to a page marked write-through or cache-inhibited.

For other accesses, the e500mc performs misaligned accesses in hardware within a single cycle if the misaligned operand lies within a doubleword boundary. Accesses that cross a doubleword boundary degrade performance. Although many misaligned memory accesses are supported in hardware, their frequent use is discouraged because they can compromise overall performance. Only one outstanding misalignment at a time is supported, which means it is nonpipelined. A misaligned access that crosses a page boundary completely restarts if the second portion of the access causes a TLB miss or a DSI after the associated interrupt has been serviced and the TLB miss or DSI handler has returned to re-execute the instruction. This can cause the first access to be repeated.

When an alignment interrupt occurs, the processor suppresses execution of the instruction causing the alignment interrupt. Registers are updated as shown in Table 4-15.

Table 4-15. Alignment Interrupt Register Settings

Register	Setting
SRR0	Set to the EA of the instruction causing the alignment interrupt
SRR1	Set to the MSR contents at the time of the interrupt
MSR	• ME, CE, and DE are unchanged. • RI is not cleared. • All other defined MSR bits are cleared.
DEAR	Set to the EA of a byte in the range of bytes being accessed and on the page whose access caused the exception
ESR	The following bits may be set: ST Set only if the instruction causing the exception is a store and is cleared for a load All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR5[48–59] || 0b0000.

4.9.8 Program Interrupt—IVOR6

A program interrupt occurs when no higher priority exception exists and a program interrupt is presented to the interrupt mechanism. This table lists program interrupt exceptions.

Table 4-16. Program Interrupt Exception Conditions

Exception	Cause	ESR Bits Set
Floating-point enabled	A floating-point enabled exception is caused when FPSCR[FEX] is set to 1 by the execution of a floating-point instruction that causes an enabled exception, including the case of a Move to FPSCR instruction that causes an exception bit and the corresponding enable bit both to be 1. Note that in this context, the term 'enabled exception' refers to the enabling provided by control bits in the FPSCR.	FP
Illegal instruction	Attempted execution of any of the following causes an illegal instruction exception. • A reserved-illegal instruction or an undefined instruction encoding. • A **mtspr** or **mfspr** that specifies a SPRN value that is not implemented. • A **mtspr** that specifies a read-only SPRN. • A **mfspr** that specifies a write-only SPRN. • A defined, unimplemented instruction. On e500mc an instruction in an invalid form causes boundedly undefined results.	PIL
Privileged instruction	MSR[PR] = 1 and execution is attempted of any of the following: • A privileged instruction or a hypervisor privileged instruction. • **mtspr** or **mfspr** that specifies a privileged SPR. • **mtpmr** or **mfpmr** that specifies a privileged PMR.	PPR
Trap	When any of the conditions specified in a trap instruction are met and the exception is not also enabled as a debug interrupt. If enabled as a debug interrupt (that is, (DBCR0[TRAP] = 1 & DBCR0[IDM] = 1 & MSR[DE] = 1) & (MSR[GS] | ~EPCR[DUVD])), then a debug interrupt is taken instead of the program interrupt.	PTR
Unimplemented operation	e500mc does not take unimplemented operation exceptions. All defined, but unimplemented instructions take an illegal instruction exception.	—

Registers are updated as shown in this table.

Table 4-17. Program Interrupt Register Settings

Register	Description
SRR0	Set to the EA of the instruction that caused the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, CE, and DE are unchanged. • RI is not cleared. • All other defined MSR bits are cleared.
ESR	FP Set if an enabled floating-point exception-type program interrupt; otherwise cleared. PIL Set if an illegal instruction exception-type program interrupt; otherwise cleared. PPR Set if a privileged instruction exception-type program interrupt; otherwise cleared. PTR Set if a trap exception-type program interrupt; otherwise cleared. All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR6[48–59] || 0b0000.

4.9.9 Floating-Point Unavailable Interrupt—IVOR7

A floating-point unavailable interrupt occurs when no higher priority interrupt exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, and move instructions), and the floating-point available bit in the MSR is disabled (MSR[FP] = 0). SRR0, SRR1, and MSR are updated as shown in this table.

Table 4-18. Floating-Point Unavailable Interrupt Register Settings

Register	Description
SRR0	Set to the EA of the instruction causing the floating-point unavailable interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, CE, and DE are unchanged. • RI is not cleared. • All other defined MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR7[48–59] || 0b0000.

4.9.10 System Call/Hypervisor System Call Interrupt—IVOR8/GIVOR8/IVOR40

A system call interrupt occurs when no higher priority exception exists and a System Call (**sc**) instruction with LEV = 0 is executed. (G)SRR0, (G)SRR1, and MSR are updated as shown in Table 4-20.

The system call interrupt is directed to the hypervisor if executed in hypervisor state (MSR[GS] = 0) and is directed to the guest supervisor if executed in guest state (MSR[GS] = 1).

A hypervisor system call interrupt occurs when no higher priority exception exists and a System Call (**sc**) instruction with LEV = 1 is executed. SRR0, SRR1, and MSR are updated as shown in.

This table describes which (G)IVOR is taken based on the setting of MSR[GS] and the value of the LEV operand.

Table 4-19. System Call / Hypervisor System Call Interrupt Selection

LEV	MSR[GS]	Interrupt
> 1	—	Undefined[1]
1	—	IVOR40
0	0	IVOR8
	1	GIVOR8

[1] For e500mc, only the low order bit of the LEV field is used and the (G)IVOR is used accordingly, however software should not depend on this behavior.

Table 4-20. System Call/Hypervisor System Call Interrupt Register Settings

Register	Description
(G)SRR0	Set to the EA of the instruction after the **sc** instruction.
(G)SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, CE, and DE are unchanged. • GS, UCLE, and PMM are cleared if the interrupt is directed to the hypervisor state. • UCLE and PMM are cleared If the interrupt is directed to the guest state and the associated bits of MSRP are 0. • RI is not cleared. • All other defined MSR bits are cleared.

For a system call interrupt, instruction execution resumes at address (G)IVPR[32–47] || (G)IVOR8[48–59] || 0b0000.

For a hypervisor system call interrupt, instruction execution resumes at address IVPR[32–47] || IVOR40[48–59] || 0b0000.

Hypervisor system call interrupts are provided as way to communicate with the hypervisor software.

NOTE

The hypervisor should check SRR1[PR,GS] to determine the privilege level of the software making a hypervisor system call to determine what action, if any, should be taken as a result of the hypervisor system call.

4.9.11 Decrementer Interrupt—IVOR10

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 or MSR[GS])).

NOTE

MSR[EE] also enables external input, processor doorbell, guest processor doorbell, fixed-interval timer, and performance monitor interrupts.

This table shows register updates.

Table 4-21. Decrementer Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, CE, and DE are unchanged. • RI is not cleared. • All other defined MSR bits are cleared.
TSR	DIS is set.

Instruction execution resumes at address IVPR[32–47] || IVOR10[48–59] || 0b0000.

NOTE

To avoid a subsequent redundant decrementer interrupt, software is responsible for clearing the decrementer exception status prior to re-enabling MSR[EE] or MSR[GS]. To clear the decrementer exception, the interrupt handling routine must clear TSR[DIS] by writing a word to TSR using **mtspr** with a 1 in any bit position that is to be cleared and 0 in all other positions. The write-data to the TSR is not direct data, but a mask: A 1 causes the bit to be cleared, and a 0 has no effect.

4.9.12 Fixed-Interval Timer Interrupt—IVOR11

A fixed-interval timer interrupt occurs when no higher priority interrupt exists, a fixed-interval timer exception exists (TSR[FIS] = 1), and the interrupt is enabled (TCR[FIE] = 1 and (MSR[EE] or MSR[GS] = 1)). The “Timers” chapter in the *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors* describes the architecture definition of the fixed-interval timer.

MSR[EE] also enables external input, processor doorbell, guest processor doorbell, decrementer interrupts and performance monitor interrupts.

The fixed-interval timer period is determined by TCR[FP], which, when concatenated with TCR[FPEXT], specifies one of 64 bit locations of the time base used to signal a fixed-interval timer exception on a transition from 0 to 1.

TCR[FPEXT || FP] = 000000 selects bit 0 of the Time Base (TBL[0] or TBU[32]).
TCR[FPEXT || FP] = 11_1111 selects TBL[63].

Registers are updated as shown in this table.

Table 4-22. Fixed-Interval Timer Interrupt Register Settings

Register	Setting
SRR0	Set to the EA of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, CE, and DE are unchanged. • RI is not cleared. • All other defined MSR bits are cleared.
TSR	FIS is set.

Instruction execution resumes at address IVPR[32–47] || IVOR11[48–59] || 0b0000.

NOTE

To avoid redundant fixed-interval timer interrupts, before reenabling MSR[EE], the interrupt handler must clear TSR[FIS] by writing a word to TSR with a 1 in any bit position to be cleared and 0 in all others. Data written to the TSR is a mask. Writing a 1 causes the bit to be cleared; writing a 0 has no effect.

4.9.13 Watchdog Timer Interrupt—IVOR12

A watchdog timer interrupt occurs when no higher priority interrupt exists, a watchdog timer exception exists (TSR[WIS] = 1), and the interrupt is enabled (TCR[WIE] = 1 and (MSR[CE] or MSR[GS] = 1)). The “Timers” chapter in the *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors* describes the architecture definition of the watchdog timer.

NOTE

MSR[CE] also enables the critical input, processor doorbell critical, and guest processor doorbell critical interrupts.

Registers are updated as shown in this table.

Table 4-23. Watchdog Timer Interrupt Register Settings

Register	Setting
CSRR0	Set to the EA of the next instruction to be executed.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME and DE are unchanged. • RI is not cleared. • All other defined MSR bits are cleared.
TSR	WIS is set.

Instruction execution resumes at address IVPR[32–47] || IVOR12[48–59] || 0b0000.

NOTE

To avoid redundant watchdog timer interrupts, before reenabling MSR[CE], the interrupt handling routine must clear TSR[WIS] by writing a word to TSR with a 1 in any bit position to be cleared and 0 in all others. Data written to the TSR is a mask. Writing a 1 to this bit causes it to be cleared; writing a 0 has no effect.

4.9.14 Data TLB Error Interrupt—IVOR13/GIVOR13

A data TLB error interrupt occurs when no higher priority interrupt exists and the exception described in Table 4-24 is presented to the interrupt mechanism. The interrupt is directed to the hypervisor unless the following conditions exist determined as follows:

- The state in which the exception occurred is the guest state (MSR[GS] = 1).
- The interrupt is programmed to be directed to the guest state (EPCR[DTLBGS] = 1).

If all the above conditions are met, the DTLB is directed to the guest supervisor state.

Table 4-24. Data TLB Error Interrupt Exception Condition

Exception	Description
Data TLB miss exception	Virtual addresses associated with a data access do not match any valid TLB entry.

When the interrupt occurs, the processor suppresses execution of the excepting instruction. Registers are updated as shown in this table.

Table 4-25. Data TLB Error Interrupt Register Settings

Register	Setting
(G)SRR0	Set to the EA of the instruction causing the data TLB error interrupt.
(G)SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, CE, and DE are unchanged. • GS, UCLE, and PMM are cleared if the interrupt is directed to the hypervisor state. • UCLE and PMM are cleared If the interrupt is directed to the guest state and the associated bits of MSRP are 0. • RI is not cleared. • All other defined MSR bits are cleared.
(G)DEAR	Set to the EA of a byte that is both within the range of the bytes being accessed by the memory access or cache management instruction and within the page whose access caused the exception.
(G)ESR	[ST] Set if the instruction causing the interrupt is a store, **dcbi**, **dcbz**, or **dcbzl**; otherwise cleared [FP] Set if the instruction causing the interrupt is a floating-point load or store. [EPID] Set if the instruction causing the interrupt is an external pid instruction. All other defined ESR bits are cleared
MAS*n*	If EPCR[DMIUH] = 1, and a Instruction or Data TLB Error, ISI, or DSI is directed to the hypervisor, MAS registers are not changed. See Table 6-6.

Instruction execution resumes at address (G)IVPR[32–47] || (G)IVOR13[48–59] || 0b0000.

NOTE: Implementation

If a store conditional instruction produces an EA for which a normal store would cause a data TLB error interrupt, but the processor does not have the reservation from a load and reserve instruction, e500mc always takes the DTLB interrupt.

4.9.15 Instruction TLB Error Interrupt—IVOR14/GIVOR14

An instruction TLB error interrupt occurs when no higher priority interrupt exists and the exception described in Table 4-26 is presented to the interrupt mechanism. The interrupt is directed to the hypervisor unless the following conditions exist: determined as follows:

- The state in which the exception occurred is the guest state (MSR[GS] = 1).
- The interrupt is programmed to be directed to the guest state (EPCR[ITLBGS] = 1).

If all the above conditions are met, the ITLB is directed to the guest supervisor state.

Table 4-26. Instruction TLB Error Interrupt Exception Condition

Exception	Description
Instruction TLB miss exception	Virtual addresses associated with an instruction fetch do not match any valid TLB entry.

When an instruction TLB error interrupt occurs, the processor suppresses execution of the instruction causing the exception.

Registers are updated as shown in Table 4-27.

Table 4-27. Data TLB Error Interrupt Register Settings

Register	Setting
(G)SRR0	Set to the EA of the instruction causing the instruction TLB error interrupt.
(G)SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, CE, and DE are unchanged. • GS, UCLE, and PMM are cleared if the interrupt is directed to the hypervisor state. • UCLE and PMM are cleared If the interrupt is directed to the guest state and the associated bits of MSRP are 0. • RI is not cleared. • All other defined MSR bits are cleared.
MAS*n*	If EPCR[DMIUH] = 1, and a Instruction or Data TLB Error, ISI, or DSI is directed to the hypervisor, MAS registers are not changed. See Table 6-6.

Instruction execution resumes at address (G)IVPR[32–47] || (G)IVOR14[48–59] || 0b0000.

4.9.16 Debug Interrupt—IVOR15

A debug interrupt occurs when no higher priority interrupt exists, a debug exception is indicated in the DBSR, and debug interrupts are enabled (DBCR0[IDM] = 1 and MSR[DE] = 1). A debug exception occurs when a debug event causes a corresponding DBSR bit to be set.

Any time that a DBSR bit is allowed to be set while MSR[DE] = 0, a special DBSR bit, imprecise debug event (DBSR[IDE]), is also set. DBSR[IDE] indicates that the associated debug exception bit in DBSR was set while debug interrupts were disabled (MSR[DE] = 0). Debug interrupt handler software uses this bit to determine whether the address recorded in DSRR0 should be interpreted as the address associated with the instruction causing the exception or the address of the instruction after the one that set MSR[DE] and thereby enabled the delayed debug interrupt. See Section 4.9.16.2, “Delayed Debug Interrupts.” The “Debug Support,” chapter of the *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors* describes such architectural aspects of the debug interrupt.

Registers are updated as shown in this table.

Table 4-28. Debug Interrupt Register Settings

Register	Description
DSRR0	For exceptions occurring while debug interrupts are enabled (DBCR0[IDM] and MSR[DE] = 1), DSRR0 is set as follows: • For instruction address compare (IAC registers), data address compare (DAC1R, DAC1W, DAC2R, and DAC2W), trap (TRAP), or branch taken (BRT) debug exceptions, set to the EA of the instruction causing the interrupt. • For interrupt taken (IRPT) debug exceptions (CIRPT for critical interrupts), set to the EA of the first instruction of the interrupt that caused the event. • For instruction complete (ICMP) debug exceptions, set to the EA of the instruction that would have executed after the one that caused the interrupt. • For return from interrupt (RET) debug exceptions, set to the EA of the instruction (**rfi**, **rfci**, or **rfgi**) that caused the interrupt. • For unconditional debug event (UDE) debug exceptions, set to the EA of the instruction that would have executed next had the interrupt not occurred. For exceptions occurring while debug interrupts are disabled (DBCR0[IDM] = 0 or MSR[DE] = 0), the interrupt occurs at the next synchronizing event if DBCR0[IDM] and MSR[DE] are modified such that they are both set and if the DBSR still indicates status. When this occurs, DSRR0 holds the EA of the instruction that would have executed next, not the address of the instruction that modified DBCR0 or MSR and caused the interrupt.
DSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, is unchanged.RI is not cleared. • All other defined MSR bits are cleared.
DBSR	Set to indicate type of debug event. See Section 2.17.6, “Debug Status Register (DBSR/DBSRWR).”

Note that on the e500mc, if DBCR0[IDM] is cleared, no debug events occur. That is, regardless of MSR, DBCR0, DBCR1, and DBCR2 settings, no debug events are logged in DBSR and no debug interrupts are taken.

The e500mc complies with the architecture debug definition, except as follows:

- Data address compare is only supported for effective addresses.
- Instruction address compares IAC3 and IAC4 are not supported.
- Instruction address compare is only supported for effective addresses.
- Data value compare is not supported.

Instruction execution resumes at address IVPR[32–47] || IVOR15[48–59] || 0b0000.

4.9.16.1 Suppressing Debug Events in Hypervisor Mode

Synchronous debug events can be suppressed when executing in hypervisor state. This prevents debug events from being recorded (and subsequent debug interrupts from occurring) when executing in hypervisor state when the guest operating system is using the debug facility.

When EPCR[DUVD] = 1 and MSR[GS] = 0, all debug events, except the unconditional debug event, are suppressed and are not posted in the DBSR and the associated exceptions do not occur.

4.9.16.2 Delayed Debug Interrupts

On the e500mc, delayed debug interrupts can be taken under two circumstances:

- A **mtmsr** instruction that sets MSR[DE] = 1 and any DBSR bit is set (including IDE, but excluding MRR). In this case, DSRR0 holds the address of the instruction following the **mtmsr**.
- Any return from interrupt class (**rfi**, **rfgi**, **rfci, rfdi**, **rfmci)** instruction sets MSR[DE] and any DBSR bit is a one (including IDE, but excluding MRR). In this case, DSRR0 holds the address of the target of the return from interrupt instruction.

The e500mc uses DBCR0[IDM] to enable/disable recognition of debug events, and it uses MSR[DE] to enable/disable taking debug interrupts when debug events are recognized. When a debug event is recognized, the event is logged in DBSR and, if debug interrupts are enabled, a debug interrupt also occurs.

A delayed debug interrupt is a delayed response to a previously logged event. Although DBCR0[IDM] is a condition for recognizing and logging a debug event, it is not a condition for taking a delayed debug interrupt. This is different from previous versions of e500, for which a delayed debug interrupt required IDM = 1.

4.9.17 Performance Monitor Interrupt—IVOR35

A performance monitor interrupt is implemented as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. Conditions that can be programmed to trigger an interrupt on the e500mc are described in Section 9.11, "Performance Monitor." The interrupt is triggered by an enabled performance monitor condition or event. For a performance monitor interrupt to be signaled on an enabled condition or event, PMGC0[PMIE] must be set. A PMC*n* register overflow condition occurs with the following settings:

- PMLCa*n*[CE] = 1; that is, for the given counter the overflow condition is enabled.
- PMC*n*[OV] = 1; that is, the given counter indicates an overflow.

Performance monitor counters can be frozen on a triggering enabled condition or event if PMGC0[FCECE] is set.

Although the interrupt condition could occur with MSR[EE] = 0, the interrupt cannot be taken until MSR[EE] or MSR[GS] = 1. If a counter overflows while PMGC0[FCECE] = 0, PMLCa*n*[CE] = 1, and MSR[EE] = 0, the counter can wrap around to all zeros again without the interrupt being taken.

Registers are updated as shown in this table.

Table 4-29. Performance Monitor Interrupt Register Settings

Register	Setting
SRR0	Set to the EA of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, CE, and DE are unchanged. • RI is not cleared. • All other defined MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR35[48–59] || 0b0000.

4.9.18 Doorbell Interrupts—IVOR36–IVOR39

Doorbell interrupts provide a mechanism for a processor to send messages to all devices within its coherence domain. These messages can generate interrupts on core devices, and can be filtered by the processors that receive the message to observe (cause an exception) or to ignore the message.

Doorbell interrupts are useful for sending interrupts to a processor. Power ISA 2.06 defines how processors send messages and the actions that processors take on the receipt of a message. Actions taken by devices other than processors are not defined.

The instructions **msgsnd** and **msgclr** are provided for sending and messages to processors and clearing received and accepted messages. These instructions are hypervisor privileged. See Section 3.4.11.4, "Message Clear and Message Send Instructions."

The e500mc filters, accepts, and handles the following message types defined in Table 4-30. These message types result in the exceptions and interrupts described later in this section.

The message type is specified in the message and is determined by the contents of register **r**B[32–36] used as the operand in the **msgsnd** instruction.

Table 4-30. Message Types

Value	Description
0	Doorbell interrupt (DBELL). Causes a processor doorbell exception on a processor that receives and accepts the message.
1	Doorbell critical interrupt (DBELL_CRIT). Causes a processor doorbell critical exception on a processor that receives and accepts the message.
2	Guest processor doorbell interrupt (G_DBELL). Causes a guest processor doorbell exception on a processor that receives and accepts the message.
3	Guest processor doorbell critical interrupt (G_DBELL_CRIT). Causes a guest processor doorbell critical exception on a processor that receives and accepts the message.
4	Guest processor doorbell machine check interrupt (G_DBELL_MC). Causes a guest processor doorbell machine check exception on a processor that receives and accepts the message.

No other message type is accepted on the e500mc.

4.9.18.1 Doorbell Interrupt Definitions

The architecture defines the following doorbell interrupts, which are implemented on the e500mc:

- Processor doorbell (IVOR36)
- Processor doorbell critical (IVOR37)
- Guest processor doorbell (IVOR38). Note that guest processor doorbell uses GSRR0 and GSRR1 to save state.
- Guest processor doorbell critical (IVOR39)
- Guest processor doorbell machine check (IVOR39)

4.9.18.1.1 Processor Doorbell Interrupt (IVOR36)

A processor doorbell interrupt occurs when no higher priority exception exists, a processor doorbell exception is present, and MSR[EE] or MSR[GS] = 1. Processor doorbell exceptions are generated when doorbell type messages are received and accepted by the processor.

Registers are updated as shown in this table.

Table 4-31. Processor Doorbell Interrupt Register Settings

Register	Setting
SRR0	Set to the EA of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, CE, and DE are unchanged.RI is not cleared. • All other defined MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR36[48–59] || 0b0000.

4.9.18.1.2 Processor Doorbell Critical Interrupt (IVOR37)

A processor doorbell critical interrupt occurs when no higher priority exception exists, a processor doorbell critical exception is present, and MSR[CE] or MSR[GS] = 1. Processor critical doorbell exceptions are generated when doorbell critical type messages are received and accepted by the processor.

Registers are updated as shown in this table.

Table 4-32. Processor Doorbell Critical Interrupt Register Settings

Register	Setting
CSRR0	Set to the EA of the next instruction to be executed.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME and DE are unchanged.RI is not cleared. • All other defined MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR37[48–59] || 0b0000.

4.9.18.1.3 Guest Processor Doorbell Interrupts (IVOR38)

A guest processor doorbell interrupt occurs when no higher priority exception exists, a guest processor doorbell exception is present, and MSR[EE] and MSR[GS] = 1. Guest processor doorbell exceptions are generated when guest doorbell type messages are received and accepted by the processor.

Registers are updated as shown in this table.

Table 4-33. Guest Processor Doorbell Interrupt Register Settings

Register	Setting
GSRR0	Set to the EA of the next instruction to be executed.

Table 4-33. Guest Processor Doorbell Interrupt Register Settings (continued)

Register	Setting
GSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, CE, and DE are unchanged. • RI is not cleared. • All other defined MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR38[48–59] || 0b0000.

NOTE

Even though the guest processor doorbell interrupt is always directed to the hypervisor, it uses GSRR0 and GSRR1 to save state. This is because the interrupt is guaranteed to interrupt out of guest state when it is safe to update the guest save/restore registers. The hypervisor should use this mechanism to reflect interrupts to the guest state. In this scenario, GSRR0 and GSRR1 is already set appropriately for the hypervisor.

4.9.18.1.4 Guest Processor Doorbell Critical Interrupts (IVOR39)

A guest processor doorbell critical interrupt occurs when no higher priority exception exists, a processor doorbell exception is present, and MSR[CE] and MSR[GS] = 1. Guest processor doorbell critical exceptions are generated when guest doorbell critical type messages are received and accepted by the processor.

Registers are updated as shown in this table.

Table 4-34. Guest Processor Doorbell Critical Interrupt Register Settings

Register	Setting
CSRR0	Set to the EA of the next instruction to be executed.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME and DE are unchanged.RI is not cleared. • All other defined MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR39[48–59] || 0b0000.

NOTE

The guest processor doorbell critical and the guest processor doorbell machine check interrupts use the same IVOR to vector interrupts. Software can examine CSRR1 and its own data structures to determine which interrupt occurred.

4.9.18.1.5 Guest Processor Doorbell Machine Check Interrupts (IVOR39)

A guest processor doorbell machine check interrupt occurs when no higher priority exception exists, a guest processor doorbell machine check exception is present, and MSR[ME] and MSR[GS] = 1. Guest

processor doorbell machine check exceptions are generated when guest doorbell machine check type messages are received and accepted by the processor.

Registers are updated as shown in this table.

Table 4-35. Guest Processor Doorbell Machine Check Interrupt Register Settings

Register	Setting
CSRR0	Set to the EA of the next instruction to be executed.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME and DE are unchanged.RI is not cleared. • All other defined MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR39[48–59] || 0b0000.

NOTE

The guest processor doorbell critical and the guest processor doorbell machine check interrupts use the same IVOR to vector interrupts. Software can examine CSRR1 and its own data structures to determine which interrupt occurred.

4.9.19 Hypervisor Privilege Interrupt—IVOR41

A hypervisor privilege exception occurs when the processor executes an instruction in the guest supervisor state and the operation is allowed only in the hypervisor state. A hypervisor privilege exception also occurs when an **ehpriv** instruction is executed, regardless of the state of the processor. See Section 3.4.5.7, "Hypervisor Privilege Instruction."

Registers are updated as shown in this table.

Table 4-36. Hypervisor Privilege Interrupt Register Settings

Register	Setting
SRR0	Set to the EA of the instruction which caused the exception.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• ME, CE, and DE are unchanged.RI is not cleared. • All other defined MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR41[48–59] || 0b0000.

Hypervisor privilege interrupts are provided as a means for restricting the guest supervisor state from performing operations allowed only in the hypervisor state. Table 4-37 lists the resources that cause a hypervisor privilege exception when accessed in guest supervisor state.

Table 4-37. Hypervisor Privilege Exceptions from Guest Supervisor State

Resource	Hypervisor Privilege on Read	Hypervisor Privilege on Write	Hypervisor Privilege on Execute	Notes
Instructions				
ehpriv	—	—	Yes	—
msgclr	—	—	Yes	—
msgsnd	—	—	Yes	—
rfci	—	—	Yes	—
rfdi	—	—	Yes	—
rfi	—	—	No	Guest supervisor state execution of **rfi** maps to **rfgi**.
rfmci	—	—	Yes	—
tlbilx	—	—	Yes or No	Hypervisor privilege occurs only when EPCR[DGTMI] = 1
tlbivax	—	—	Yes	—
tlbre	—	—	Yes	—
tlbsx	—	—	Yes	—
tlbsync	—	—	Yes	—
tlbwe	—	—	Yes	—
SPRs				
CDCSR0	Yes	Yes	—	—
BUCSR	Yes	Yes	—	—
CSRR0	Yes	Yes	—	—
CSRR1	Yes	Yes	—	—
DAC*n*	Yes	Yes	—	—
DBCR*n*	Yes	Yes	—	—
DBSR	Yes	Yes	—	—
DBSRWR	—	Yes	—	Write only register.
DEAR	No	No	—	Guest supervisor state access to DEAR maps to GDEAR.
DEC	Yes	Yes	—	—
DECAR	Yes	Yes	—	e500mc allows reading of DECAR although Power ISA does not define it.
EPCR	Yes	Yes	—	New register, allows hypervisor to direct certain interrupts and mask hypervisor debug events.
EPR	No	No	—	Guest supervisor state access to EPR maps to GEPR.

Table 4-37. Hypervisor Privilege Exceptions from Guest Supervisor State (continued)

Resource	Hypervisor Privilege on Read	Hypervisor Privilege on Write	Hypervisor Privilege on Execute	Notes
ESR	No	No	—	Guest supervisor state access to ESR maps to GESR.
GIVOR*n*	No	Yes	—	Hypervisor privilege occurs on **mtspr** in guest state.
GIVPR	No	Yes	—	Occurs on **mtspr** in guest state.
GPIR	No	Yes	—	—
HID0	Yes	Yes	—	—
IAC*n*	Yes	Yes	—	—
IVOR*n*	Yes	Yes	—	—
IVPR	Yes	Yes	—	—
L1CSR*n*	Yes	Yes	—	—
L2CAPTDATAHI	Yes	Yes	—	—
L2CAPTDATALO	Yes	Yes	—	—
L2CAPTECC	Yes	Yes	—	—
L2CSR*n*	Yes	Yes	—	—
L2ERRADDR	Yes	Yes	—	—
L2ERRATTR	Yes	Yes	—	—
L2ERRCTL	Yes	Yes	—	—
L2ERRDET	Yes	Yes	—	—
L2ERRDIS	Yes	Yes	—	—
L2ERREADDR	Yes	Yes	—	—
L2ERRINJCTL	Yes	Yes	—	—
L2ERRINJHI	Yes	Yes	—	—
L2ERRINJLO	Yes	Yes	—	—
L2ERRINTEN	Yes	Yes	—	—
LPIDR	Yes	Yes	—	—
MAS5	Yes	Yes	—	—
MAS8	Yes	Yes	—	—
MCAR	Yes	Yes	—	—
MCARU	Yes	Yes	—	—
MCSR	Yes	Yes	—	—
MCSRR*n*	Yes	Yes	—	—
MMUCFG	Yes	Yes	—	—

Table 4-37. Hypervisor Privilege Exceptions from Guest Supervisor State (continued)

Resource	Hypervisor Privilege on Read	Hypervisor Privilege on Write	Hypervisor Privilege on Execute	Notes
MMUCSR0	Yes	Yes	—	—
MSRP	Yes	Yes	—	—
NSPC	Yes	Yes	—	—
NSPD	Yes	Yes	—	—
PIR	No	Yes	—	Guest supervisor state access to PIR maps to GPIR for reads.
SPRG0–SPRG3	No	No	—	Guest supervisor state access to SPRG0–SPRG3 maps to GSPRG0–GSPRG3.
SPRG8	Yes	Yes	—	—
SRR0	No	No	—	Guest supervisor state access maps to GSRR0
SRR1	No	No	—	Guest supervisor state access maps to GSRR1
TBL(R)	No	—	—	Read only register
TBL(W)	Yes	Yes	—	
TBU(R)	No	—	—	Read only register
TBU(W)	Yes	Yes	—	
TCR	Yes	Yes	—	—
TLB0CFG	Yes	—	—	Read only register
TLB1CFG	Yes	—	—	Read only register
TSR	Yes	Yes	—	—
USPRG*1-3*[1]	No	No	—	Guest user state access to USPRG*n* maps to GSPRG*n*.
PMRs				
PMC*n*	Yes/no[2]	Yes/no2	—	—
PMLCA*n*	Yes/no2	Yes/no2	—	—
PMLCB*n*	Yes/no2	Yes/no2	—	—
PMGC0	Yes/no2	Yes/no2	—	—

1 USPRG0 is a separate physical register from SPRG0.

2 Access to PMRs is based on the setting of MSRP[PMMP]. If MSRP[PMMP] = 0 reads and writes are allowed to PMRs. If MSRP[PMMP] = 1 reads and writes produce a hypervisor privilege exception is supervisor mode and are NOPed in user mode.

4.10 Guidelines for System Software

When software takes an interrupt, it generally wants to save the save/restore registers in case another exception occurs while processing the current interrupt. In general software must ensure that no other interrupt occurs before the save/restore registers are appropriately saved to memory (usually the stack). Hardware automatically disables asynchronous interrupt enables associated with the save/restore register pair when the new MSR is established taking the interrupt (for example, on taking a interrupt that uses

SRR0/1, MSR[EE] is set to 0 preventing external input, decrementer, fixed interval timer, and processor doorbell interrupts from occurring). Software must ensure that synchronous exceptions do not occur prior to saving the save/restore registers.

This table lists actions system software must avoid before saving save/restore register contents.

Table 4-38. Operations to Avoid Before Save/Restore Register are Saved to Memory

Operation	Reason
Reenabling MSR[EE] , MSR[CE], MSR[DE], or MSR[ME] in interrupt handlers	Prevents any asynchronous interrupts, as well as (in the case of MSR[DE]) any debug interrupts, including synchronous and asynchronous types
Branching (or sequential execution) to addresses not mapped by the TLB or mapped without SX set.	Prevents instruction storage and instruction TLB error interrupts
Load, store, or cache management instructions to addresses not mapped or without permissions.	Prevents data storage and data TLB error interrupts
Execution of System Call (**sc**), trap (**tw**, **twi**, **td**, **tdi**), or **ehpriv** instructions	Prevents system call and trap exception-type program interrupts. Note that ehpriv instructions can be executed in guest supervisor state.
Re-enabling of MSR[PR]	Prevents privileged instruction exception-type program interrupts. Alternatively, software could reenable MSR[PR] but avoid executing any privileged instructions.
Execution of any illegal instructions	Prevents illegal instruction exception-type program interrupts
Execution of any instruction that could cause an alignment interrupt	Prevents alignment interrupts, as described in Section 4.9.7, “Alignment Interrupt—IVOR5.”

4.11 Interrupt Priorities

Except for the occurrence of multiple synchronous imprecise interrupts, all synchronous (precise and imprecise) interrupts are reported in program order, as required by the sequential execution model. 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 the (G)ESR and in any status registers associated with the particular exception.

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 is 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. 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 prevents the setting of

that other exception, independent of whether that other exception's corresponding interrupt type is enabled or disabled.

Except as specifically noted, only one of the exception types listed for a given instruction type is permitted to be generated at any given time.

NOTE

Mutually exclusive exception types otherwise with the same priority are listed in the order suggested by the sequential execution model.

4.12 Exception Priorities

The architecture defines exception priorities for all exceptions including those defined in optional functionality. Exception types are defined to be either synchronous, in which case the exception occurs as a direct result of an instruction in execution, or asynchronous, which occurs based on an event external to the execution of a particular instruction or an instruction removes a gating condition to a pending exception. Exceptions are exclusively either synchronous or asynchronous.

Because asynchronous exceptions may temporally be sampled either before or after an instruction is completed, an implementation can order asynchronous exceptions among only asynchronous exceptions and can order synchronous exceptions among only synchronous exceptions. The distinction is important because certain synchronous exceptions require post-completion actions. These exceptions (for example, system call and debug instruction complete) cannot be separated from the completion of the instruction. therefore, asynchronous exceptions cannot be sampled during the completion and post-completion synchronous exceptions for a given instruction.

Table 4-39 and Table 4-40 describes the relative priority of each exception type. Exception priority is listed from highest to lowest and the lower the numerical relative priorities shown implies a higher priority. In many cases, it is impossible for certain exceptions (such as, the trap and illegal program exceptions) to occur at the same time. Such exceptions are grouped together at the same relative priority.

Table 4-39. Asynchronous Exception Priorities

Relative Priority	Exception	Interrupt Level[1]	Interrupt Nature	Pre or Post Completion[2]	Comments
0	Machine Check	Machine Check	Asynch	N/A	Asynchronous exceptions may come from the processor or from an external source.
1	Guest Processor Doorbell Machine Check	Critical	Asynch	N/A	—

Table 4-39. Asynchronous Exception Priorities (continued)

Relative Priority	Exception	Interrupt Level[1]	Interrupt Nature	Pre or Post Completion[2]	Comments
2	Debug- UDE	Debug	Asynch	N/A	Debug-UDE is generally used for an externally generated high priority attention signal.
	Debug- IDE	Debug	Asynch	N/A	Imprecise debug event usually taken after MSR_{DE} goes from 0 to 1 via **rfdi** or **mtmsr**.
	Debug - Interrupt Taken	Debug	Asynch	N/A	Debug interrupt taken after original interrupt has changed NIA (Next Instruction Address) and MSR.
	Debug - Critical Interrupt Taken	Debug	Asynch	N/A	Debug interrupt taken after original critical interrupt has changed NIA and MSR.
3	Critical Input	Critical	Asynch	N/A	—
4	Watchdog	Critical	Asynch	N/A	—
5	Processor Doorbell Critical	Critical	Asynch	N/A	—
6	Guest Processor Doorbell Critical	Critical	Asynch	N/A	—
7	External Input	Base	Asynch	N/A	—
13	Program - Delayed Floating Point Enabled	Base	Asynch	N/A	Delayed Floating Point Enabled exceptions occur when FPCSR[FEX] = 1 and MSR[FE0,FE1] change from 0b00 to a non-zero value.
22	Fixed Interval Timer	Base	Asynch	N/A	—
23	Decrementer	Base	Asynch	N/A	—
24	Processor Doorbell	Base	Asynch	N/A	—
25	Guest Processor Doorbell	Base	Asynch	N/A	—
26	Performance Monitor	Base	Asynch	N/A	—

1 The interrupt level defines which set of save/restore registers are used when the interrupt is taken. They are: Base: SRR0/1, Critical: CSRR0/1, Debug: DSRR0/1, and Machine Check: MCSRR0/1.

2 Pre or Post Completion refers to whether the exception occurs before an instruction completes (pre) and the corresponding interrupt points to the instruction causing the exception, or if the instruction completes (post) and the corresponding interrupt points to the next instruction to be executed.

Table 4-40. Synchronous Exception Priorities

Relative Priority	Exception	Interrupt Level[1]	Interrupt Nature	Pre or Post Completion[2]	Comments
0	Error Report	Machine Check	Synch	pre	—
8	Debug - Instruction Address Compare	Debug	Synch	pre	—
9	ITLB	Base	Synch	pre	—
	ISI	Base	Synch	pre	—
10	Program - Privileged Instruction	Base	Synch	pre	—
	Embedded Hypervisor Privilege	Base	Synch	pre	—
11	FP Unavailable	Base	Synch	pre	—
12	Debug - Trap	Debug	Synch	pre	—
13	Program - Illegal Instruction	Base	Synch	pre	—
	Program - Unimplemented Operation	Base	Synch	pre	—
	Program - Trap	Base	Synch	pre	—
	Program - Floating Point Enabled	Base	Synch	pre	—
15	DTLB	Base	Synch	pre	—
	DSI	Base	Synch	pre	A DSI Virtualization Fault always takes priority over all other causes of DSI.
16	Alignment	Base	Synch	pre	—
17	System Call	Base	Synch	post	System Call Interrupt has SRR0 pointing to instruction after sc (that is, post completion).
	Embedded Hypervisor System Call	Base	Synch	post	Embedded Hypervisor System Call Interrupt has SRR0 pointing to instruction after sc (that is, post completion).
18	Debug - Return from Interrupt	Debug	Synch	pre	—
	Debug - Return from Critical Interrupt	Debug	Synch	pre	—
	Debug - Branch Taken	Debug	Synch	pre	—
19	Debug - Data Address Compare	Debug	Synch	pre	—
21	Debug - Instruction Complete	Debug	Synch	post	Debug - Instruction Complete Interrupt has DSRR0 pointing to next instruction (that is, post completion).

[1] The interrupt level defines which set of save/restore registers are used when the interrupt is taken. They are: Base: SRR0/1, Critical: CSRR0/1, Debug: DSRR0/1, and Machine Check: MCSRR0/1.

[2] Pre or Post Completion refers to whether the exception occurs before an instruction completes (pre) and the corresponding interrupt points to the instruction causing the exception, or if the instruction completes (post) and the corresponding interrupt points to the next instruction to be executed.

4.13 e500mc Interrupt Latency

Interrupt latency of the e500mc is 10 cycles or less unless a guarded load or a cache-inhibited **stwcx.** instruction is in the last completion queue entry (CQ0).

Chapter 5
Core Caches and Memory Subsystem

This chapter describes the caches and cache structures that are local to the e500mc as well as the e500mc's memory subsystem (MSS), which encompasses the caches, the Load/Store Unit (LSU), the Fetch Unit, and the CoreNet interface (commonly called a Bus Interface Unit, or BIU).

The e500mc core contains separate 32-KB, eight-way set associative level 1 (L1) instruction and data caches to provide the execution units and registers rapid access to instructions and data. It also incorporates a 128-KB unified, eight-way set associative backside L2 cache and provides support for a platform cache implemented by the integrated device.

The LSU manages how data passes between the LSU and the memory resources, both with respect to how data is loaded from system memory into the on-chip caches and to how data used by those instructions is loaded and stored in the caches and system memory.

The Fetch Unit manages how instructions are passed between the memory resources and the caches and into the instruction stream.

The BIU is the interface from the core and its caches to the rest of the integrated device utilizing the CoreNet architecture for access to memory and devices that support transactions to addresses in real storage space.

NOTE

In this chapter, the term 'multiprocessor' is used in the context of maintaining cache coherency. These multiprocessor devices could be processors or other devices that can access system memory, maintain their own caches, and function as bus masters requiring cache coherency.

The terms 'cache line' and 'cache block' are used interchangeably. In particular, cache control instructions include the term 'cache block' in their names. Note that the size of a cache block is determined by the implementation. and that on the e500mc, a cache block, or line, is 16 words.

5.1 Overview

This section lists features of the LSU, the Fetch Unit, the L1 cache, the L2 cache and CoreNet interface.

The LSU has the following features:

- System memory accesses critical quad-word first. For data accesses, the LSU receives the critical quad word as soon as it is available; it does not wait for all 64 bytes. That data is forwarded to the requesting unit before being written to the cache, minimizing stalls due to cache fill latency.

- Store queueing. Stores cannot execute speculatively and remain queued until completion logic indicates that the store is to be committed. When the L1 cache is accessed, stores are deallocated from the queue (regardless of whether the cache is updated). If the address is caching-inhibited, the store passes from the queue to the BIU and into the memory subsystem.
- L1 load miss queueing. On a load miss, the LSU allocates buffers and then queues a bus transaction to read the line. The LSU processes load hits and load misses until one of the following conditions occurs:
  - There are more than nine outstanding load misses.
  - The LSU tries to perform a load miss and there is no place to buffer a new cache line.
- Store miss merging. When a caching-allowed store misses in the data cache, the store data is written to a cache line–wide buffer. The bytes in the cache line not specified by the store are allocated when the cache line is eventually fetched from memory. When all 64 bytes are valid, the cache line is reloaded into the data cache. This behavior is known as store miss merging.

  If a subsequent store miss hits in the buffered data, the new data is buffered along with the original store. Any number of subsequent stores intended for that cache line can be buffered before the corresponding data cache line is allocated.
- Data line fill buffering extends the cache for loads and caching-allowed stores. Accesses to pages marked as cacheable may keep copies of data. Therefore, cache management instructions, such as **dcbf**, are required even if the L1 data cache is disabled.

The L1 cache implementation has the following features:

- Separate 32-KB instruction and data caches (Harvard architecture)
- Eight-way set associative, nonblocking caches
- Physically addressed cache directories. The physical (real) address tag is stored in the cache directory.
- 2-cycle access time provides 3-cycle read latency for instruction and data caches accesses; pipelined accesses provide single-cycle throughput from caches. For details about latency issues, see Chapter 10, “Execution Timing.”
- Instruction and data caches have 64-byte cache blocks. A cache block is the block of memory that a coherency state describes, also referred to as a cache line.
- Four-state modified/exclusive/shared/invalid (MESI) protocol supported for the data cache. See Section 5.5.1, “Data Cache Coherency Model.”
- Both L1 caches support error detection (enabled through L1CSR0 and L1CSR1 bits), as follows:
  - Instruction cache: 1 parity bit per word of instruction, 1 bit of parity per tag
  - Data cache: 1 parity bit per byte of data, 1 bit of parity per tag
    See Section 5.4.4, “L1 Cache Error Detection and Correction.”
- Both caches also support error injection, which provides a way to test error recovery software by intentionally injecting errors into the instruction and data caches. See Section 5.4.5, “Cache Error Injection.”
- The L1 instruction cache supports automatic error correction by invalidation when an access detects a parity error. The subsequent reporting and taking of a machine check or error report interrupt causes the instruction to be refetched after invalidation thus correcting the error.

- The L1 data cache supports automatic error correction by invalidation when operating in write shadow mode. In write shadow mode, all writes to the L1 data cache are written through to the L2 cache. When an access detects an uncorrectable error, the cache is invalidated, and the subsequent reporting and taking of a machine check or error report interrupt causes the instruction to be re-executed after invalidation thus correcting the error. See Section 5.4.2, “Write Shadow Mode”.
- Each cache can be independently invalidated through cache flash invalidate (CFI) control bits located in L1CSR1 and L1CSR0. See Section 5.6.3, “L1 Cache Flash Invalidation.”
- Pseudo–least-recently-used (PLRU) replacement algorithm. See Section 5.8.2.1, “PLRU Replacement.”
- Support for individual line locking. See Section 5.6.4, “Instruction and Data Cache Line Locking/Unlocking.”
- Support for cache stashing to the L1 data cache from other devices in the integrated device.
- Both instruction and data cache lines are filled in a single-cycle, 64 -byte write from line fill buffers as described in Section 5.3.1, “Load/Store Unit (LSU).” Cache line fills write all 64 bytes at once, and therefore do not occur until all data has been buffered from the CoreNet interface.

The L2 write-back, backside cache has the following features:

- Dynamic Harvard architecture, merged instruction and data cache
- 128-KB array organized as 256 eight-way sets of 64-byte cache lines
- 36-bit physical address
- Exclusive, modified, shared, invalid, incoherent, locked, and stale states
- 8-way set associativity with a streaming, 7-bit, pseudo-LRU (PLRU) algorithm with aging replacement
- Supports data- and instruction-only and way partitioned cache operation. See Section 5.9.3, “L2 Configuration and Partitioning.”
- 64-byte (16-word) cache-line, coherency-granule size
- Support for individual line locking. See Section 5.9.2, “L2 Line Locking.”
- The L2 is a victim cache for data lines and generally inclusive for instruction lines. The L2 contains only those cache entries that have been cast out from the L1 data cache (the L2 is not reloaded when the data is reloaded in the L1 data cache). The L1 and L2 caches may or may not have valid copies of the same line at the same time.
- The L2 is reloaded whenever the L1 instruction cache is reloaded, but L1 instruction cache entries remain even if they are evicted from the L2 (there is no back invalidation).
- An instruction fetch does not cause eviction of modified lines if they hit in L2. Both the instruction cache and L2 may have a copy of the line.
- For a transaction with L2 cache, CT = 2, a hit in L1 remains in the L1 unless the transaction is **dcbtls** or **dcbtstls**, which cause the line to be cast out of the L1 cache.See section
- Locked L2 cache lines are not reloaded with a lock in L1 or vice versa.
- L2 cache lookup happens only if L1 cache lookup misses in L1 for the load- or store-type instructions. Snoop starts in L1 and L2 caches in parallel.
- Two-cycle, nonpipelined data array access

- Latency of 9 cycles after L1 access with one access every two cycles
- Configurable ECC or parity protection for data array
- Parity protection for tag array
- Support for cache stashing to the L2 data cache from other devices in the integrated device.
- ABIST support

The BIU is the core's interface manager to CoreNet and the rest of the system. The BIU sends and receives transactions from CoreNet and routes them to the appropriate other units in the core that require them.

The BIU is connected to the CoreNet interface which provides the interprocessor and inter-device connection for address based transactions. CoreNet itself is not described in this document, but has the following features:

- The CoreNet interface fabric provides interconnections among the cores, peripheral devices, and system memory in a multicore implementation. Along with handling basic storage accesses, it manages cache coherency and consistency. CoreNet interfaces run synchronously or asynchronously to the processor core frequency. When asynchronous, it allows arbitrary frequency ratios between the core the rest of the system. The synchronous or asynchronous nature of the CoreNet interface is a function of the design of the integrated device.
- Power Architecture® ordering semantics
- Power Architecture coherency support
- Supports intervention (where a cache line is supplied directly from another cache without having to first be written to memory)
- Non-retry based protocol
- Supports stashing to core caches from certain devices

5.2 The Cache Programming Model

This section describes aspects of the cache programming model architecture in the context of the implementation of architecture-defined resources implemented on the e500mc.

5.2.1 Cache Identifiers

Instructions having a CT (cache target) or TH field for specifying a specific cache hierarchy such as **dcbt**, **dcbtst**, **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, **icblc**, and **icbt** use the values described in Section 3.4.10.1.1, "CT Field Values," for cache targets.

5.2.2 Cache Stashing

Caches may be targets of cache stashing, an operation initiated by a device, specifying a hint that the addresses should be prefetched into a target cache specified by a cache identifier set by system software or predefined by hardware. For the L1 data cache, the identifier is defined in L1CSR2[DCSTASHID]. For the backside L2 cache, the identifier is defined in L2CSR1[L2STASHID]. A cache identifier value of 0 indicates that the cache does not accept or perform stashing.

Cache identifiers (stash IDs) within the entire system should be set to unique values. That is, cache IDs should not be set such that more than one cache in the system has the same ID (other than 0, which disables stashing for that cache). Doing so is considered a programming error and may cause a core or the system to hang.

Like a prefetch or “touch” operation, stashing to a cache is a performance hint. The stash operation initiated by a device can improve performance if the stashed data is prefetched into the targeted cache prior to when the data is accessed. This avoids the latency of bringing the data into the cache at the time it is needed by the processor. However, since stash operations are hints, depending on conditions within the memory hierarchy and the core, stashes may not always be performed when requested. An integrated device that initiate stashing operations to the core can optimize its usage of stashing if it is configured to understand the amount of buffering dedicated to incoming stashing operations.

The e500mc reserves two Data Line Fill Buffers (holding a cacheline of storage each) to perform incoming stashing operations. If both the L1 and L2 cache have stashing disabled, the Data Line Fill Buffers reserved for stashing are freed to be used for other core linefill operations. See the reference manual for the integrated device for information on configuring devices that perform stashes to optimize use of stashing based on the core's resources reserved for handling stashes.

5.3 Block Diagram

The instruction and data caches are integrated with the LSU, the instruction unit, and the bus interface unit in the memory subsystem is shown in Figure 5-1.

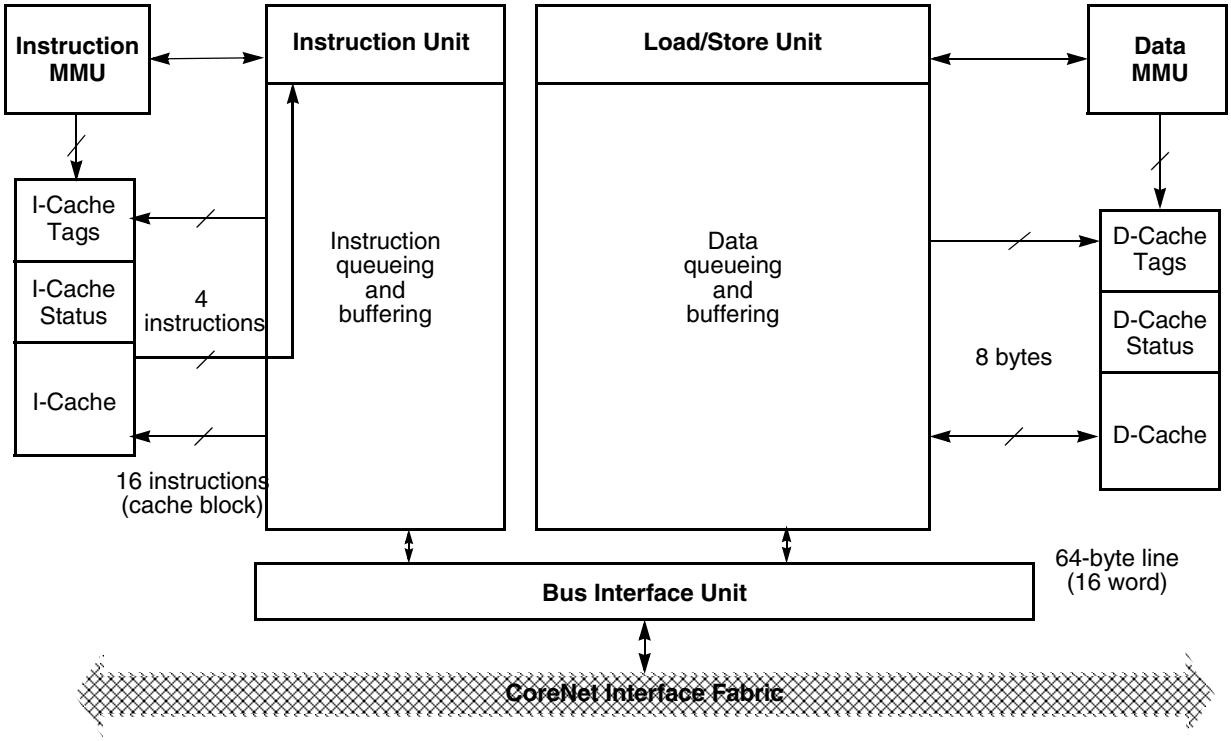


Figure 5-1. Cache/Core Interface Unit Integration

The following sections briefly describe the LSU, the instruction unit, the BIU, and the CoreNet interface.

5.3.1 Load/Store Unit (LSU)

The LSU executes integer and floating-point load instructions and manages transactions between the caches and the register files (GPRs and FPRs). It provides the logic required to calculate effective addresses, handles data alignment, and interfaces with the BIU. Write operations to the data cache can be performed on a byte, halfword, word, or doubleword basis. The data cache is provided with a 64-byte interface (the width of a cache block).

This section provides an overview of how the LSU coordinates traffic in the instruction pipeline with load and store traffic with memory, ensuring that the core maintains a coherent and consistent view of data. See Section 5.5.5, "Load/Store Operation Ordering," for information on architectural coherency implications of load/store operations and the LSU. Section 10.4.3, "Load/Store Execution," describes other aspects of the LSU and instruction scheduling.

5.3.1.1 Caching-Allowed Loads and the LSU

When free of data dependencies, cached loads execute in the LSU in a speculative manner with a maximum throughput of one instruction per cycle and a total 3-cycle latency for integer loads. Data returned from the cache on a load is held in a rename buffer until the completion logic commits the value to the processor state. Cache inhibited loads that are not guarded also execute in the LSU in a speculative manner, but the latency is longer and is based on the latency through the BIU, CoreNet, and the target device.

5.3.1.2 Data Line Fill Buffer (DLFB)

The data line fill buffer (DLFB) is located in the LSU; there are five entries in the DLFB. DLFB entries are used for loads, caching-allowed stores, and cache stashes targeted to the processor. If cache stashing is enabled, two of the five entries are reserved for handling incoming cache stashes. The DLFB acts as a mini-cache. Whenever pages marked as cacheable are accessed, the DLFB (and possibly other internal structures) may keep copies of the data. therefore, cache management instructions, such as **dcbf**, may be required even if the L1 data cache is disabled. Unlike the L1 and L2 caches, if the target of a cache inhibited load is valid in a DLFB, that load returns the data from the DLFB and is not sent to the BIU to be accessed from memory.

DLFBs are updated with data from cacheable stores and the rest of the cache line is obtained from reads of that line from the L2 cache or if not in the L2 cache from CoreNet through the BIU. The DLFB merges the stores (which represent changes to the line) and the non-stored line. When a DLFB acquires a full line of data, the data is written to the L1 data cache.

5.3.2 Instruction Unit

The instruction unit (also called the fetch unit) interfaces with the L1 instruction cache and the BIU. As with the data caches, instructions that miss in the instruction cache are buffered as they are fetched into instruction line fill buffers (ILFBs). After an entire line is available, it is written into the instruction cache.

5.3.3 Bus Interface Unit (BIU)

The bus interface unit handles all ordering and bus protocol and is the interface between the core and the external memory and caches.

The bus interface unit performs transactions through the CoreNet interface by transferring the critical quad word first (16 bytes). The CoreNet interface also captures snoop addresses for the L1 data cache, the L2 data cache, the DLFBs, the MMU (**tlbivax**), the L1 instruction cache (**icbi**), and the memory reservation (load and store conditional instructions) operations.

5.4 L1 Cache Structure

The L1 instruction and data caches are each organized as 64 sets of eight blocks with 64 bytes in each cache line. The following subsections describe the differences in the organization of the instruction and data caches.

5.4.1 L1 Data Cache Dimensions

Figure 5-2 shows the dimensions of the L1 data cache.

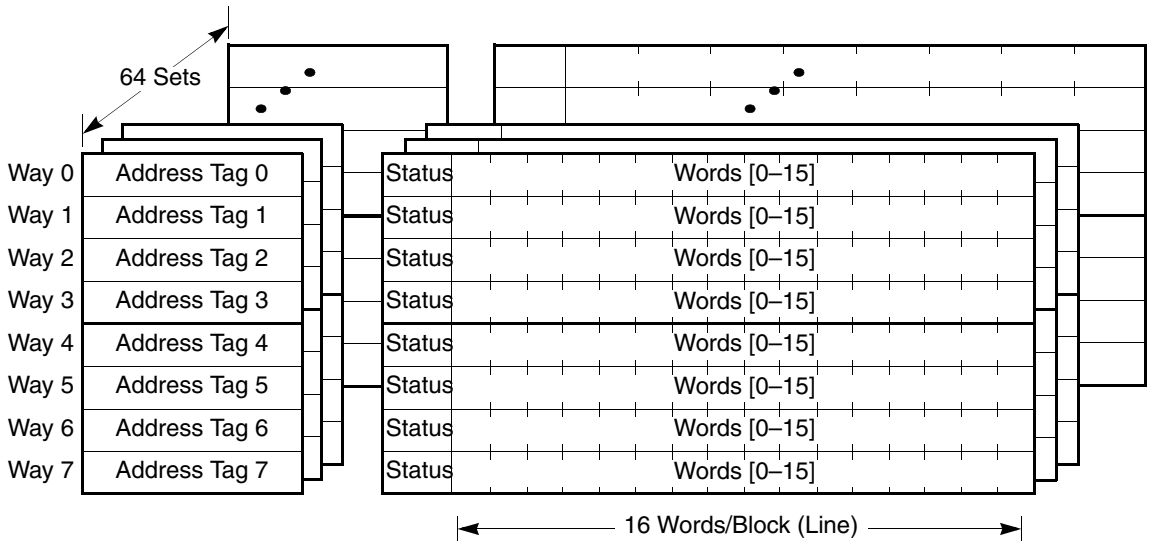


Figure 5-2. L1 Data Cache Organization

Each block (line) consists of 64 bytes of data, 3 status bits (M, V, and S), 1 lock bit, 1 cast-out bit and an address tag. For the L1 data cache, a cache block is the 64-byte cache line. Also, although it is not shown in Figure 5-2, the data cache has 1 parity bit/byte and 1 parity bit/tag.

Each cache block contains 16 contiguous words from memory that are loaded from an 16-word boundary (that is, physical addresses bits 30–35 are zero). Cache blocks are also aligned on page boundaries. Physical address bits PA[24:29] provide the index to select a cache set. The tags consist of physical address bits PA[0:23]. Address translation occurs in parallel with set selection (from PA[24:29]). Lower address bits PA[30:35] locate a byte within the selected block.

The data cache can be accessed internally while a fill for a miss is pending (allowing hits under misses) and the data from a hit can be used as soon as it is available. The LSU forwards the critical doubleword to any pending load misses and allows them to finish. Later, when all the data for the miss has arrived, the entire cache line is reloaded. In addition, subsequent misses can also be sent to the memory subsystem before the original miss is serviced (allowing misses under misses). Up to nine misses can be pending, however those nine misses can only occur to up to five different cache lines.

A cast-out bit indicates whether a cache line chosen for eviction should be cast out to the L2 cache. In general a line is cast out of the L1 cache to the L2 cache when it is victimized for replacement.

5.4.2 Write Shadow Mode

Caching can be configured, by setting L1CSR2[DCWS] = 1 (write shadow mode), such that all modified data in the L1 cache is written through into the L2 cache. If L1CSR2[DCWS] = 0, the L2 cache is generally modified only when an L1 cache line is evicted.

Using write shadow mode ensures that if data or parity tags are corrupted in the L1 cache, it can be invalidated and repopulated with the valid data from the rest of the memory hierarchy.

Only certain configurations of cache operation are supported when using write shadow mode. Invalid configurations are not guaranteed to preserve coherency for store operations performed by the processor. Table 5-1 shows valid configurations for write shadow mode (when L1CSR2[DCWS] = 1).

Table 5-1. Valid Write Shadow Mode Configurations (when L1CSR2[DCWS] = 1)

L1 Data Cache Enabled (L1CSR0[CE])	L2 Cache Enabled (L2CSR0[L2E])	L2 Allocation Policy L2CSR0[L2IO,L2DO]	Supported Configuration?	Notes
Yes	Yes	L2IO=0,L2DO=0	Yes	Normal configuration for write shadow mode.
Yes	Yes	L2IO=1,L2DO=1	Yes	Although the L2IO=1, this special case is supported even though data allocations are not performed in the L2 cache.
No	X	X	No	L1 data cache must always be enabled when using write shadow mode.
Yes	No	X	No	L2 data cache must always be enabled when using write shadow mode.
Yes	Yes	L2IO=1,L2DO=0	No	L2 data cache must be able to allocate and hit on data accesses.

5.4.3 L1 Instruction Cache Organization

The L1 instruction cache is organized as shown in Figure 5-3.

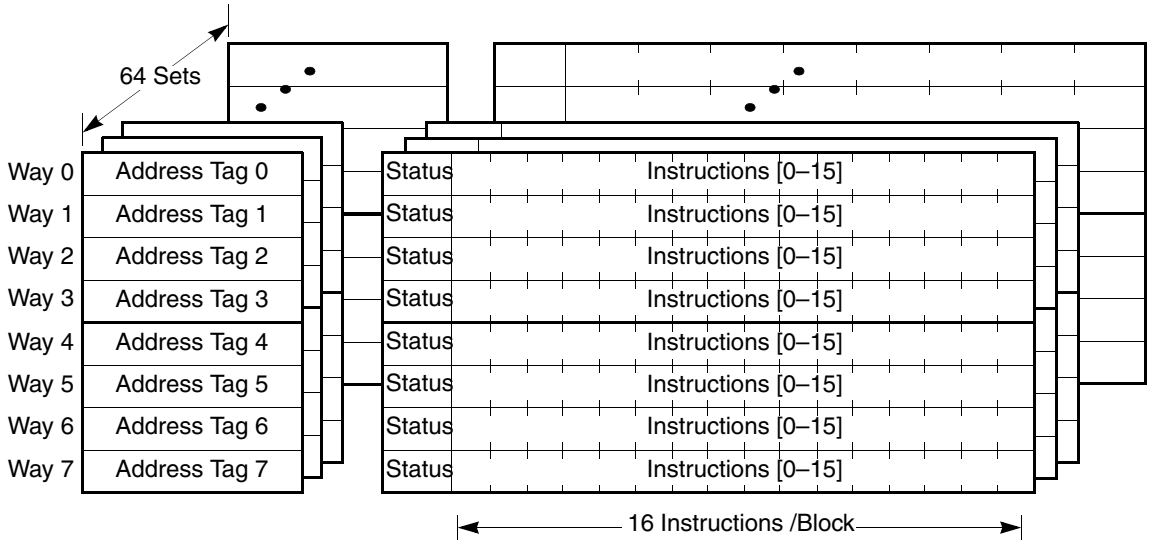


Figure 5-3. L1 Instruction Cache Organization

Each block consists of 16 instructions, 1 status bit (V), 1 lock bit, and an address tag. Also, although it is not shown in Figure 5-3, the instruction cache has 1 parity bit/word (8 parity bits for each line) and one parity bit/tag.

As with the data cache, each block is loaded from a 16-word boundary (that is, bits 30–35 of the physical addresses are zero). Instruction cache blocks are also aligned on page boundaries. Also, PA[24:29] provides the index to select a set and PA[30:33] selects an instruction within a block. The tags consist of physical address bits PA[0:23]. Address translation occurs in parallel with set selection.

The instruction cache can be accessed internally while a fill for a miss is pending (allowing hits under misses). Although the data cannot be used, the hit information stops a subsequent miss from requesting a fill. In addition, subsequent misses can also be sent to the memory subsystem before the original miss is serviced (allowing misses under misses). When a miss is actually updating the cache, subsequent accesses are blocked for 1 cycle. (But up to four instructions being loaded into the instruction cache can be forwarded simultaneously to the instruction unit.)

The instruction cache does not implement a full coherence protocol; a single status bit indicates whether a cache block is valid. Each line has a single bit for locking. Victimized lines from the L1 instruction cache are not cast-out to the L2 cache.

5.4.4 L1 Cache Error Detection and Correction

The L1 instruction cache is protected by parity on both tags and data. Parity information is written into the L1 instruction cache when a line fill occurs (anytime new instructions are written into the cache from possibly a fetch or a cache locking operation.)

The L1 data cache is protected by parity on both tags and data. Parity information is written into the L1 data cache when a line fill occurs (anytime new data is written into the cache.)

L1 cache error detection occurs whenever:

- A load instruction hits in the L1 data cache
- An instruction fetch hits in the L1 instruction cache
- A line is cast out of the L1 data cache

Error detection is performed on the L1 instruction cache using parity for tags and parity for data. The e500mc implements a cache tag parity bit per entry/set. Cache tag parity is checked for all cache transactions.

L1 cache error checking is disabled by default, and can be enabled by setting L1CSR0[CECE] and L1CSR1[ICECE].

If an instruction cache data or tag parity error is detected, the following occurs:

- The instruction cache is automatically flash invalidated. Note L1CSR1[ICEA] = 0 and L1CSR1[ICEDT] = 0 configure the behavior for e500mc. These are the only error actions and detection types supported for the L1 instruction cache.
- a machine check interrupt (or an error report machine check interrupt) occurs (as described in Section 4.9.3, “Machine Check Interrupt—IVOR1”).

If a data cache data or tag parity error is detected, the following occurs:

- If write shadow mode is configured, the data cache is automatically flash invalidated. See Section 5.4.2, “Write Shadow Mode.”
- A machine check interrupt (or an error report machine check interrupt) occurs (as described in Section 4.9.3, “Machine Check Interrupt—IVOR1”).

5.4.5 Cache Error Injection

Cache error injection provides a way to test error recovery software by intentionally injecting errors into the instruction and data caches, as follows:

- If L1CSR1[ICEI] is set, any instruction cache line fill has all of its parity bits inverted in the instruction cache.
- If L1CSR0[CEI] is set, any data line fill has errors injected as follows based on L1CSR0[CEIT] as follows:
  - 0b00: A single-bit error is injected into all the bytes of the cache line which are line filled or which are written as the result of a store operation. The parity bit on the accessed tag during a line allocation is inverted.
  - 0b01: The value is reserved. This may cause boundedly-undefined behavior.

— 0b10: The value is reserved. This may cause boundedly undefined behavior.

— 0b11: The value is reserved. This may cause boundedly undefined behavior.
Line fill operations to the L1 instruction cache can be created by invalidating addresses in the cache using **icbi**, then causing those instructions to be fetched. Line fill operations to the L1 data cache can be created by invalidating addresses using **dcbf** then performing load operations to those addresses. Store operations can be created by writing data to cacheable memory using store (or store class) instructions.
Single-bit errors injected into the data array are accomplished by inverting the parity bit for each byte.

NOTE

Error checking for the L1 instruction cache must be enabled (L1CSR1[ICECE] = 1) when L1CSR1[ICEI] is set. Similarly for the data cache, L1CSR0[CECE] must be set if L1CSR0[CEI] is set. L1CSR0[CEII] cannot be set (using **mtspr**) without setting L1CSR0[CECE].
L1CSR1[ICEI] cannot be set without setting L1CSR1[ICECE].

As described above, if a cache error is detected, a machine check interrupt occurs. Sources for cache errors are described in Section 4.9.3, "Machine Check Interrupt—IVOR1."

5.5 Cache Coherency Support and Memory Access Ordering

This section describes the L1 cache coherency and coherency support.

5.5.1 Data Cache Coherency Model

The L1 data cache and L2 cache supports a MESI (Modified/Exclusive/Shared/Invalid) based cache coherency protocol for each cache line.

The MESI based protocol supports efficient and frequent sharing of data between masters.

Each 64-byte data cache block contains status that define the coherency state of the cache line. The CoreNet interface uses this status to support coherency protocols and to direct coherency operations. Table 5-2 describes general data cache states.

Table 5-2. Cache Line State Definitions

Name	Description
Modified (M)	The line in the cache is modified with respect to main memory. It does not reside in any other coherent cache.
Exclusive (E)	The line is in the cache, and this cache has exclusive ownership of it. It is in no other coherent cache and it is the same as main memory. This processor may subsequently modify this line without notifying other bus masters.
Shared (S)	The addressed line is in the cache, it may be in another coherent cache, and it is the same as main memory. It cannot be modified by any processor.
Invalid (I)	The cache location does not contain valid data.

Every data cache block state is defined by its status. Note that in a multiprocessor system, a cache line can exist in the exclusive state in at most one L1 data cache at a time.

The core provides full hardware support for cache coherency and ordering instructions and for TLB management instructions.

The core broadcasts cache management instructions (**dcbst**, **dcbstep**, **dcbf**, **dcbi** (M=1), **icbi**, **icbiep**), synchronization instructions (**mbar** - all forms, **sync 0**), TLB management instructions (**tlbsync**, **tlbivax**), and cache touch or locking instructions with CT=1.

5.5.2 Instruction Cache Coherency Model

The instruction cache supports only invalid and valid states.

The instruction cache is loaded only as a result of instruction fetching or by an Instruction Cache Block Touch and Lock Set (**icbtls**) instruction. It is not snooped for general coherency with other caches; however, it is snooped when the Instruction Cache Block Invalidate (**icbi** or **icbiep**) instruction is executed by this processor or any other processor in the system. Instruction cache coherency must be maintained by software and is supported by a fast hardware flash invalidation capability as described in Section 5.6.3, "L1 Cache Flash Invalidation." Also, the flushing requirement of modifying code from the data cache is described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

5.5.3 Snoop Signaling

Hardware maintains cache coherency by snooping address transactions on the CoreNet interface. Software enables such transactions to be made visible to other masters in the coherence domain by setting the coherency-required bit (M) in the TLBs (WIMGE = 0bxx1xx). The M bit state is sent with the address on CoreNet transactions. If asserted, the CoreNet interface transaction should be snooped by other bus masters.

The instruction cache is not snooped, except in the case of transactions initiated by a **icbi**, so coherency must be maintained by software.

5.5.4 WIMGE Settings and Effect on Caches

All instruction and data accesses are performed under control of the WIMGE bits. This section generally describes how WIMGE bit settings affect the behavior of the L1 and L2 caches when accesses are marked with the "M" bit set (that is, are coherent). The detailed description of all the states and transitions are beyond the scope of this manual. For more information about WIMGE bits and their meanings. see the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

5.5.4.1 Write-Back Stores

A write-back store is a store to a memory address that has a WIMGE setting of 0b00xxx.

A write-back store that hits a line that is already in exclusive state is immediately stored to the line; the state is changed to modified. If a write-back store hits a line that is already in the modified state, it is immediately stored to the line, and the line stays as modified.

If a write-back store operation (that is, caching-allowed and not write-through) hits a line in the shared state, the cache line is first invalidated and a read-with-intent-to-modify is issued to the BIU and CoreNet. The line is received through the BIU and the written data is merged into the line in the DLFB. The line is then written to the cache marked as modified. If a write-back store misses in the cache, the action is the same as the shared case, except the line is not first invalidated (as it is not present).

5.5.4.2 Write-Through Stores

A write-through store is a store to a memory address that has a WIMGE setting of 0b10xxx.

A write-through store operation may hit an exclusive cache line. In this case, the store data is written into the data cache and the write-through store goes to the L2, the BIU and CoreNet as a single-beat write. The cache line stays exclusive.

A write-through store operation may hit in a shared cache line. In this case, that cache line is invalidated from the cache, and the write-through store goes to the BIU and CoreNet as a single-beat write.

A write-through store may also hit in a cache line that is already in the modified state. This situation normally occurs as a result of page table aliasing in which two effective addresses are mapped to the same physical page, but with one mapped as write-through and the other mapped as write-back (that is, not write-through). In this case, the cache line remains in its current state, the store data is written into the data cache, and the store goes to the BIU and CoreNet as a single-beat write. Such aliasing should in general not be used as coherency is not enforced outside of the processor that performs the aliasing.

If a copy exists in both the L1 and the L2 cache and the L2 is enabled, a data write-through store also updates the L2 copy.

5.5.4.3 Caching-Inhibited Loads and Stores

A caching-inhibited load or store (WIMGE = 0bx1xxx) that hits in the cache presents a cache coherency paradox and is normally considered a programming error. If a caching-inhibited load hits in the cache, the cache data is ignored and the load is provided from the BIU as a single-beat read. If a caching-inhibited store hits in the cache or the DLFB, the cache (or DLFB) may be altered but the store is performed on CoreNet anyway as a single-beat write.

A caching inhibited load that hits in a DLFB is serviced out of the DLFB and is not sent to the BIU or CoreNet and is not seen outside the processor. This is a special case of the cache coherency paradox and can produce results not intended by software. If the aliasing of caching and caching inhibited writes must be performed, software should ensure that all cached addresses are flushed with **dcbf** followed by **sync** before executing caching-inhibited loads and stores using the aliased addresses.

5.5.4.4 Misaligned Accesses and the Endian (E) Bit

Misaligned accesses that cross page boundaries could corrupt data if one page is big endian and the other is little endian. When this situation occurs, the core takes a DSI and sets the BO (byte ordering) bit in the exception syndrome register (ESR) instead of performing the accesses.

5.5.4.5 Speculative Accesses and Guarded Memory

If a memory area is marked as execute-permitted (UX/SX = 1), there is no restriction on how the core performs instruction fetching from guarded memory and software should assume that any page that is marked as execute-permitted generates instruction fetches even if software never attempts to execute those addresses. This is because the fetch unit can generate fetch addresses based on mispredicted speculative paths for which the resulting addresses would be such that they are never actually generated by software. Note that to prevent inadvertent instruction fetching from memory, such memory should be marked as no-execute (UX/SX = 0). Then, if the effective address of a fetched instruction is in no-execute memory, an execute access control exception occurs, preventing the access from occurring to that address.

Speculative data accesses to memory have special consideration as well. Memory address must be marked as guarded (G = 1) to prevent speculative load accesses to those addresses. Like speculative fetching, the processor can generate any effective memory address as the result of a mispredicted branch (including forming addresses on that path from index registers which may hold unknown contents at the time). Thus to avoid inadvertent speculative references that may cause undesired results, memory that is not “well behaved” (well-behaved memory can tolerate speculative reads without any side effects) should always be marked as guarded (G = 1) or if there is no underlying real addresses in the system, should not be mapped in the TLB.

The core does not perform speculative stores to guarded memory (or to any memory). However, loads from guarded memory may be accessed speculatively if the target location is valid in the data cache.

For more information, see the *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors*.

5.5.5 Load/Store Operation Ordering

Load and store operations in Power Architecture are considered to be weakly ordered. That is, certain memory accesses can be performed in a different order than the sequential processor execution model specifies them. While this appears extraordinarily complicated to the programmer, in fact several restrictions placed by the architecture, *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors*, and the implementation simplify this greatly. In practice this requires that the programmer only really be aware of the ordering of memory accesses that are used by another core or another device and the other core or device care about the order. In general, this reduces even further to the following three scenarios:

- The SMP case
  - Code is running on more than one processor
  - Data being manipulated is accessed from more than one processor.

— Software is designed, in general, with some sort of mutual exclusion or locking mechanism regardless of the architecture (because software running on one processor must make several updates to data structure atomically).

- The device driver case
  - — Code is running that controls a device through memory-mapped addresses.
  - — Accesses to these memory-mapped registers usually need to occur in a specific order because the accesses have side effects (for example a store to an address causes the device to perform some action and the order these actions are performed must be explicitly controlled in order for the device to perform correctly).
  - — Addresses are usually marked as caching-inhibited and guarded because the memory is not "well behaved."
- The processor synchronization case.
  - — Some registers within the processor, such as the MSR, have special synchronization requirements associated with them to guarantee when changes which may effect memory accesses, occur. (see Section 3.3.3, "Synchronization Requirements," for the specific registers and their synchronization requirements).
  - — Only system programmers modifying these special registers need be aware of these cases.

5.5.5.1 Architecture Ordering Requirements

Power Architecture and *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* require certain memory accesses to be ordered implicitly, as follows:

1. All loads and stores appear to execute in-order on the same processor. That is, each memory access a processor performs, if that memory location is not stored to by another processor or device, it appears to be performed in order to the processor. For example, a processor executes the following sequence:

   ```
   lwz   r3,0(r4)
   lwz   r5,100(r4)
   ```

   Because there is no way for the processor to distinguish which order these loads occurred in (because the memory is "well behaved"), the loads can be performed in any order. Similarly the sequence

   ```
   stw   r3,0(r4)
   stw   r5,100(r4)
   ```

   may also be performed out of order because the processor cannot distinguish which order the stores are performed in. However, the sequence

   ```
   stw   r3,0(r4)
   lwz   r5,0(r4)
   ```

   must be performed in order because the processor can distinguish a difference depending on whether the store or the load is performed first.

   In general this means that the processor performs memory accesses in order between any two accesses to overlapping addresses. The core may decide that accesses overlap if they touch the same cache line and not merely a common byte.
2. Any load or store that depends on data from a previous load or store must be performed in order. For example, a load retrieves the address that is used in a subsequent load:

```
lwz    r3,0(r4)
lwz    r5,0(r3)
```

Because the second load's address depends on the first load being performed and providing data, the processor must ensure that the first load occurs before the second is attempted (and in fact must be sure the first load has returned data before even attempting translation).

3. Guarded caching-inhibited stores must be performed in order with respect to other guarded caching-inhibited stores and guarded caching-inhibited loads must be performed in order with respect to other guarded caching-inhibited loads. This generally only applies to writing device drivers that control memory mapped devices with side effects through store operations.
4. A store operation cannot be performed before a previous load operation regardless of the addresses. That is a load is followed by a store, then the load is always performed before the store is. This is an *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* requirement of Freescale processors. It is unlikely, but possible that other Power Architecture cores may not require this.

5.5.5.2 Forcing Load and Store Ordering (Memory Barriers)

The implicit ordering requirements enforced by the processor handle the vast majority of all the programming cases when accessing memory locations from a single core. Normal software should only be concerned in ordering when the memory locations being accessed are done so in an SMP environment or the memory locations are part of a device's memory mapped locations. If these cases occur, then software must place explicit memory barriers to control the order of memory accesses. A memory barrier causes ordering between memory accesses that occur before the barrier in the instruction stream and memory accesses that occur after the barrier in the instruction stream.

There are four memory barriers that can be used on e500mc to order memory accesses, depending on the type of memory (the WIMGE attributes) being accessed and the level of performance desired. Memory barriers, by definition, can slow down the processor because they prevent the processor from performing loads and stores in their most efficient order. The barriers from strongest (that is, enforces the most ordering between different types of accesses) to the weakest are:

- **sync** (or **sync** 0 or **msync**)—**sync** creates a barrier such that *all* (regardless of WIMGE attributes) memory accesses that occur before the **sync** are performed before any accesses after the **sync**. **sync** also ensures that no other instructions after the **sync** are initiated until the instructions before the **sync** and the **sync** itself, have performed their operations. **sync** also has the most negative effect on performance. **sync** can be used regardless of the memory attributes of the access and can be used in the place of any of the other barriers. However, it should only be used when performance isn't an issue, or if no other barrier orders the memory accesses.
- **mbar** (or **mbar** 0)—**mbar** creates the same barrier **sync** does, however it does not restrict instructions following **mbar** from being initiated. It does prevent memory accesses following the **mbar** from being performed until all the memory accesses prior to the **mbar** have been performed. **mbar** affects performance almost as much as **sync** does.
- **mbar** 1—**mbar** 1 creates a memory barrier that is the same as the **eieio** instruction from the original PowerPC architecture. It creates two different barriers:

— Loads and stores that are both caching-inhibited and guarded (WIMGE = 0b01*x*1*x*) as well as stores that are write-through required (WIMGE = 0b10*xxx*). This is useful for the device driver case which would be doing loads and stores to caching-inhibited memory.

— Stores that have the following attributes: not caching-inhibited, not write-through required, and memory coherence required (WIMGE = 0b001*xx*). These are stores to normal cacheable coherent memory.

mbar 1 is a better performing memory barrier than **sync** or **mbar**. For more details refer Chapter 5-Instruction Set in the *"EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors"*

- **lwsync** (or **sync** 1)—**lwsync** (lightweight sync) creates a barrier for normal cacheable memory accesses (WIMGE = 0b001*xx*). It orders all combinations of the loads and stores *except* for a store followed by a load. This is the most efficient barrier for normal SMP programming when dealing with multiprocessor locks and critical regions.

  lwsync is a better performing memory barrier than **sync**, **mbar**, or **mbar** 1.

Another method also exists for ordering all caching-inhibited loads and stores which are guarded. The HID0[CIGLSO] bit can be set to force all caching-inhibited loads and stores which are guarded to be performed in order. This is not a barrier, per se, but a system attribute that causes the core to always order these accesses. Setting this bit is a good way to deal with the device driver case over a broad range of code if the memory accesses to the device are caching-inhibited and guarded which is normally the case. This is likely to perform better than inserting **mbar** in specific places since the implementation of the e500mc already orders all of these except for a guarded caching-inhibited store followed by a guarded caching-inhibited load. In this case, the e500mc simply ensures that the store is performed on CoreNet prior to attempting the load.

5.5.5.2.1 Simplified Memory Barrier Recommendations

The general simplistic recommendation for adding required barriers is as follows:

- For the device driver case, device drivers that access caching-inhibited memory, ensure that memory is also guarded and at boot time set HID0[CIGLSO] to 1. This should order all such cache-inhibited guarded accesses. If there is software that deals with other types of memory attributes (or needs to order accesses between cached and caching-inhibited memory), those barriers must be inserted into the code at the appropriate places. In general, those barriers are **mbar** 0.
- For the SMP case, normally all that needs to be done is to deal with interactions between multiple cores. This is generally already isolated into locking routines that acquire multiprocessor locks and release multiprocessor locks. In general, all that is required to modify such routines is to:
  — Insert a **lwsync** barrier after the lock has been acquired, and before the first load of any data protected by the lock. This ensures that the load of the protected data structure occurs after the load of the lock itself. Note that **lwarx** and **stwcx.** should be used to ensure the lock is properly acquired.
  — Insert a **lwsync** barrier after the last store to the protected data structure and the store that releases the lock. This ensures that the store to the protected data structure occurs prior to the store that releases the lock.

Locking software and multiprocessing software may have various other types of mutual exclusion and those should be examined with ordering semantics in mind. Power ISA 2.06 Book II Appendix B gives programming examples for various types of shared storage accesses.

5.5.5.3 Memory Access Ordering

Table 5-3 displays the Power ISA and *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* memory access ordering requirements based on the WIMG attributes and access type. For access where the attributes differ, ordering between these types of access generally requires **mbar** 0 (or **sync**) except that write-through required and guarded caching inhibited loads or stores may be ordered with **mbar** 1. For Table 5-3, entries suggest the most efficient barrier (or may suggest more than one). 'Yes' means that the given ordering is already guaranteed by the architecture and no barrier is required. Not all possible barriers are listed and **sync** 0 or **mbar** 0 enforces all barriers.

Table 5-3. Architectural Memory Access Ordering

Memory Access Attributes	WIMGE	Store-Store Ordered	Load-Load Ordered	Store-Load Ordered	Load-Store Ordered
Caching-inhibited and Guarded	0b01x1x	Yes	Yes	HID0[CIGLSO] mbar 1	Yes
Caching-inhibited and non Guarded	0b01x0x	mbar 0	mbar 0	mbar 0	Yes
Write-through	0b10xxx	mbar 1	mbar 0	mbar 0	Yes
Write-back	0b00xxx	lwsync	lwsync	mbar 0	Yes

5.5.5.4 msgsnd Ordering

It may be required to order when messages are sent (which may cause interrupts on other cores) with stores performed by the core executing **msgsnd**. A typical example of this is a producer stores a value in memory and then sends a message to another core to cause an interrupt telling the receiving core that there is work for it to do (represented by the stores performed by the sending processor). In this case, a **sync 0** should be placed between the stores and the **msgsnd**. this guarantees that the store is performed before the message is sent.

In all respects of memory ordering and barriers, **msgsnd** is ordered as if it is a cache inhibited store.

5.5.5.5 Atomic Memory References

The e500mc implements **lwarx** and **stwcx.** as described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

The e500mc takes a data storage interrupt if the location is write-through required but does not take the interrupt if the location is caching inhibited (i.e caching inhibited reservations are permitted). Software should avoid all possible using reservations on storage that are caching inhibited as future cores may not support these.

If the EA is not naturally aligned for any load and reserve or store conditional instruction, an alignment interrupt is invoked.

As specified in the architecture, the core requires that, for a store conditional instruction to succeed, its real address must be to the same reservation granule as the real address of a preceding load and reserve instruction that established the reservation. The e500mc makes reservations on behalf of aligned 64-byte blocks of the memory address space.

If the reservation has been canceled for any reason (or the reservation does not match the real address specified by the store conditional instruction), then the store conditional instruction fails and clears CR0[EQ]. The reservation may be invalidated by several events. They are described in Section 3.4.9, "Reservations."

5.6 L1 Cache Control

This section describes how the cache control instructions and L1CSR*n* bits are used to control the L1 cache.

5.6.1 Cache Control Instructions

The e500mc implements the cache control instructions as described in Section 3.4.10.1, "User-Level Cache Instructions," and Section 3.4.11.3.1, "Supervisor-Level Cache Instruction." Note that on the e500mc, Data Cache Block Store (**dcbst**) is mapped to **dcbf**, **dcbstep** is mapped to **dcbfep**, and Instruction Cache Touch (**icbt**) when CT=0 is treated as a NOP.

If the effective address cannot be translated, all cache control instructions generate TLB miss exceptions except **dcba**, **dcbal**, **dcbt**, **dcbtep**, **icbt**, **dcbtst**, and **dcbtstep**, which are treated as NOPs (and do not cause DAC debug exceptions).

If a **dcbt**, **dcbtep**, **dcbtst**, or **dcbtstep** instruction accesses a page marked caching-inhibited, it is treated as a NOP.

5.6.2 Enabling and Disabling the L1 Caches

The instruction and data caches are enabled and disabled with the cache enable bits, L1CSR0[CE] and L1CSR1[ICE], respectively. Disabling a cache does not cause all memory accesses to be performed as caching inhibited. When caching-inhibited accesses are desired, the pages must be marked as caching inhibited in the MMU pages.

When either the instruction or data cache is disabled, the cache tag state bits are ignored and the corresponding cache is not accessed. Caches are disabled at start-up L1CSR0[CE] and L1CSR1[ICE] = 0.

Disabling the data cache has the following effects:

- Touch instructions (**dcbt**, **dcbtst**, **dcblc**, **dcbtls**, **dcbtstls**, **icblc**, and **icbtls**) targeted to the disabled cache do not affect the cache.
- A **dcbz**, **dcbzl**, **dcba**, or **dcbal** instruction to a disabled data cache zeros the cache line in memory, but does not affect the cache when it is disabled.
- Cache lines are not snooped. Before the data cache is disabled it must be flushed and invalidated to prevent coherency problems when it is enabled again.

- Cacheable data accesses bypass the data cache, are forwarded to the memory subsystem as caching-allowed, and proceed to the CoreNet interface. Returned data is forwarded to the requesting execution unit, but is not loaded into any of the caches.
- Other cache management instructions do not affect the disabled cache.

NOTE

Data line fill buffering, which extends the cache for loads and caching-allowed stores, remains enabled. Pages marked as cacheable are accessed and may keep copies of data. therefore, cache management instructions, such as **dcbf**, may be required even if the L1 data cache is disabled.

When the instruction cache is disabled (L1CSR1[ICE] = 0), instruction accesses bypass the instruction cache. These accesses are forwarded to the memory subsystem as caching-allowed and proceed to the CoreNet interface. When the instructions are returned, they are forwarded to the instruction unit but are not loaded into the instruction cache.

NOTE

Instruction line fill buffering, which extends the cache for fetches, remains enabled. Pages marked as cacheable are accessed by performing a cache-line burst transaction even when the cache is disabled and may keep copies of instructions in line fill buffers. therefore, cache management instructions, such as **icbi**, may be required even if the L1 instruction cache is disabled.

When an L1 cache is enabled, software must first properly flash invalidate it to prevent stale data (in the case where it has been disabled for some period of time during operation) or unknown state (in the case of power on reset). Software should perform the invalidation by setting the flash invalidation bit (CFI or ICFI) in the appropriate L1 cache control and status register, and then continue to read CFI (or ICFI) until the bit is cleared. Software should then perform an **isync** to ensure that instructions that may have been prefetched prior to the cache invalidation are discarded. The setting of L1CSR0[CE] or L1CSR1[ICE] must be preceded by a **sync** and **isync** instruction, to prevent a cache from being disabled or enabled in the middle of a data or instruction access. See Section 3.3.3, “Synchronization Requirements,” for more information on synchronization requirements.

5.6.3 L1 Cache Flash Invalidation

The data cache can be invalidated by executing a series of **dcbi** instructions, or it can be flash invalidated by setting L1CSR0[CFI]. The data cache is automatically flash invalidated if write shadow mode is configured and any unrecoverable error (tag parity or data parity) occurs. See Section 5.4.2, “Write Shadow Mode.”

If software can guarantee that data is not modified, the cache can be invalidated without updating system memory; if a modified line is invalidated, the data is lost. To prevent the loss of data, modified cache lines must be flushed, as described in Section 5.7, “L1 Data Cache Flushing.”

Because the instruction cache never contains modified data, there is no need to flush the instruction cache before it is invalidated.

The instruction cache can be invalidated by setting L1CSR1[ICFI]. The L1 caches can be flash invalidated independently. The setting of L1CSR0[CFI] and L1CSR1[ICFI] must be preceded by an **msync** and **isync**, respectively.

The instruction cache is automatically flash invalidated if any parity error (tag or data) occurs.

Valid bits in both caches are cleared automatically upon reset. If software desires to clear all valid bits in the caches during operation, software must use the Flash Invalidation bits in L1CSR0 and L1CSR1 (CFI bits). This causes a flash invalidation, after which the CFI bits are cleared automatically (CFI bits are not sticky). Flash invalidate operations are local only to the processor which performs them, other processor's L1 caches are not affected. Software should always poll the CFI bits after setting them to determine when the invalidation has been completed and then perform an **isync**. Software must use care when invalidating the entire data cache to ensure that no modified data exists in the cache by first flushing the cache unless software does not care about the state that any previous memory operations may have attained.

Individual instruction or data cache blocks can be invalidated by using **icbi** and **dcbi**. Note that invalidating the caches resets lock bits (causing the locks to be lost) in the L1 caches. Also note that with **dcbi**, the e500mc invalidates the cache block without pushing it out to memory if WIMGE=0bx00xx. If WIMGE=0bx01xx, the e500mc performs a **dcbf** and pushes any modified state to memory before invalidating the cache block. See Section 3.4.11.3.1, "Supervisor-Level Cache Instruction."

Exceptions and other events that can access the L1 cache should be disabled during this time so that the PLRU algorithm can function undisturbed.

5.6.4 Instruction and Data Cache Line Locking/Unlocking

User-mode instructions perform cache line locking/unlocking based on the complete address of the cache line. **dcbtls**, **dcbtstls**, and **dcblc** are for data cache locking and unlocking and **icbtls** and **icblc** are for instruction cache locking and unlocking. For descriptions, see Section 3.4.10.2, "Cache Locking Instructions." The CT operand is used to indicate the cache target of the cache line locking instruction. See Section 3.4.10.1.1, "CT Field Values."

Lock instructions (including **icbtls** and **icblc**) are treated as loads when translated by the data TLB, and they cause exceptions when data TLB errors or data storage interrupts occur.

The user-mode cache lock enable bit, MSR[UCLE], is used to restrict user-mode cache line locking by the operating system. If MSR[UCLE] = 0, any cache lock instruction executed in user mode (MSR[PR] = 1) causes a cache-locking DSI exception and sets either ESR[DLK] or ESR[ILK]. This allows the OS to manage and track the locking/unlocking of cache lines by user-mode tasks. If MSR[UCLE] is set, the cache-locking instructions can be executed in user mode and do not cause a DSI for cache locking. However, they may still cause a DSI for access violations.

Table 5-4 shows how cache locking operations are affected by MSR[GS,PR,UCLE] and MSRP[UCLEP] which determine whether the core is operating in hypervisor, guest-supervisor, or user (problem state mode).

Table 5-4. Cache Locking Based on MSR[GS,PR,UCLE] and MSRP[UCLEP]

MSR[GS]	MSR[PR]	MSR[UCLE]	MSRP[UCLEP]	Result
0	0	—	—	Execute
x	1	0	—	DSI, ESR[DLK or ILK] set
x	1	1	—	Execute
-	0	—	0	Execute
1	0	—	1	Embedded hypervisor privilege

If all of the ways are locked in a cache set, an attempt to lock another line in that set results in an overlocking situation. The new line is not placed in the cache, and either the data cache overlock bit L1CSR0[CLO] or instruction cache overlock bit L1CSR1[ICLO] is set. This does not cause an exception condition. See Section 3.4.10.2, “Cache Locking Instructions” for a description of what conditions set these bits.

It is acceptable to lock all ways of a cache set. A nonlocking line fill for modified data to a new address in a completely locked cache set is not put into the cache. However it is loaded into a DWB and creates the appropriate normal burst write transfer.

The cache-locking DSI handler must decide whether to lock a given cache line based on available cache resources.

5.6.4.1 Effects of Other Cache Instructions on Locked Lines

Other cache management instructions have no effect on the locked state of lines unless that instruction causes an invalidate operation on that line. If a **dcbi**, **icbi**, **icbiep**, **dcbf**, **dcbfep**, **dcbst**, or **dcbstep** target a locked line, the line is invalidated and the lock is cleared.

5.6.4.2 Effects of Stores on Locked Lines

Stores can also cause line locks to be cleared. A store to a locked line which is in shared state can cause the line to be invalidated before the store is performed, causing the lock on the line to be lost. To avoid this scenario, if a locked line is to be stored to, it should not be used by a processor other than the one which has it locked.

5.6.4.3 Flash Clearing of Lock Bits

The core allows flash clearing of the instruction and data cache lock bits under software control. Each cache’s lock bits can be independently flash cleared through the CLFC control bits in L1CSR0 and L1CSR1.

Lock bits in both caches are cleared automatically upon reset. If software desires to clear all lock bits in the caches during operation, software must use the CLFC controls. Setting CLFC bits causes a flash

invalidation of the lock bits performed in a single CPU cycle, after which the CLFC bits are automatically cleared (CLFC bits are not sticky).

5.7 L1 Data Cache Flushing

Any modified entries in the data cache can be copied back to memory by using a cache flushing instruction (**dcbf**, **dcbst**, **dcbfep**, or **dcbstep**) if the particular addresses which are required to be flushed are known by software. However, in some cases, system software may need to force modified data in the L1 data cache to be written to memory, and the contents of the cache as well as whether the contents are modified is not known. This can happen when software wishes to go to a power management state in which the cache does not retain state or a number of other conditions when system software wishes to know that the L1 data cache contains no modified state. Forcing all the modified lines in the L1 data cache is called a cache flush. To perform a cache flush, software must ensure that all valid lines in the cache are replaced by performing a series of reads (loads) in which the cache lines which are read force all the lines to be replaced. When a modified line is replaced (evicted), the processor writes any modified data in the replaced line to the memory subsystem.

Selection of lines to replace in the L1 data cache when a line is accessed is determined by the PLRU (pseudo least recently used) bits, whether a given line is locked, and whether the line already exists in the L1 data cache. The cache flush algorithm must control these factors including how the PLRU bits for other lines in the cache are affected. In effect each set must have enough accesses to cause all ways in the set to be evicted given how the PLRU bits are set. The method for performing a cache flush is as follows:

- Block all interrupts (set MSR[EE,CE,DE,ME] = 0). This prevents an interrupt from occurring during the flush algorithm and changing the cache state.
- Perform **sync**. This ensures that any stores that have completed are performed.
- Unlock any locked cache lines or ensure that the locked lines are flushed. This can be skipped if system software does not allow cache line locking, or if it is known that no locked lines contain modified data. Locks can be cleared by writing a 1 to L1CSR0[CLFC], performing the required synchronization and polling until the bit is clear. If system software knows the addresses of all lines locked in the cache it could instead perform **dcbf** or **dcbst** type instructions to these lines.
- Ensure that all the existing lines in the cache are replaced through a series of operations which cause new lines to be allocated which contain no modified data, or if the new lines contain modified data, those modifications can be discarded. To accomplish this perform one of the following methods:
  - Perform a series of loads which access each cache line once within a contiguous real 52-KB region. Software must ensure that no cache line within the 52-KB region is in the L1 data cache in the modified state prior to performing the loads.
  - Perform a series of loads or **dcbz** instructions which access each cache line once within a contiguous real 48-KB scratch region. Software must ensure that no cache line within the 48-KB region is in the L1 data cache in any state prior to performing the loads or **dcbz** instructions. This can be ensured by only mapping the real pages in the region in the MMU when the cache flushing routine is performed. The pages must be marked as guarded and cacheable.

— Set HID0[DCFA], perform **isync**, then perform a series of loads which access each cache line once within a contiguous real 36-KB region. Software must ensure that no cache line within the 52-KB region is in the L1 data cache in the modified state prior to performing the loads. Clear HID0[DCFA], perform **isync**.

— Set HID0[DCFA], perform **isync**, then perform a series of loads or **dcbz** instructions which access each cache line once within a contiguous real 32-KB scratch region. Software must ensure that no cache line within the 32-KB region is in the L1 data cache in any state prior to performing the loads or **dcbz** instructions. This can be ensured by only mapping the real pages in the region in the MMU when the cache flushing routine is performed. The pages must be marked as guarded and cacheable. Clear HID0[DCFA], perform **isync**.

- Ensure that all the replaced lines have been written to the memory subsystem by executing **sync**.
- Flash invalidate the cache by writing a 1 to L1CSR0[CFI], performing the required synchronization, then polling until the bit is cleared. This ensures that the memory region that was used to cause line replacement in the cache is not present in the cache should the cache flush routine get called again before the lines get naturally evicted.
- Re-enable any interrupts that were disabled at the beginning of the cache flush routine.

NOTE

Since the hypervisor can interrupt the guest in the middle of the cache flush routine, this can cause the PLRU bits to change and perturb the flush algorithm possibly leaving modified lines in the L1 data cache which are not flushed. This can be handled by either having the hypervisor treat the setting of L1CSR0[CFI] to 1 by the guest as a flush and invalidate request, or by providing an hcall service to perform the flush.

5.8 L1 Cache Operation

This section describes operations performed by the L1 instruction and data caches.

5.8.1 Cache Miss and Reload Operations

This section describes the actions taken by the L1 caches on misses for caching-allowed accesses. It also describes what happens on cache misses for caching-inhibited accesses as well as disabled and locked L1 cache conditions.

5.8.1.1 Data Cache Fills

The core data cache blocks are filled (sometimes referred to as a cache reload) from an L2 cache or the memory subsystem when cache misses occur for caching-allowed accesses, as described in Section 5.3.1, “Load/Store Unit (LSU).”

When the data cache is disabled (L1CSR0[CE] = 0), cacheable data accesses bypass the data cache, are forwarded to the memory subsystem as caching-allowed, and proceed to the CoreNet interface. Returned data is forwarded to the requesting execution unit, but is not loaded into any of the caches. Such transactions are kept in DLFBs. See Section 5.6.2, “Enabling and Disabling the L1 Caches.”

Each of the eight ways of each set in the data cache can be locked (by locking all of the cache lines in the way with the **dcbtls** or **dcbtstls** instruction). When at least one way is unlocked, misses are treated normally and are allocated to one of the unlocked ways on a reload. If all eight ways are locked, store/load misses proceed to the memory subsystem as normal caching-allowed accesses. In this case, the data is forwarded to the requesting execution unit when it returns, but it is not loaded into the data cache. If the data is modified, it creates the appropriate normal burst write transfer.

Note that caching-inhibited stores should not access any of the caches (see Section 5.5.4.3, "Caching-Inhibited Loads and Stores," for more information).

5.8.1.2 Instruction Cache Fills

The instruction cache provides a 128-bit interface to the instruction unit, so as many as four instructions can be made available to the instruction unit in a single clock cycle on an instruction cache hit. On instruction cache hits, the instructions are delivered directly from the instruction cache to the instruction unit.

On a miss, an instruction line fill buffer is allocated and the fetch request is sent to the L2 cache. On an L2 cache hit, the data from the L2 cache is stored in the line fill buffer. When all the data in the line fill buffer is received, the instructions in the fetch group from the fetch are transferred to the instruction unit and the line fill buffer is written to the instruction cache, writing the entire cache line. If the L2 cache misses, the caching-allowed access is sent to the memory subsystem and CoreNet interface. When data is returned and all bytes in the line fill buffer are received, the instructions in the fetch group from the fetch are transferred to the instruction unit and the line fill buffer is written to the instruction cache, writing the entire cache line. In this case the L2 cache also receives the data returned from CoreNet and writes the cache line to the L2 cache. When data from an instruction line fill buffer is written to the instruction cache is loaded in one 64-byte write from the line fill buffer.

The instruction cache is non-blocking, providing for hits under misses.

If the instruction cache is disabled (L1CSR1[ICE] = 0) or all ways of the associated set are locked, the instruction line fill buffer is not written to the instruction cache when the instruction line fill buffer has received all the data. The instruction cache operates similarly to the data cache when all eight ways of a set are locked.

For caching-inhibited instruction fetches the instruction unit fetches up to four instructions at a time directly from the memory subsystem by performing a cache-line burst (although the transaction is marked as caching-inhibited) and discards the other instructions which are not part of the fetch group. When data is returned and all bytes in the line fill buffer are received, the instructions in the fetch group from the fetch are transferred to the instruction unit.

Caching-inhibited fetches utilize a line fill buffer to perform their read operation to the memory subsystem, but in general do not use the other instructions returned which were not part of the fetch group and the other instructions are effectively discarded when the line fill buffer becomes invalid after the fetch is complete.

5.8.1.3 Cache Allocation on Misses

Instruction cache misses cause a new line to be allocated into the instruction cache on a PLRU basis, provided the cache is not completely locked or disabled.

If there is a data cache miss for a caching-allowed load or store (including touch instructions) and the line is not already going to be allocated into the data cache as a result of a previous load/store miss, the miss causes a new line to be allocated into the data cache on a PLRU basis, provided the cache is not completely locked or disabled. A store that is write-through or caching-inhibited that misses in the data cache does not cause an allocation. Also, cache operations such as **dcbi** and **dcbf** that miss in the cache do not cause a fill.

5.8.1.4 Data Cache Block Push Operation

When an L1 cache block in the core is snooped (by another bus master) and the data hits and is modified, the cache block must be made available to the snooping device. The push operation propagates to the intervention buffer and then to the CoreNet interface.

5.8.2 L1 Cache Block Replacement

When a new block needs to be placed in the instruction or data cache, a line in the set is chosen to hold the new block. If any line in the set is invalid, the lowest numbered way that is invalid is chosen. If no line is invalid, the pseudo-least-recently-used (PLRU) replacement algorithm is used. Note that data cache replacement selection is performed at reload time and not when the miss occurs. Instruction cache replacement selection occurs when an instruction cache miss is first recognized.

When a cache line is accessed, it is tagged as the most-recently-used line of the set. When a miss occurs, if all lines in the set are valid (occupied), the least-recently-used line is replaced with the new data. The PLRU bits in the cache are updated each time a cache hit occurs based on the most-recently-used cache line.

Modified data to be replaced is written back to main memory.

Data load or write-back store accesses that miss in the L1 data cache function similarly to L1 instruction cache misses. They cause a new line to be allocated on a PLRU basis, provided the cache is not completely locked or disabled.

Note that modified data in the replacement line of the data cache can cause a cast-out to occur to the CoreNet interface. In all such cases, the cast-out is not initiated until new data is ready to be loaded.

5.8.2.1 PLRU Replacement

Block replacement is performed using a binary decision tree, PLRU algorithm. There is an identifying bit for each cache way, L[0–7]. There are seven PLRU bits, B[0–6] for each set in the cache to determine the line to be cast out (replacement victim). The PLRU bits are updated when a new line is allocated or replaced and when there is a hit in the set.

This algorithm prioritizes the replacement of invalid entries over valid ones (starting with way 0). Otherwise, if all ways are valid, one is selected for replacement according to the PLRU bit encodings shown in Table 5-5.

Table 5-5. L1 PLRU Replacement Way Selection

PLRU Bits						Way Selected for Replacement
B0	0	**B1**	0	**B3**	0	L0
	0		0		1	L1
	0		1	**B4**	0	L2
	0		1		1	L3
	1	**B2**	0	**B5**	0	L4
	1		0		1	L5
	1		1	**B6**	0	L6
	1		1		1	L7

Figure 5-4 shows the decision tree used to generate the victim line in the PLRU algorithm.

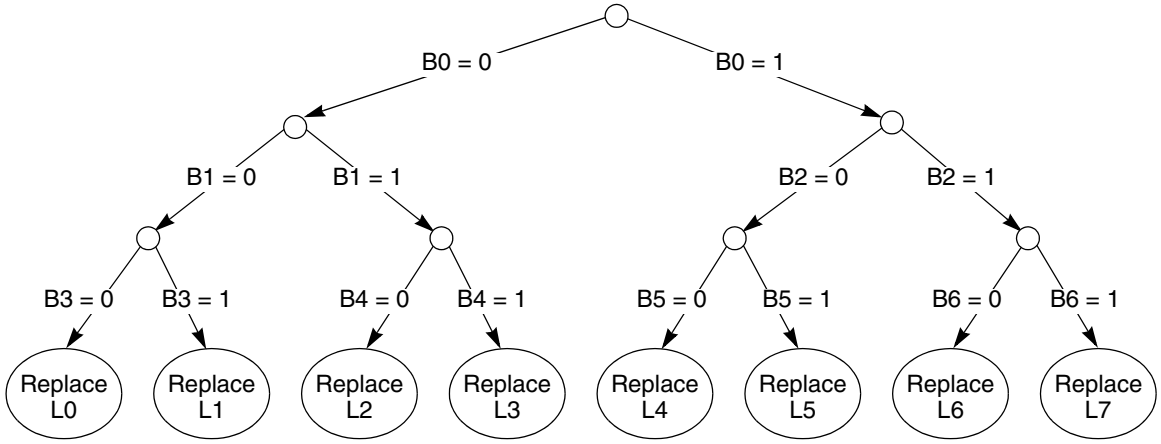


Figure 5-4. PLRU Replacement Algorithm

During reset, the PLRU and valid bits of the L1 caches are automatically cleared to point to way L0 of each set.

5.8.2.2 PLRU Bit Updates

Except for snoop accesses, each time a cache block is accessed, it is tagged as the most-recently-used way of the set. For every hit in the cache or when a new block is reloaded, the PLRU bits for the set are updated using the rules specified in Table 5-6.

Table 5-6. PLRU Bit Update Rules

Current Access	New State of the PLRU Bits						
	B0	B1	B2	B3	B4	B5	B6
L0	1	1	No change	1	No change	No change	No change
L1	1	1	No change	0	No change	No change	No change
L2	1	0	No change	No change	1	No change	No change
L3	1	0	No change	No change	0	No change	No change
L4	0	No change	1	No change	No change	1	No change
L5	0	No change	1	No change	No change	0	No change
L6	0	No change	0	No change	No change	No change	1
L7	0	No change	0	No change	No change	No change	0

Note that only three PLRU bits are updated for any access.

The core does not replace locked lines. Lock bits are used at reload time to steer the PLRU algorithm away from selecting locked cache lines.

5.9 Backside L2 Cache

The L2 write-back, backside cache has the following features:

- Dynamic Harvard architecture, a unified instruction and data cache, with some characteristics of a harvard (split instruction and data) cache
- 128-KB array organized as 256 eight-way sets of 64-byte cache lines
- 36-bit physical address
- Exclusive, modified, shared, incoherent, invalid, locked, and stale states
- 8-way set associativity with selectable replacement algorithms:
  - pseudo-LRU (PLRU)
  - streaming PLRU
  - streaming PLRU with aging
- Supports unified-, instruction-, and data-only cache operation
  - The L2 cache can be programmed as unified, instruction-only, or data-only. Data-only prevents cache lines from being allocated on an instruction fetch miss. Instruction-only prevents cache lines from being allocated when a line is victimized from the L1 data cache.
  - Partitioning can be configured through L2CSR0[L2WP]. If the L2 cache is configured to allocate lines for both data and instruction accesses that miss in the L2 cache (L2CSR0[L2IO,L2DO] are both 0) the ways are partitioned to allocate new lines in ways based on whether the allocation is for instructions or data.
  - 64-byte (16-word) cache-line, coherency-granule size

  See Section 5.9.3, “L2 Configuration and Partitioning.”
- Supports line locking using CT = 2. Unlike the L1 line locking, the L2 locking is persistent and locks are not lost when a line is invalidated.
- Data side is a victim cache. The L2 contains only those cache entries that have been cast out from the L1 data cache (the L2 is not reloaded when the data is reloaded in the L1 data cache).
- Configurable ECC or parity protection for data array
- Parity protection for tag array
- ABIST support

5.9.1 Dynamic Harvard Implementation

The L2 cache is implemented as a unified cache. That is, entries in the cache can be either instructions that were fetched, or data resulting from L1 data cache cast outs. The L2 cache treats lines that are fetched as instructions as incoherent in a manner similar to the way that the line would be treated if the L2 cache had separate instruction and data caches (as for example, the L1 caches are). Instead of providing a separate structures for instruction and data, the fetched instructions are marked with a status bit (N) to denote that the line was loaded incoherently. Once N is set, L2 data-side transactions do not hit to it, and when a fetch establishes an instruction line in the L2 cache that fetch access is performed non global and is not snooped by other processors. This L2 cache implementation is called “Dynamic Harvard” since it has the properties of a harvard cache in that the behaviors of the instruction side and the data side differ, but also has the

properties that the instruction side and the data side both allocate out of the same pool of available lines (that is, the cache is physically unified).

This dynamic harvard implementation allows fetches to be treated as non global and reduces the overall snoop overhead that otherwise might be required by the system, while still allowing instructions and data lines to allocate from the same pool of available lines in the L2 cache. This means that the amount of lines in use by instructions or data varies according to how the processor is executing.

When N is set for any line (when it is allocated as the result of an instruction fetch), the transaction to read that line is sent to CoreNet and marked as non global. A later data transaction does not hit to that line, and any data transaction that targets a line with the N bit set is sent out to CoreNet to acquire coherent data. When the data line is received by the L2 cache, if a line with the same tag exists which is valid and has the N bit set, the line is replaced in the L2 cache by the data line and the N bit status is cleared.

To implement dynamic harvard, the L2 cache snoops **icbi** operations that are performed, regardless of the core that performs them. Also operations on the processor that can potentially fill the L2 cache from the fetch path must be propagated to the L2 cache. **icbi** operations do not hit to lines that are marked as coherent (N is not set), since the operation effects the instruction cache only. Similarly, snoops for data operations from data cache block operations, or from stores do not hit to lines that are marked as incoherent (N is set) since the operation effects the data cache only.

Software must deal with the incoherence of instruction lines in the L2 cache in the same manner that it does with the harvard L1 instruction cache. To perform instruction modification, data must first be pushed from the L2 cache, and when that operation is complete, the instruction side must be invalidated using **icbi**. Power Architecture already requires software to perform this operation, so no additional software is required. If software had previously depended on the flash invalidation of the L1 instruction cache to clear any cache fetched instructions, this method does not work when the L2 cache is enabled and caching instruction fetches. For this reason, software is strongly encouraged to perform the architectural method of modifying instructions using **dcbf** and **icbi**.

5.9.2 L2 Line Locking

Lines are locked in L2 cache by software using a series of “touch and lock set” instructions. The following instructions can lock a line in L2 cache:

- Data Cache Block Touch and Lock Set—**dcbtls** (CT = 2)
- Data Cache Block Touch for Store and Lock Set—**dcbtstls** (CT = 2)
- Instruction Cache Block Touch and Lock Set—**icbtls** (CT = 2)

Similarly, lines are unlocked from L2 cache by software using a series of “lock clear” instructions. The following instructions are used to clear the lock in L2 cache.

- Data Cache Block Lock Clear—**dcblc** (CT = 2)
- Instruction Cache Block Lock Clear—**icblc** (CT = 2)

There is no distinction between **icblc** and **dcblc** in the L2, because both clear the lock on a line regardless of whether the lock was previously established as an instruction side or data side lock.

Software can clear all the locks in the L2 cache by L2CSR0[L2LFC], as described in Section 2.15, "L2 Cache Registers." Note that this operation takes many cycles.

5.9.3 L2 Configuration and Partitioning

The L2 cache can be programmed as data-only, instruction-only, or unified, through the L2CSR0[L2IO,L2DO] fields. Setting L2IO (without setting L2DO) indicates that L2 cache lines are allocated only for instruction cache transactions that miss in the L2 cache (preventing lines from being allocated for data). Data accesses do not hit in the L2 cache and are not allocated due to the setting of L2IO. Such accesses are serviced by other parts of the memory hierarchy. Data transactions are not snooped and any lines in the L2 cache are not coherent with respect to data transactions.

Attempting to execute data locking instructions when L2IO = 1, causes L1CSR0[CUL] to be set.

Setting L2DO (without setting L2IO) indicates that cache lines are allocated only for data transactions. The L2 cache continues to hit data transactions and participate in the coherence protocol. Instruction fetches hit in the L2 cache but no new instruction fetches allocate.

Setting both L2DO and L2IO prevents any new lines from being allocated in the L2 cache, effectively locking the entire L2 cache.

Partitioning can be configured through L2CSR0[L2WP]. If the L2 cache is configured to allocate lines for both data and instruction accesses that miss in the L2 cache (L2CSR0[L2IO,L2DO] are both 0) the ways are partitioned to allocate new lines in ways based on whether the allocation is for instructions or data.

A value of 0 allows all ways to be used for either instructions or data. A nonzero value specifies the number of ways to be used for allocating instructions. The number of ways specified for data references is the total number of ways minus the value in the L2WP field (for example, the value 1 makes one way available for instruction allocation, and seven ways available for data allocation). See Section 2.15.2, "L2 Cache Control and Status Register (L2CSR0)."

5.9.4 Special Scenarios for Backside L2

This section describes special scenarios of operations in the L2 cache.

5.9.4.1 Instruction Cache Block Invalidate (icbi)

icbi operations are snooped from the CoreNet interface. If an **icbi** snoop hits, the line is invalidated if it is marked as non-coherent. No special actions are performed for **icbi** executed on the local processor as those operations are also snooped when the **icbi** is sent out on the CoreNet interface.

5.9.5 Errors

Table 5-7 describes when L2ERRDET is updated based on error type.

Table 5-7. Errors in Different Arrays

Error	L2CSR0	L2ERRDIS					L2ERRINTEN					L2ERRDET				
	L2PE	TMHITDIS	TPARDIS	MBECCDIS	SBECCDIS	PARDIS	TMHITINTEN	TPARINTEN	MBECCINTEN	SBECCINTEN	PARINTEN	TMHIT	TPARERR	MBECCERR	SBECCERR	PARERR
Tag multi-way hit	0	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0
	1	0	x	x	x	x	0	x	x	x	x	1	0	0	0	0
	1	0	x	x	x	x	1	x	x	x	x	1	0	0	0	0
	1	1	x	x	x	x	0	x	x	x	x	0	0	0	0	0
	1	1	x	x	x	x	1	x	x	x	x	0	0	0	0	0
Tag parity error	0	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0
	1	x	0	x	x	x	x	0	x	x	x	0	1	0	0	0
	1	x	0	x	x	x	x	1	x	x	x	0	1	0	0	0
	1	x	1	x	x	x	x	0	x	x	x	0	0	0	0	0
	1	x	1	x	x	x	x	1	x	x	x	0	0	0	0	0
Data parity error	0	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0
	1	x	x	1	1	0	x	x	x	x	0	0	0	0	0	1
	1	x	x	1	1	0	x	x	x	x	1	0	0	0	0	1
	1	x	x	1	1	1	x	x	x	x	x	0	0	0	0	0
Single bit ECC error	0	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0
	1	x	x	x	0	x	x	x	x	0	x	0	0	0	1	0
	1	x	x	x	0	x	x	x	x	1	x	0	0	0	1	0
	1	x	x	x	1	x	x	x	x	x	x	0	0	0	0	0
Multi bit ECC error	0	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0
	1	x	x	0	x	x	x	x	0	x	x	0	0	1	0	0
	1	x	x	0	x	x	x	x	1	x	x	0	0	1	0	0
	1	x	x	1	x	x	x	x	x	x	x	0	0	0	0	0

5.9.6 Performance Monitor Events

Performance monitor events associated with the L2 cache are described in 9.11.6, “Event Selection.”

Chapter 6
Memory Management Units (MMUs)

This chapter describes the implementation details of the e500mc MMU. The *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* provides full descriptions of the MMU definition, and the register, instruction, and interrupt models as they are defined by the Power ISA™ and the Freescale implementation standards.

6.1 e500mc MMU Overview

The e500mc extends the MMU design of previous e500 cores to address the additional needs presented by the integration of multiple cores in a single integrated device. In particular, resources are defined that support the additional privilege level required to distinguish system-wide, hypervisor level access from user and guest supervisor privilege levels. In particular, the machine state register (MSR) problem state (MSR[PR]) and guest state (MSR[GS]) bits together determine privilege level, as follows:

- User state (problem state): MSR[PR] = 1
- Hypervisor state: MSR[PR] = 0, MSR[GS] = 0
- Guest supervisor state: MSR[PR] = 0, MSR[GS] = 1

Resources are defined that identify the logical partition with which a memory access is associated. In particular, a logical partition is identified by the value in the logical partition ID register (LPIDR).

The LPIDR and MSR[GS] fields now form part of the virtual address for memory accesses and are compared against corresponding fields in the TLBs (TLPID and TGS), as shown in Section 6.2, "Effective-to-Real Address Translation."

e500mc cores employ a two-level MMU architecture with separate data and instruction level 1 (L1) MMUs in hardware backed up by a unified level 2 (L2) MMU. The L1 MMUs are completely invisible with respect to the architecture and software programming model. The programming model for implementing translation look-aside buffers (TLBs) provided by the architecture applies to the L2 MMU.

NOTE

Because a "bare-metal" operating system has no knowledge of explicit embedded hypervisor resources for partitioning (such as the LPIDR register and MSR[GS]), these values should remain unchanged from 0 values, in effect producing the same virtual address spaces that exist without the embedded hypervisor functionality. That is, the virtual addresses that are produced are essentially:

```
0 ||   0  || AS || PID || EA
0 ||   0   || AS ||  0  || EA
```

In practice this produces the same effect as not having embedded hypervisor.

6.1.1 MMU Features

The e500mc core has the following features:

- 32-bit effective address (EA) translated to 36-bit real (physical) address (using a 48-bit interim virtual address)
- Two-level MMU containing a total of six TLBs for maximizing TLB hit rates
- Processor register (PID) for supporting up to 255 translation IDs at any time in the TLB
- TLB entries for variable-sized, 4-Kbyte to 4-Gbyte pages and fixed-size (4-Kbyte) pages
- No page table format is defined; software is free to use its own page table format.
- TLBs maintained by system software through the TLB instructions and nine MMU assist MAS registers

The Level 1 MMUs have the following features:

- Two 8-entry, fully-associative TLB arrays (one for instruction accesses and one for data accesses) supporting the eleven variable size page (VSP) page sizes shown in Section 6.2.3, “Variable-Sized Pages.”
- Two 64-entry, 4-way set-associative TLB arrays (one for instruction accesses and one for data accesses) that support only 4-Kbyte pages
- L1 MMU access occurs in parallel with L1 cache access time (address translation/L1 cache access can be fully pipelined so one load/store can be completed on every clock).
- Performs parallel L1 TLB lookups for instruction and data accesses
- L1 TLB entries are a proper subset of TLB entries resident in L2 MMU (completely maintained by the hardware).
- Automatically performs invalidations to maintain consistency with L2 TLBs

The Level 2 MMU has the following features:

- A 64-entry, fully-associative unified (for instruction and data accesses) L2 TLB array (TLB1) supports the 11 VSP page sizes shown in Section 6.2.3, “Variable-Sized Pages.”
- A 512-entry, 4-way set-associative unified (for instruction and data accesses) L2 TLB array (TLB0) supports only 4-Kbyte pages.
- Hardware assistance for TLB miss exceptions
- TLB1 and TLB0 managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, **tlbivax**, **tlbilx**, and **mtspr** instructions
- Performs invalidations in TLB1 and TLB0 caused by **tlbivax** and **tlbilx** instructions executed by this core. Performs invalidations in TLB1 when MMUCSR0[L2TLB1_FI] is set and invalidations in TLB0 when MMUCSR0[L2TLB0_FI] is set. Snoops TLB1 and TLB0 for **tlbivax** invalidations executed by other masters.
- Setting IPROT implemented in TLB1 protects critical entries from invalidation.

6.1.2 TLB Entry Maintenance Features

The TLB entries must be loaded and maintained by the system software; this includes performing any required table search operations in memory. The e500mc provides support for maintaining TLB entries in software with the resources shown in Table 6-1. Section 6.5, "TLB Entry Maintenance—Details," describes hardware assistance features.

Table 6-1. TLB Maintenance Programming Model

Features		Description	Section/Page
TLB Instructions	**tlbre**	TLB Read Entry instruction	6.4.1/6-15
	tlbwe	TLB Write Entry instruction	6.4.2/6-15
	tlbsx rA, rB (preferred form: **tlbsx 0,** rB)	TLB Search for entry instruction	6.4.3/6-16
	tlbilx	TLB Invalidate Local instruction	6.4.4/6-17
	tlbivax rA, rB	TLB Invalidate instruction	6.4.5/6-17
	tlbsync	TLB synchronize invalidations	6.4.6/6-18
Registers	PID	Process ID register	Table 6-5
	MMUCSR0	MMU control and status register	
	MMUCFG	MMU configuration register	
	TLB0CFG–TLB1CFG	TLB configuration registers	
	MAS0–MAS8	MMU assist registers	
	(G)DEAR	(Guest)Data exception address register	
	(G)ESR	(Guest)Exception syndrome register	
Exceptions/ Interrupts	Instruction TLB miss	Causes instruction TLB error interrupt	4.9.15/4-33
	Data TLB miss	Causes data TLB error interrupt	4.9.14/4-32
	Instruction permissions violation	Causes ISI interrupt	4.9.5/4-23
	Data permissions violation	Causes DSI interrupt	4.9.4/4-21

6.2 Effective-to-Real Address Translation

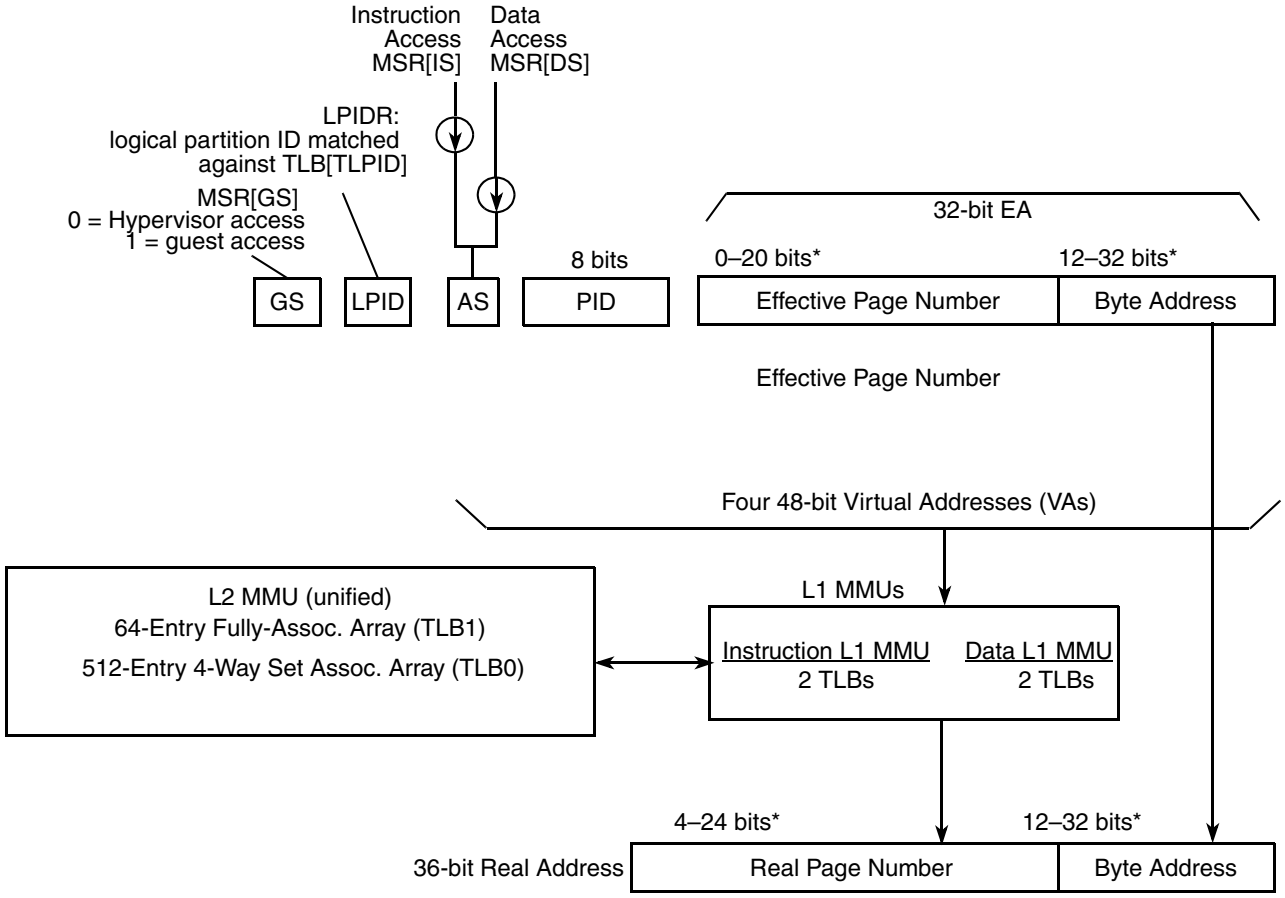
This section discusses effective-to-real address translation.

6.2.1 Address Translation

The fetch and load/store units generate 32-bit effective addresses. The MMU translates these addresses to 36-bit real addresses (which are used for memory accesses) using an interim virtual address. In multicore implementations, such as the e500mc, the virtual address is formed by concatenating MSR[GS] || LPIDR || MSR[IS|AS] || PID || EA, as shown in Figure 6-1.

The appropriate L1 MMU (instruction or data) is checked for a matching address translation. The instruction L1 MMU and data L1 MMU operate independently and can be accessed in parallel, so that hits for instruction accesses and data accesses can occur in the same clock. If an L1 MMU misses, the request for translation is forwarded to the unified (instruction and data) L2 MMU. If found, the contents of the TLB entry are concatenated with the page offset to obtain the physical address of the requested access. On

misses in the L1 MMU which hit in the L2 MMU, the L1 TLB entries are replaced from their L2 TLB counterparts using a true LRU algorithm.



* Number of bits depends on page size: 4 Kbytes–4 Gbytes

Figure 6-1. Effective-to-Real Address Translation Flow in e500mc

6.2.2 Address Translation Using External PID Addressing

External PID addressing provides an efficient way for system software to move data and perform cache operations across disjunct address spaces. On the e500mc, this functionality includes the following external PID versions of standard load, store, and cache instructions:

- Load-type instructions: **lbepx**, **lhepx**, **lwepx**, **ldepx**, **dcbtep**, **dcbtstep**, **dcbfep**, **dcbstep**, **icbiep**, and **lfdepx**
- Store-type instructions: **stbepx**, **sthepx**, **stwepx**, **stdepx**, **dcbzep**, **dcbzlep**, and **stfdepx**

Memory translation is performed by substituting the values configured in the external PID load/store control registers (EPLC and EPSC):

- External load context PR (EPR) replaces MSR[PR] for permissions checking.

- The following fields replace the standard values shown in Figure 6-1 to form a virtual address (as shown in Figure 6-2):
  - External guest state (EGS) replaces MSR[GS] in forming the virtual address and is compared against TLB[TGS] during translation. EGS is writable only in hypervisor state.
  - External logical partition ID (ELPID) replaces LPIDR and is compared against TLB[TLPID]. ELPID is writable only in hypervisor state.
  - External load context AS (EAS) replaces MSR[DS] and is compared against TLB[TS].
  - External load context process ID (EPID) replaces PID and is compared against TLB[TID].

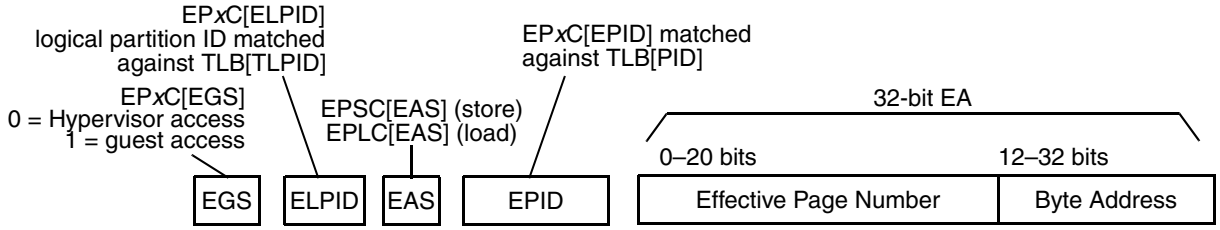


Figure 6-2. Forming a Virtual Address Using External PID

6.2.3 Variable-Sized Pages

The following page sizes are supported by the fully-associative TLBs that support variable-sized pages (VSPs).

- 4 Kbyte
- 16 Kbyte
- 64 Kbyte
- 256 Kbyte
- 1 Mbyte
- 4 Mbyte
- 16 Mbyte
- 64 Mbyte
- 256 Mbyte
- 1 Gbyte
- 4 Gbyte

6.2.3.1 Checking for TLB Entry Hit

Figure 6-3 shows the compare function used to check the MMU structures for a hit for a virtual address that corresponds to an instruction or data access.

A hit to multiple matching TLB entries is considered a programming error. If this occurs, the TLB generates an invalid address and TLB entries may be corrupted and a machine check or error report interrupt is generated if HID0[EN_L2MMU_MHD] is set. If HID0[EN_L2MMU_MHD] is not set when the error occurs the resulting translation is undefined.

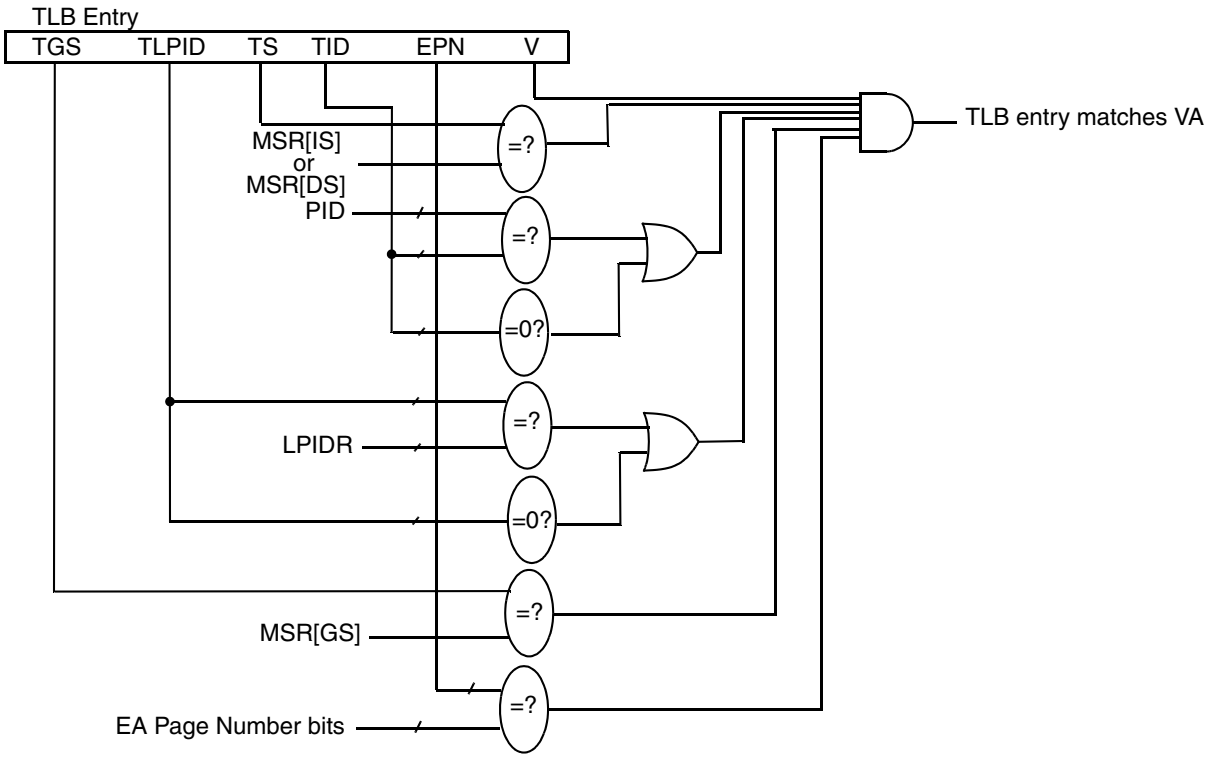


Figure 6-3. Virtual Address and TLB-Entry Compare Process

6.2.4 Checking for Access Permissions

When a TLB entry matches with a virtual address of an access, the permission bits of the TLB entry are compared with attribute information of the access (read/write, execute/data, user/supervisor) to see if the access is allowed to that page. The checking of permissions on the e500mc functions as described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

6.3 Translation Lookaside Buffers (TLBs)

To maximize address translation performance and to provide ample flexibility for the operating system, the e500mc implements six TLB arrays. Figure 6-4 contains a more detailed description of the 2-level structure. Note that for an instruction access, both the I-L1VSP and the I-L1TLB4K are checked in parallel for a TLB hit. Similarly, for a data access, both the D-L1VSP and the D-L1TLB4K are checked in parallel for a TLB hit. The instruction L1 MMU and data L1 MMU operate independently and can be accessed in parallel, so that hits for instruction accesses and data accesses can occur in the same clock. This figure shows the 36-bit real addresses and the 4-way set associative TLB0 used in the e500mc.

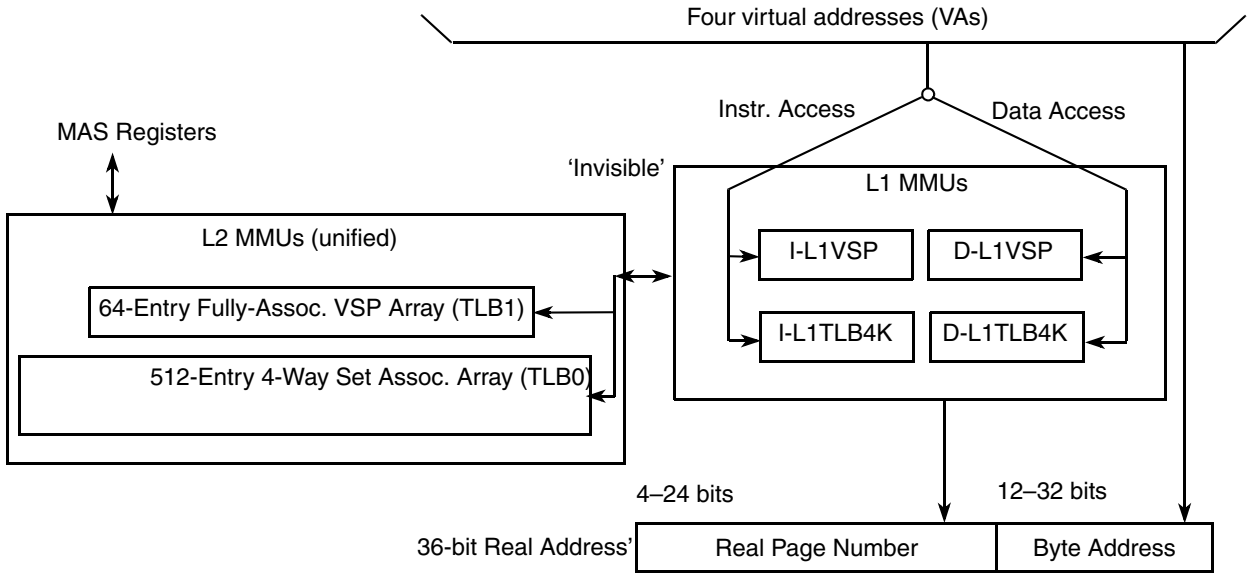


Figure 6-4. Two-Level MMU Structure

Additionally, Figure 6-4 shows that when the L2 MMU is checked for a TLB entry, both TLB1 and TLB0 are checked in parallel. It also identifies the L1 MMUs as invisible to the programming model (not accessible to the operating system); they are managed completely by the hardware as inclusive caches of the corresponding L2 MMU TLB entries. Conversely, the L2 MMU is managed by the TLB instructions by way of the MAS registers.

A hit to multiple TLB entries in the L1 MMU (even if they are in separate arrays) is considered to be a programming error. This is also the case if an access results in a hit to multiple TLB entries in the L2 MMU.

Table 6-2 lists the various TLBs and describes their characteristics.

Table 6-2. Index of TLBs

Location	Name	Page Sizes Supported	Associativity	Number of TLB Entries	Translations	Filled by
Instruction L1 MMU	I-L1VSP	11 page sizes [1]	Fully associative	8	Instruction	TLB1 hit
	I-L1TLB4K	4 Kbyte	4-way	64	Instruction	TLB0 hit
Data L1 MMU	D-L1VSP	11 page sizes [1]	Fully associative	8	Data	TLB1 hit
	D-L1TLB4K	4 Kbyte	4-way	64	Data	TLB0 hit
L2 MMU	TLB1	11 page sizes [1]	Fully associative	64	Unified (I and D)	**tlbwe**
	TLB0	4 Kbyte	4-way	512	Unified (I and D)	**tlbwe**

[1] See Section 6.2.3, "Variable-Sized Pages," for supported page sizes.

6.3.1 L1 TLB Arrays

As shown in Figure 6-1, there are two level 1 (L1) MMUs. As shown in Figure 6-4 and Table 6-2, the instruction and data L1 MMUs each implement a 8-entry, fully associative L1VSP array and a 64-entry, 4-way set associative L1TLB4K array, comprising the following L1 MMU arrays:

- Instruction L1VSP—8-entry, fully-associative
- Instruction L1TLB4K—64-entry, 4-way set-associative
- Data L1VSP—8-entry, fully associative
- Data L1TLB4K—64-entry, 4-way set-associative

As their names imply, L1TLB4K arrays support fixed, 4-Kbyte pages and L1VSP arrays support eleven page sizes. To perform a lookup for instruction accesses, both L1TLB4K and L1VSP TLBs in the instruction MMU are searched in parallel for the matching entry. Similarly, for data accesses, both L1TLB4K and L1VSP TLBs in the data MMU are searched in parallel for the matching entry. The contents of a matching entry are concatenated with the page offset of the original EA; the bit range that is translated is determined by the page size. The result constitutes the real (physical) address for the access.

L1TLB4K TLB entries are replaced based on a true LRU algorithm. The L1VSP entries are also replaced based on a true LRU replacement algorithm. The LRU bits are updated each time a TLB entry is accessed for translation. However, there are other speculative accesses performed to the L1 MMUs that cause the LRU bits to be updated. The performance of the L1 MMUs is high, even though it is not possible to predict exactly which entry is the next to be replaced.

Figure 6-5 shows the organization of the L1 TLBs in both the instruction and data L1 MMUs.

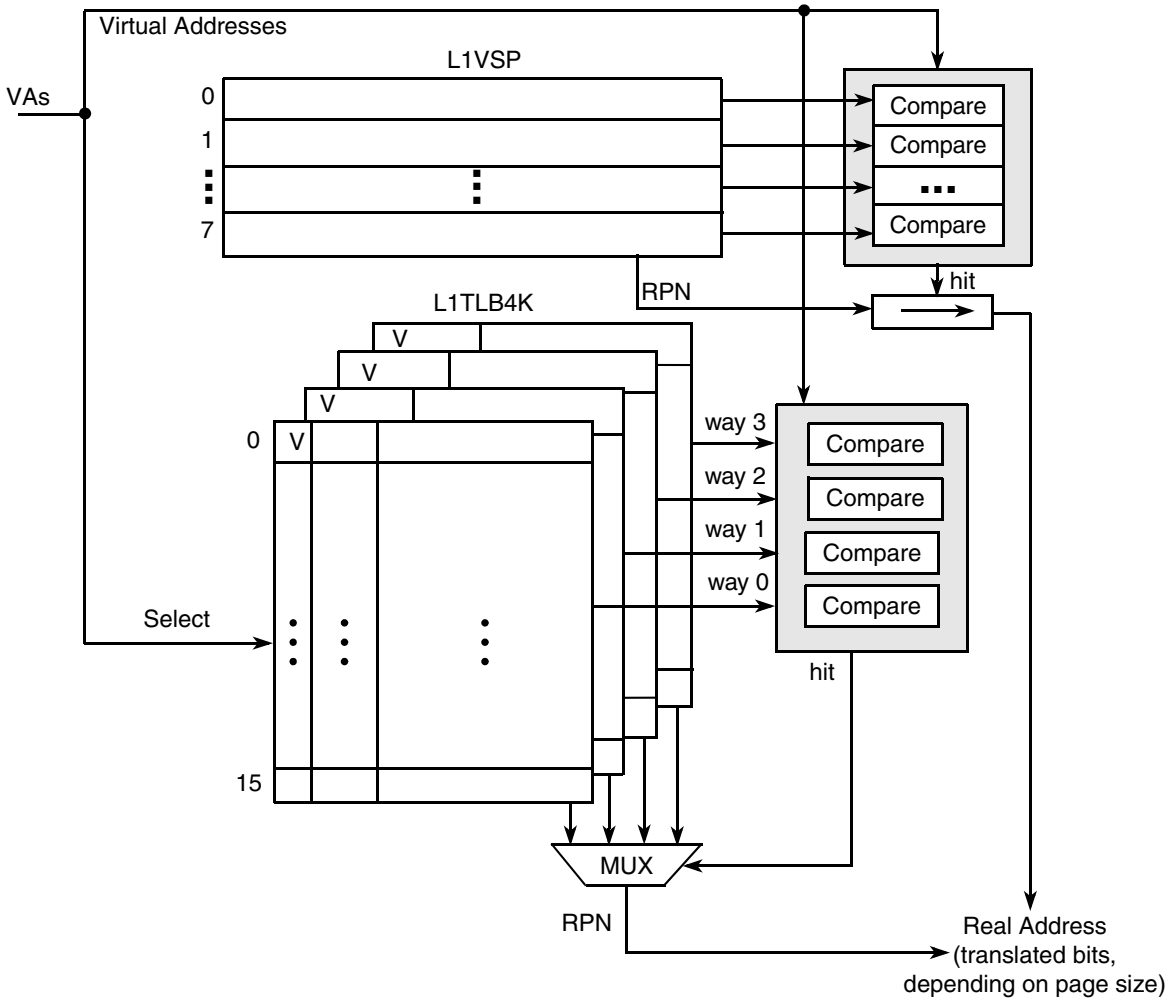


Figure 6-5. L1 MMU TLB Organization

6.3.2 L2 TLB Arrays

The level 1 MMUs are backed up by a unified level 2 MMU that translates both instruction and data addresses. Like each L1 MMU, the L2 MMU consists of two TLB arrays:

- TLB1: a 64-entry, fully associative array that supports eleven page sizes.
- TLB0: 512-entry, 4-way set associative array that supports only 4-Kbyte page sizes.

Figure 6-6 shows the L2 TLBs.

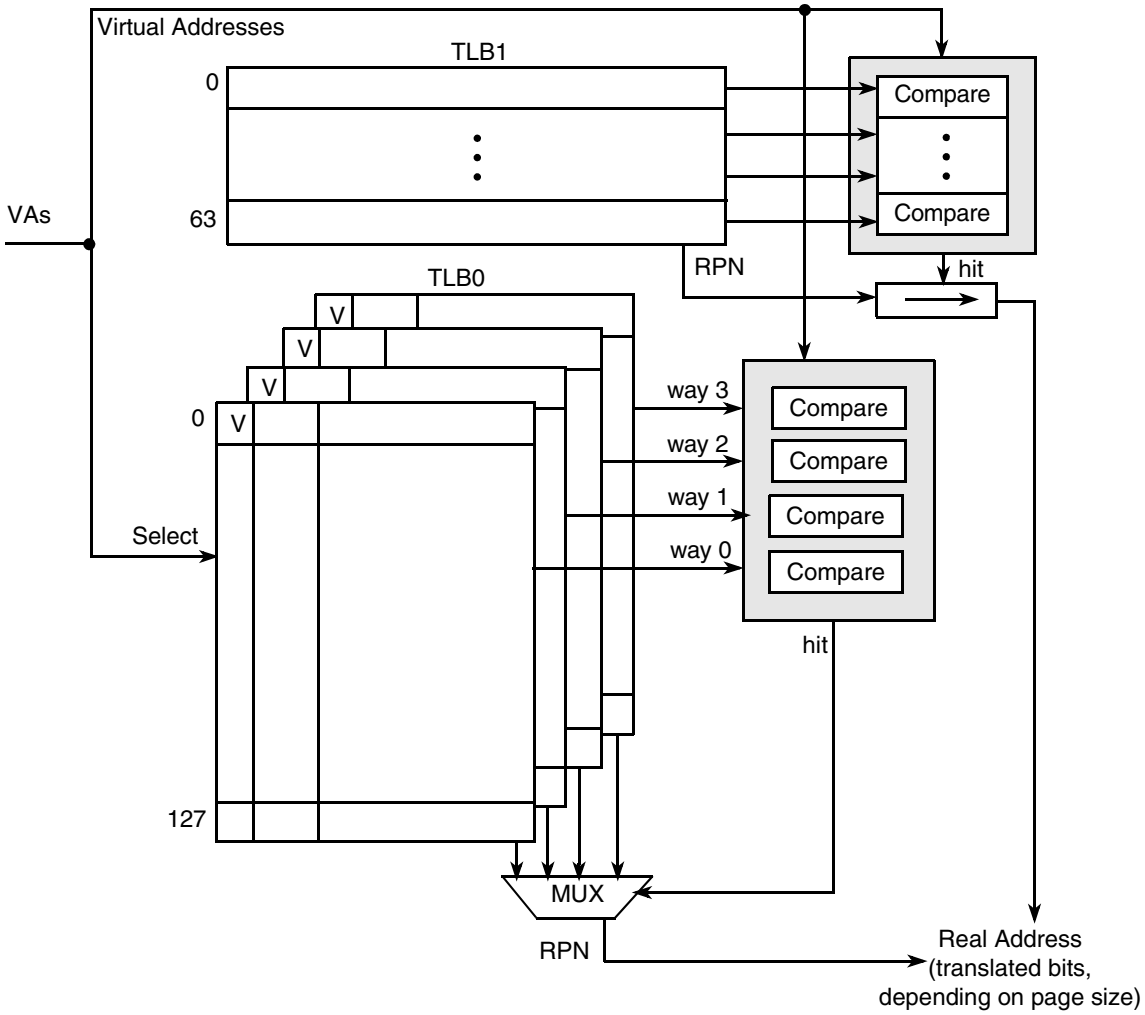


Figure 6-6. L2 MMU TLB Organization

6.3.2.1 IPROT Invalidation Protection in TLB1

TLB1 entries with IPROT set can never be invalidated by a **tlbivax** or **tlbilx** instruction executed by this processor (even when the INV_ALL command is indicated), by a **tlbivax** instruction executed by another processor, or by a flash invalidate initiated by writing to the MMUCSR0. IPROT can be used to protect critical code and data such as interrupt vectors/handlers in order to guarantee that the instruction fetch of those vectors never takes a TLB miss exception. Entries with IPROT set can be invalidated only by writing a 0 to the valid bit of the entry (by using the MAS registers and executing the **tlbwe** instruction).

Only TLB entries in TLB1 can be protected from invalidation; entries in TLB0 cannot because they do not implement IPROT.

Invalidation operations are guaranteed to invalidate the entry that translates the address specified in the operand of the **tlbivax** or **tlbilx** instruction. Other entries may also be invalidated by this operation if they are not protected with IPROT. A precise invalidation can be performed by writing a 0 to the valid bit of a

TLB entry. Note that successful invalidation operations in the L2 MMU also invalidate matching entries in the L1 MMU.

6.3.2.2 Replacement Algorithms for L2 MMU Entries

The replacement algorithm for TLB1 must be implemented completely by the system software. Thus, when an entry in TLB1 is to be replaced, the software selects which entry to replace and writes the entry number to MAS0[ESEL] before executing a **tlbwe** instruction.

TLB0 entry replacement is also implemented by software. To assist the software with TLB0 replacement, the core provides a hint that can be used for implementing a round-robin replacement algorithm. The hint is supplied in the appropriate MAS register fields when certain exceptions occur or a **tlbsx** instruction finds a valid entry. The only parameter required to select the entry to replace is the way select value for the new entry. (The entry within the way is selected by EA[45–51].) The mechanism for the round-robin replacement uses the following fields:

- TLB0[NV]—the next victim field within TLB0. The next victim field is value which points to a way in the set which should be used as the next victim if a new TLB entry is to be allocated. There is one next victim value for each set in TLB0.
- MAS0[NV]—the next victim field of MAS0
- MAS0[ESEL]—selects the way to be replaced on **tlbwe**

Table 6-6 describes MAS register updates on various exception conditions.

Note that the system software can load any value into MAS0[ESEL] and MAS0[NV] prior to execution of **tlbwe**, effectively overwriting this round robin replacement algorithm. In this case, the value written by software into MAS0[NV] is used as the next TLB0[NV] value on a TLB miss.

Also, note that the MAS0[NV] value is indeterminate after any TLB entry invalidate operation (including a flash invalidate). To know its value after an invalidate operation, MAS0[NV] must be read explicitly.

6.3.2.2.1 Round-Robin Replacement for TLB0

The core has a 4-way set associative TLB0, and so fully implements the round-robin scheme with a simple 2-bit counter that increments the 2-bit value of NV from the selected set of TLB0 entries on each TLB miss and loads the incremented value into MAS0[NV] for use by the next **tlbwe** instruction. Set selection is performed using bits from the EA that caused the TLB miss.

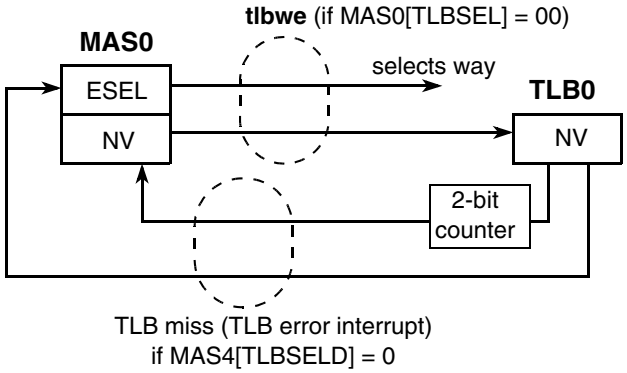


Figure 6-7. Round Robin Replacement for TLB0

When **tlbwe** executes, MAS0[ESEL] selects way 0, 1, 2, or 3 of TLB0 to be loaded, and if MAS0[TLBSEL] = 0 (selecting TLB0), TLB0[NV] is loaded with the MAS0[NV] value. When a TLB miss exception causes a TLB error interrupt and if MAS4[TLBSELD] = 0, the hardware automatically loads the current value of TLB0[NV] for the selected set into MAS0[ESEL] and the incremented value of TLB0[NV] for the selected set into MAS0[NV]. This sets up MAS0 such that if those values are not overwritten, the next way is selected on the next execution of **tlbwe**.

6.3.3 Consistency Between L1 and L2 TLBs

The contents of the L1 TLBs are always a proper subset of the TLB entries currently resident in the L2 MMU. They serve to improve performance because they have a faster access time than the larger L2 TLBs. The relationships between the six TLBs are shown in Figure 6-8.

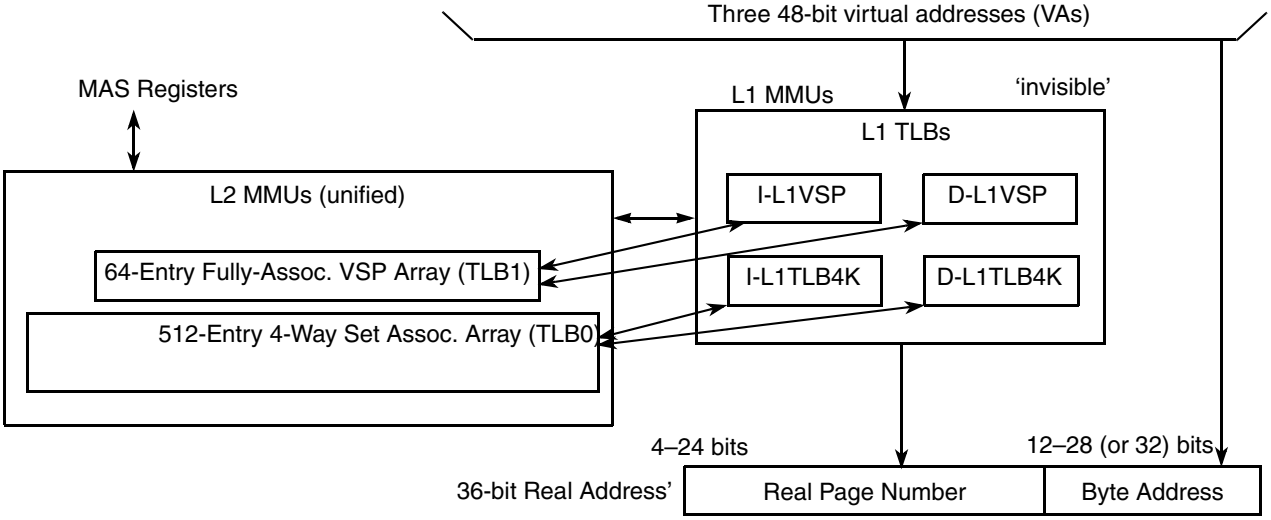


Figure 6-8. L1 MMU TLB Relationships with L2 TLBs

On an L1 MMU miss, L1 MMU array entries are automatically reloaded using entries from their level 2 array equivalent. For example, if the L1 data MMU misses but there is a hit for a virtual address in TLB1, the matching entry is automatically loaded into the data L1VSP array. Likewise, if the L1 data MMU

misses, but there is a hit for the access in TLB0, the matching entry is automatically loaded into the data L1TLB4K array.

NOTE

Writing to LPIDR or PID causes all L1 entries to be invalidated.

Writing to EPLC or EPSC causes all data-side L1 entries to be invalidated.

Any **tlbilx** with T = 0 or 1 that clears an L2MMU TLB0/1 entry causes all L1 TLBs to be invalidated.

NOTE

When any L2 TLB entry is written or invalidated (through any invalidation mechanism), the corresponding entries in any L1 TLB will also be invalidated. Changing PID, LPID, EPLC, or EPSC may cause all L1 entries to be invalidated.

6.3.4 The G Bit (of WIMGE)

The Guarded (G) bit provides protection from bus accesses due to speculative and faultable instruction execution. A speculative access is defined as an access caused by an instruction that is downstream from an unresolved branch. A faultable access is defined as an access that could be cancelled due to an exception on an uncompleted instruction.

On the e500mc, if the page for this type of access is marked with G = 0 (unguarded), this type of access may be issued to the CoreNet interface regardless of the completion status of other instructions. If G = 1, the access stalls if it misses in the cache until the access is known to be required by the program execution model; that is, all previous instructions will have completed without exception and no asynchronous interrupts occur between the time that the access is issued to CoreNet and the time that CoreNet transaction request completes. For reads, this requires that the data be returned and the instruction is retired. For writes, the instruction retires when the write transaction is committed to be sent to the CoreNet.

An access with G = 1 attribute that has gone out to the CoreNet interface is guaranteed to complete. That is, after the address tenure is acknowledged on the CoreNet, the access completes, even if an asynchronous interrupt is pending.

Note that G = 1 misaligned accesses are not guaranteed to be accessed only once. For example a load address that crosses a page boundary where one of the parts encounters a TLB miss and the other does not, the non-TLB miss part may occur, and the TLB miss exception may be taken. When software loads a valid TLB entry for the part that missed, the instruction will be returned to and re-execute performing the load operation again to both parts of the misaligned accesses.

The G bit is ignored for instruction fetches, and instructions are fetched speculatively from guarded pages. To prevent speculative fetches from guarded pages without instructions, the page should be also designated as no-execute (with the UX/SX page permission bits cleared).

6.3.5 TLB Entry Field Definitions

Table 6-3 summarizes the fields of e500mc TLB entries. These fields are defined by the architecture and described in detail in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

Table 6-3. TLB Entry Bit Definitions

Field	Comments
V	Valid bit for entry
TS	Translation address space (compared with AS bit of the current access. For external PID accesses, TS is compared with EPLC[EAS] or EPSC[EAS].
TID[0–7]	Translation ID (compared with PID). For external PID accesses, TID is compared with EPLC[EPID] or EPSC[EPID].
EPN[0–19]	Effective page number (compared with EA[32–51] for 4-Kbyte pages)
RPN[0–23]	Real page number: Translated address RA[28–51] for 4-Kbyte pages
SIZE[0–3]	Encoded page size. Values not shown are reserved 0000 Reserved 0001 4 Kbytes 0010 16 Kbytes 0011 64 Kbytes 0100 256 Kbytes 0101 1 Mbyte 0110 4 Mbytes 0111 16 Mbytes 1000 64 Mbytes 1001 256 Mbytes 1010 1 Gbyte 1011 4 Gbytes
PERMIS[0–5]	Supervisor execute, write, and read permission bits, and user execute, write, and read permission bits.
WIMGE	Memory/cache attributes (write-through, cache-inhibit, memory coherence required, guarded, endian)
X0, X1	Extra system attribute bits
U0–U3	User attribute bits—used only by software. These bits exist in the L2 MMU TLBs only (TLB1 and TLB0)
IPROT	Invalidation protection (exists in TLB1 only)
TGS	Translation guest space.
VF	Virtualization fault. If set, a DSI occurs on data accesses to this page, regardless of the setting of the permission bits. 0 Data accesses translated by this TLB entry occur normally. 1 Data accesses translated by this TLB entry always cause a Data Storage Interrupt that is directed to the hypervisor state.
TLPID	Translation logical partition ID.

6.4 TLB Instructions—Implementation

As described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*, TLBs are accessed indirectly through MMU assist (MAS) registers. Software can write and read the MAS registers with **mtspr** and **mfspr**. MAS registers contain information related to reading and writing a given entry within the TLBs. For example, data is read from the TLBs into the MAS registers with a TLB Read Entry (**tlbre**) instruction, and data is written to the TLBs from the MAS registers with a TLB Write Entry (**tlbwe**) instruction.

The **tlbre**, **tlbwe**, **tlbsx**, **tlbivax**, **tlbsync**, and **tlbilx** instructions are summarized in this section.

6.4.1 TLB Read Entry (tlbre) Instruction

TLB entries can be read by executing **tlbre** instructions. When **tlbre** executes, MAS registers are used to index a specific TLB entry and upon completion of the **tlbre**, they contain the contents of the indexed TLB entry.

Selection of the TLB entry to read is performed by setting MAS0[TLBSEL,ESEL] and MAS2[EPN] to indicate the entry to read. MAS0[TLBSEL] selects which TLB the entry should be read from and MAS2[EPN] selects the set of entries from which MAS0[ESEL] selects an entry. For fully associative TLBs, MAS2[EPN] is not required because MAS0[ESEL] fully identifies the TLB entry.

The selected entry is then used to update the following MAS register fields: V, IPROT, TID, TS, TSIZE, EPN, WIMGE, RPN, U0—U3, X0, X1, TLPID, TGS, VF, and permission bits. If the TLB array supports NV, it is used to update the NV field in the MAS registers, otherwise the contents of NV field are undefined. The update of MAS registers as a result of a **tlbre** instruction is summarized in Table 6-6.

tlbre can only be executed by the hypervisor. If the guest supervisor attempts a **tlbre**, an embedded hypervisor privilege interrupt occurs.

Note that architecturally, if the instruction specifies a TLB entry that is not found, the results placed in MAS0–MAS3, MAS5, MAS7 and MAS8 are undefined. However, for e500mc, the TLBSEL, ESEL and EPN fields always index to an existing L2 TLB entry and that indexed entry is read. Note that EPN bits are only used to index into TLB0. In the case of TLB1, the EPN field is unused for **tlbre**. See the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* for information at the architecture level.

6.4.1.1 Reading TLB1 and TLB0 Array Entries

TLB entries are read by first writing the entry-identifying information into MAS0 (and MAS2 for TLB0), using **mtspr** and then executing the **tlbre** instruction.

To read a TLB1 entry, MAS0[TLBSEL] must be = 01 and MAS0[ESEL] must point to the desired entry. To read a TLB0 entry, MAS0[TLBSEL] must be = 00, MAS0[ESEL] must point to the desired way, and EPN[45–51] in MAS2 must be loaded with the desired index.

6.4.2 TLB Write Entry (tlbwe)

TLB entries can be written by executing **tlbwe** instructions. When **tlbwe** executes, MAS registers are used to index a specific TLB entry and contain the contents to be written to the indexed TLB entry. Upon completion of **tlbwe**, the TLB entry contents of the MAS registers are written to the indexed TLB entry.

The TLB entry to write is determined by the MAS0[TLBSEL,ESEL] and MAS2[EPN] values. MAS0[TLBSEL] selects which TLB the entry should be written to; MAS2[EPN] selects the set of entries from which MAS0[ESEL] selects an entry. For fully associative TLBs, MAS2[EPN] is not used to identify a TLB entry since the value in MAS0[ESEL] fully identifies the TLB entry. The selected entry is then

written with following MAS fields: V, IPROT, TID, TS, TSIZE, EPN, WIMGE, RPN, U0—U3, X0, X1, TLPID, TGS, VF, and permission bits. If the TLB array supports NV, it is written with the NV value.

The effects of updating the TLB entry are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation. Writing a TLB entry that is used by the programming model prior to a context synchronizing operation produces undefined behavior.

e500mc does not provide a logical to real translation (LRAT) mechanism so **tlbwe** can only be executed by the hypervisor regardless of the state of EPCR[DGTMI]. If the guest supervisor attempts a **tlbwe**, an embedded hypervisor privilege interrupt occurs.

Note that architecturally, if the instruction specifies a TLB entry that is not found, the results are undefined. However, for e500mc, the TLBSEL, ESEL and EPN fields always index to an existing L2 TLB entry and that indexed entry is written. Note that EPN bits are only used to index into TLB0. In the case of TLB1, the EPN field is unused for **tlbre**. See the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* for information at the architecture level.

6.4.2.1 Writing to the TLB1 Array

TLB1 can be written by first writing the necessary information into MAS0–MAS3, MAS5, MAS7, and MAS8 using **mtspr** and then executing the **tlbwe** instruction. To write an entry into TLB1, MAS0[TLBSEL] must be equal to 1, and MAS0[ESEL] must point to the desired entry. When the **tlbwe** instruction is executed, the TLB entry information stored in MAS0–MAS3, MAS5, MAS7, and MAS8 is written into the selected TLB entry in the TLB1 array.

6.4.2.2 Writing to the TLB0 Array

TLB0 can be written by first writing the necessary information into MAS0–MAS3, MAS5, MAS7, and MAS8 using **mtspr** and then executing the **tlbwe** instruction. To write an entry into TLB0, MAS0[TLBSEL] must be equal to zero, MAS0[ESEL] must point to the desired way, and EPN[45–51] in MAS2 must be loaded with the desired index. When **tlbwe** executes, the TLB entry information in MAS0–MAS3, MAS5, MAS7, and MAS8 is written into the selected entry in TLB0.

6.4.3 TLB Search (tlbsx)—Searching TLB1 and TLB0 Arrays

Software can search the MMU by using **tlbsx**, which uses GS, LPIDR, and PID values and an AS value from MAS5 and MAS6 instead of from LPIDR, PID, and the MSR. This allows software to search address spaces that differ from the current address space defined by the GS, AS, LPID and PID registers. This is useful for TLB fault handling.

To search for a TLB, software loads MAS5[SGS] with a GS value, MAS5[SLPID] with an LPID value, MAS6[SPID] with a PID value, and MAS6[SAS] with an AS value to search for. Software then executes **tlbsx** specifying the EA to search for.

If a matching, valid TLB entry is found, the MAS registers are loaded with the information from that TLB entry as if the TLB entry were read from by executing **tlbre**. If the search is successful, MAS1[V] is set to 1. If the search is unsuccessful, MAS1[V] is set to 0.

Executing **tlbsx** updates the MAS registers conditionally based on the success or failure of a TLB lookup in the L2 MMU. The values placed into MAS registers differ, depending on whether the search is successful. Section 6.7.1, "MAS Register Updates," describes how MAS registers are updated.

NOTE

Note that **r**A = 0 is the preferred form for **tlbsx** and that some Freescale implementations, including the e500mc, take an illegal instruction exception program interrupt if **r**A!=0.

6.4.4 TLB Invalidate Local Indexed (tlbilx) Instruction

Zero, one, or more TLB entries can be invalidated through the execution of a **tlbilx** instruction. Note that guest supervisor software can execute **tlbilx**. The behavior depends on the T operand, as follows:

- If T = 0, all TLB entries for which $entry_{TLPID}$ = MAS5[SLPID] are invalidated.
- If T = 1, all TLB entries for which $entry_{TLPID}$ = MAS5[SLPID] and $entry_{TID}$ = MAS6[SPID] are invalidated.
- If T = 3, all TLB entries for which $entry_{TLPID}$ = MAS5[SLPID] and $entry_{TID}$ = MAS6[SPID] and ($entry_{EPN}$&*m*) = ($EA_{32:51}$&*m*), where *m* is an appropriate mask based on page size, are invalidated.

If an entry selected for invalidation has IPROT set, that entry is not invalidated.

Unlike **tlbivax**, TLB entries are only invalidated on the processor which executes **tlbilx**.

NOTES

tlbilx is the preferred way of performing TLB invalidations, especially for operating systems running as a guest to the hypervisor since the invalidations are partitioned and do not require hypervisor privilege.

The preferred form of **tlbilx** has **r**A = 0. Forms where **r**A != 0 takes an illegal instruction exception on some Freescale processors.

Hypervisor should always set MAS5[SLPID] to LPIDR when dispatching to a guest.

Executing **tlbilx** with T = 0 or T = 1 may take many cycles to perform. Software should only issue these operations when an LPID or a PID value is reused or taken out of use.

6.4.5 TLB Invalidate (tlbivax) Instruction

The **tlbivax** instruction invalidates any TLB entry that corresponds to the virtual addresses calculated by the instruction. This includes entries in the executing processor and TLBs on other processors and devices throughout the coherence domain of the processor executing **tlbivax**.

EA[60] selects the TLB array to which the invalidation is to occur. EA[59] is ignored, but software should set it to 0.

If EA[61] (IA field) is set, all TLB entries in the designated TLB array are invalidated, regardless of partition, except for TLB entries with the IPROT attribute set to 1.

If EA[61] is 0, invalidation is partitioned and only TLB entries whose EPN field matches the EA[0:51] and whose TLPID field matches the MAS5[SLPID] and whose TGS field matches MAS5[SGS] of the processor executing the **tlbivax** instruction are invalidated. Other TLB entries may be invalidated by the implementation, but in no case will any TLB entries (including ones that match the invalidation criteria) with the IPROT attribute set be invalidated.

Software should also set MAS6[SPID] and MAS6[SAS] to further identify the entry which is to be invalidated although the e500mc does not use these values in the comparison (and will thus invalidate entries regardless of the content of their TID and TS fields). If portability of software to future implementations is desired, software should not assume that any TLB entry will be invalidated except the entry corresponding to the EA, MAS5[SLPID], MAS5[SGS], MAS6[SPID], and MAS6[SAS] as future implementations may invalidate to the stricter MAS6[SPID] and MAS6[SAS] criteria.

Because the virtual address can be much larger than the physical address, a subset of the full virtual address is broadcast that fits within the space of the implemented physical addressing mode.

Because the **tlbivax** does not compare PID or AS values, one **tlbivax** can invalidate multiple entries in a targeted TLB. A **tlbivax** targeting TLB0 can invalidate up to all four ways, and up to all four ways within an L1TLB4K index. A **tlbivax** targeting TLB1 can invalidate up to all 64 entries in the array, or up to all 8 entries of the L1VSPs (instruction and data). Section 6.3.2.1, "IPROT Invalidation Protection in TLB1," describes how to protect TLB1 entries from this type of invalidation.

The **tlbivax** instruction invalidates all matching entries in the instruction and data L1 TLBs simultaneously. Also, the core always snoops TLB invalidate transactions and invalidates matching TLB entries accordingly.

NOTE

Note that **r**A = 0 is the preferred form for **tlbivax** and that some Freescale implementations take an illegal instruction exception program interrupt if **r**A!=0.

6.4.5.1 TLB Selection and Invalidate All Address tlbivax Encodings

Because only a subset of the virtual address is broadcast, extra information about the targeted TLB entries is encoded in two of the lower EA bits calculated by **tlbivax**.

- Bit 60 is interpreted as the TLBSEL field, which indicates whether TLB1 or TLB0 is targeted. TLBSEL prevents unwanted invalidations of large pages in TLB1 when a **tlbivax** targets TLB0.
- Bit 61 is interpreted as the INV_ALL command. Setting bit 61 it indicates that the operation should invalidate all entries of either TLB1 or TLB0 as indicated by the TLBSEL field, and invalidate all corresponding L1 TLB entries.

6.4.6 TLB Synchronize (tlbsync) Instruction

The **tlbsync** instruction causes a TLBSYNC transaction on the CoreNet interface. This instruction effectively synchronizes the invalidation of TLB entries; **tlbsync** does not complete until all memory accesses caused by instructions issued before an earlier **tlbivax** instruction have completed.

NOTE

Software must ensure that only one **tlbsync** operation is active at a given time. A second **tlbsync** issued (from any core in the coherence domain) before the first one has completed, can cause processors to hang. Software should make sure the **tlbsync** and its associated synchronization is contained with a mutual exclusion lock that all processor must acquire before executing **tlbsync**.

6.5 TLB Entry Maintenance—Details

TLB entries must be loaded and maintained by the system software, including performing the required table search operations in memory. The e500mc provides some hardware assistance for these software tasks. Note that the system software cannot directly access the L1 TLBs, and the L1 TLBs are completely and automatically maintained in hardware as a subset of the contents of the L2 TLBs.

In addition to the resources described in Table 6-1, hardware assists TLB entries maintenance as follows:

- Automatic loading of MAS0–MAS2 and MAS6 based on the default values in MAS4 and other context when a TLB miss exceptions. This automatically generates most fields of the required TLB entry on a miss. Thus software should load MAS4 with likely values to be used in the event of a TLB miss.
- Automatic loading of the data exception address register (DEAR or GDEAR) with the EA of the load, store, or cache management instruction that caused an alignment, data TLB miss (data TLB error interrupt), or permissions violation (DSI interrupt).
- Automatic loading into SRR0 of the EA of the instruction that causes a TLB miss exception or a data storage interrupt.
- Automatic updates of the next victim (NV) field and MAS0[ESEL] fields for TLB0 entry replacement on TLB misses (TLB error interrupts); this occurs if TLBSELD = 0. See Section 6.3.2.2, “Replacement Algorithms for L2 MMU Entries.”
- When **tlbwe** executes, the information for the selected victim is read from the selected L2 TLB (TLB1 or TLB0). The victim’s EPN and TS are sent to both L1 MMUs to provide back-invalidation. Thus if the selected victim in the L2 MMU also resides in an L1 MMU, it is invalidated (or victimized) in the L1 MMU. This forces inclusion in the TLB hierarchy. Additionally, the new TLB entry contained in MAS0–MAS3, MAS7, and MAS8 is written into the selected TLB.

Note that although **tlbwe** loads an L2 TLB entry, it does not load an L1 TLB entry. L1 arrays are loaded with new entries (automatically by the hardware) only when an access misses in the L1 array, but hits in a corresponding L2 array.

See Section 6.7.1, “MAS Register Updates,” for a complete description of automatic fields loaded into the MAS registers on execution of TLB instructions and for various exception conditions.

The *EREF: A Programmer’s Reference Manual for Freescale Power Architecture® Processors* provides more information on some actions taken on MMU exceptions. The following subsections provide supplementary information that applies for the e500mc.

6.5.1 TLB Interrupt Routines

When the MMUs report an exception, all instructions dispatched before the exception are allowed to complete. The interrupt is acknowledged and MAS0–MAS2 and MAS6 are loaded as described in Section 6.7.1, "MAS Register Updates."

Most TLB miss, DSI, and ISI handlers must first save the values of enough GPRs for the handler's use. The handler should then perform an **mfcr** to copy the CR data into one of the GPRs. Before exiting the handler, an **mtcrf** must restore the CR before the original GPR data must be restored.

PID must also be restored (if modified) before exiting the handler. Note that PID register updates must be followed by an **isync**. This **isync** instruction must reside in an instruction page that is valid before the changes are made to the PID.

6.5.1.1 Permissions Violations (ISI, DSI) Interrupt Handlers

On a DSI or ISI, software must load the MAS registers appropriately if it wishes to update the TLB entry associated with the error. In general, software will ensure the appropriate PID and AS values are in MAS6 and can then execute a **tlbsx** using the value from SRR0 if it is an ISI or the value from (G)DEAR if it is a DSI.

6.6 TLB States after Reset

During reset, all L1 and L2 MMU TLB entries are flash invalidated. Then entry 0 of TLB1 is loaded with the values shown in Table 6-4. Note that only the valid bits for other TLB entries are cleared; other fields are not set to a known state so software must ensure that all fields of an entry are initialized appropriately through the MAS registers before it is used for translation.

NOTE

This default TLB entry translates the first instruction fetch out of reset (at effective address 0xFFFF_FFFC). This instruction should be a branch to the beginning of this page. Because this page is only 4 Kbytes, the initial code in this page needs to set up more valid TLB entries (and pages) so the program can branch into other pages for booting the operating system. In particular, the interrupt vector area and the pages that contain the interrupt handlers should be set up so exceptions can be handled early in the booting process.

Table 6-4. TLB1 Entry 0 Values after Reset

Field	Reset Value	Comments
V	1	Entry is valid
TS	0	Address space 0
TGS	0	Hypervisor address space
TID[0–7]	0	TID value for shared (global) page
TLPID	0	TLPID value for hypervisor page
TID[0–7]	0x00	TID value for shared (global) page

Table 6-4. TLB1 Entry 0 Values after Reset (continued)

Field	Reset Value	Comments
EPN[32–51]	0xFFFFF	Address of last 4-Kbyte page in address space
RPN[34–51]	0x3FFFF	Lower 18 bits of RPN
RPN[28–33]	SoC supplied	Upper 6 bits of real address space. These bits are supplied to the core from the SoC. See the reference manual for the integrated device.
SIZE[0–3]	0b0001	4-Kbyte page size
SX/SR/SW	0b111	Full supervisor mode access allowed
UX/UR/UW	0b000	No user mode access allowed
WIMGE	0b01000	Caching-inhibited, noncoherent, not guarded, big-endian
X0–X1	0b00	Reserved system attributes
U0–U3	0b0000	User attribute bits
IPROT	1	Page is protected from invalidation
VF	0	Page is not a virtualization page

6.7 MMU Registers

Table 6-5 provides cross-references to sections with more detailed bit descriptions for the e500mc MMU registers. The *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* lists the architecture definitions for these registers.

Table 6-5. Registers Used for MMU Functions

Registers	Section/Page
Process ID (PID)	2.16.2/2-50
Logical Process ID (LPIDR)	2.16.1/2-49
MMU control and status register (MMUCSR0)	2.16.3/2-50
MMU configuration register (MMUCFG)	2.16.4/2-51
TLB configuration registers (TLB0CFG–TLB1CFG)	2.16.5/2-51
MMU assist registers (MAS0–MAS8)	2.16.6/2-52
Data exception address register (DEAR/GDEAR)	2.9.2/2-17

6.7.1 MAS Register Updates

Table 6-6 summarizes how MAS register are updated by hardware for each stimulus. The table can be interpreted as follows:

- A field name refers to a MAS register field.
- PID, MSR, EPLC, and EPSC refer to their respective registers.
- EA refers to the effective address used for the memory access which caused a TLB error (miss).
- TLB0[NV] refers to the next victim value for the set selected by EA stored in TLB0.

- The TLB entry specified by TLBSEL and ESEL is referred to as TLB0 (if the value comes only from TLB0), TLB1 (if the value comes only from TLB1), or TLB if the value can come from the selected TLB array and the field is stored the same regardless of which array it is in.

Table 6-6. MMU Assist Register Field Updates

MAS Field	Inst TLB Error Data TLB Error	tlbsx Hit	tlbsx Miss	tlbre
MAS0				

TLBSEL	TLBSELD	if TLB0 hit
		  0
		else
		  1
			TLBSELD	—

ESEL	if TLBSELD = 0
	  0b0000 || TLB0[NV]
	else
	  0b000000
		if TLBSEL = 0
		  0b0000 || (way that hit)
		else
		  (entry that hit)
			if TLBSELD = 0
			  0b0000 || TLB0[NV]
			else
			  0b000000
				—

NV	if TLBSELD = 0
	  mod(TLB0[NV]+1,4)
	else
	  0
		if TLBSEL = 0
		  TLB0[NV]
		else
		  0
			if TLBSELD = 0
			  mod(TLB0[NV]+1,4)
			else
			  0
				if TLBSEL = 0
				  TLB0[NV]
				else
				  0

MAS1				

IPROT	0	If TLB1 hit
		  TLB1[IPROT]
		else
		  0
			0	If TLB1 hit
				  TLB1[IPROT]
				else
				  0

TID	if ext PID load
	  EPLC[EPID]
	elseif ext PID store
	  EPSC[EPID]
	else
	  PID
		TLB[TID]	SPID	TLB[TID]

TSIZE	TSIZED	if TLB1 hit
		  TLB1[TSIZE]
		else
		  1
			TSIZED	if TLB1 hit
				  TLB1[TSIZE]
				else
				  1

TS	if Data TLB Error
	  if ext PID load
	  EPLC[EAS]
	  elseif ext PID store
	  EPSC[EAS]
	  else
	  MSR[DS]
	else
	  MSR[IS]
		TLB[TS]	SAS	TLB[TS]

V	1	1	0	TLB[V]
MAS2				
WIMGE	WIMGED	TLB[WIMGE]	WIMGED	TLB[WIMGE]

Table 6-6. MMU Assist Register Field Updates (continued)

MAS Field	Inst TLB Error Data TLB Error	tlbsx Hit	tlbsx Miss	tlbre
X0, X1	X0D, X1D	TLB[X0, X1]	X0D, X1D	TLB[X0, X1]

EPN[32:51]	EA[32:51] of access	if TLBSEL = 0
		  TLB[EPN[32:44]] || EPN[45:51]
		else
		  TLB[EPN[32:51]]
			EA[32:51]	if TLBSEL = 0
				  TLB[EPN[32:44]] || EPN[45:51]
				else
				  TLB[EPN[32:51]]

MAS3				
UR,SR,UW,SW,UX,SX	Zeros	TLB[UR,SR,UW,SW,UX,SX]	Zeros	TLB[UR,SR,UW,SW,UX,SX]
U0–U3	Zeros	TLB[U0-U3]	Zeros	TLB[U0-U3]
RPN[32:51]	Zeros	TLB[RPN[32:51]]	Zeros	TLB[RPN[32:51]]
MAS4				
WIMGED	—	—	—	—
WIMGED, X0D,X1D, TIDSELD, TLBSELD, TSIZED	—	—	—	—
MAS5				
SGS	—	—	—	—
SLPID	—	—	—	—
MAS6				

SAS	if Data TLB Error
	  if ext PID load
	  EPLC[EAS]
	  elseif ext PID store
	  EPSC[EAS]
	  else
	  MSR[DS]
	else
	  MSR[IS]
		—	—	—

SPID	if ext PID load
	  EPLC[EPID]
	elseif ext PID store
	  EPSC[EPID]
	else
	  PID
		—	—	—

MAS7 (if HID0[EN_MAS7_UPDATE] = 1)				
RPN[28:31]	Zeros	TLB[RPN[28:31]]	Zeros	TLB[RPN[28:31]]
MAS8				
TGS	—	TLB[TGS]	—	TLB[TGS]

Table 6-6. MMU Assist Register Field Updates (continued)

MAS Field	Inst TLB Error Data TLB Error	tlbsx Hit	tlbsx Miss	tlbre
VF	—	TLB[VF]	—	TLB[VF]
TLPID	—	TLB[TLPID]	—	TLB[TLPID]

Chapter 7
Timer Facilities

This chapter describes specific implementation details of the e500mc implementation of architecture-defined timer facilities. These resources, which include the time base (TB), alternate time base (ATB), decrementer (DEC), fixed-interval timer (FIT), and watchdog timer, are described in detail in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

7.1 Timer Facilities

The TB, DEC, FIT, ATB, and watchdog timer provide timing functions for the system. All of these must be initialized during start-up.

- The TB provides a long-period counter driven by a frequency that is implementation dependent.
- The decrementer, a counter that is updated at the same rate as the TB, provides a means of signaling an exception after a specified amount of time has elapsed unless one of the following occurs:
  - DEC is altered by software in the interim
  - The TB update frequency changes

  The DEC is typically used as a general-purpose software timer.
- The clock source for the TB and the DEC is driven by the integrated device and is normally selectable to be a ratio of some integrated device clock frequency, or driven from a clock source external to the integrated device (that is, customer supplied). See the reference manual of the integrated device for details.
- The fixed-interval timer is essentially a selected bit of the TB, which provides a means of signaling an exception whenever the selected bit transitions from 0 to 1, in a repetitive fashion. The fixed-interval timer is typically used to trigger periodic system maintenance functions. Software may select any bit in the TB to serve as the fixed-interval timer.
- The ATB provides a 64-bit timer that cannot be written, which increments at an implementation dependent frequency. For the e500mc, the ATB frequency is the same as the core frequency which makes the ATB useful for measuring elapsed time in core clocks.
- The watchdog timer is also a selected bit of the TB, which provides a means of signalling a critical class exception whenever the selected bit transitions from 0 to 1. In addition, if software does not respond in time to the initial exception (by clearing the associated status bits in the TSR before the next expiration of the watchdog timer interval), then a watchdog timer-generated processor reset may result, if so enabled. The watchdog timer is typically used to provide a system error recovery function. Software may select any bit in the TB to serve as the watchdog timer.

The relationship of these timer facilities (except for the ATB) to each other is shown in Figure 7-1.

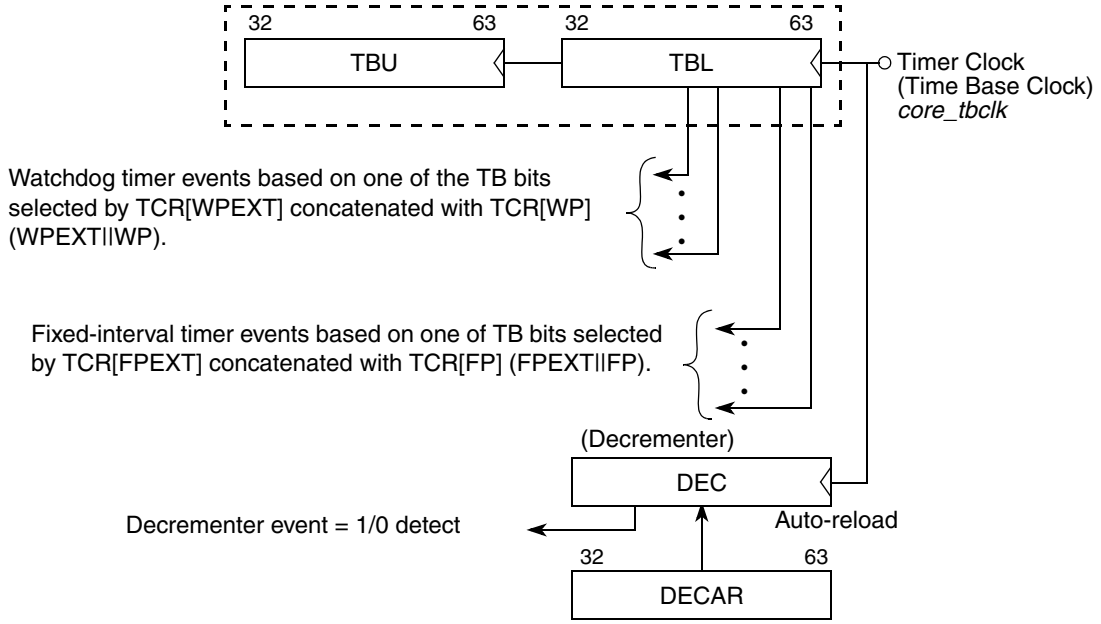


Figure 7-1. Relationship of Timer Facilities to Time Base

7.2 Timer Registers

This section describes registers used by the timer facilities.

- Timer control register (TCR). Provides control information for the on-chip timer of the core complex. The TCR controls decrementer, fixed-interval timer, and watchdog timer options. Section 2.8.1, "Timer Control Register (TCR)," describes the TCR in detail.
- Timer status register (TSR). Contains status on timer events and the most recent watchdog timer-initiated processor reset. Section 2.8.2, "Timer Status Register (TSR)," describes the TSR in detail.
- Decrementer register (DEC). DEC contents can be read into bits 32–63 of a GPR using **mfspr**, clearing bits 0–31. GPR contents can be written to the decrementer using **mtspr**. See Section 2.8.4, "Decrementer Register (DEC)," for more information.
- Decrementer auto-reload register (DECAR). Supports the auto-reload feature of the decrementer. The DECAR contents cannot be read. See Section 2.8.5, "Decrementer Auto-Reload Register (DECAR)," for more information.

7.3 Watchdog Timer Implementation

When the watchdog timer expires in such a manner as requiring a reset, the core does not perform the reset. Instead the core output signals *core_wrs*[0:1] to reflect the value of TSR[WRS]. The intention is to signal the system that a watchdog reset event has occurred. The system can then implement a reset strategy. In general, the default strategy will normally be to reset the core, however, leaving the policy decision up to the integrated device allows for other strategies to be optionally implemented. See the reference manual

for the integrated device for details on what occurs on a watchdog timer expiration that should result in reset.

7.4 Performance Monitor Time Base Event

The e500mc provides the ability to count transitions of the TBL bit selected by PMGC0[TBSEL]. This count is enabled by setting PMGC0[TBEE]. For specific information, see Chapter 9, “Debug and Performance Monitor Facilities.”

Chapter 8
Power Management

This chapter describes the power management facilities as they are implemented on the e500mc core. The scope of this chapter is limited to the features of the e500mc only. Additional power management capabilities associated with a device that integrates this core (referenced as the integrated device throughout the chapter) are documented in the integrated device's reference manual.

8.1 Overview

A complete power management scheme for a system using the e500mc requires the support of the integrated device. The programming model and control of power management states for the core is provided by the integrated device. With the exception of the **wait** instruction, all other power management states are achieved through registers provided by the integrated device.

Power management consists of separate states, shown in Table 8-1 which correspond to power management states documented in the integrated device reference manual. These states map directly to core activity states, shown in Table 8-2, which describe more of the state machine of how the core transitions between states. These transitions are driven by power management signals, shown in Table 8-3, to the e500mc from the integrated device. In general, software does not need to concern itself about core activity states or the power management signals since the transitions are handled by signals from the integrate

8.2 e500mc and Integrated Device Power Management States

The core provides four different power management states in addition to normal operation. These states are called *wait*, *doze*, *nap*, and *sleep*. The *doze*, *nap*, and *sleep* states are controlled through integrated device registers and cause the core to transition to different activity states (*pm_halted*, *pm_stopped*) while the *wait* state is initiated and controlled solely by the core. Power management states are described in Table 8-1.

Table 8-1. e500mc Power Management States

State	Description
wait	The core stops fetching and execution of instructions. All core clocks are active. Timebase continues to increment and timer functions are active. All state is retained and snooping activity for the caches and other broadcast CoreNet operations such as **msgsnd** and **tlbivax** are still active. The *wait* state is entered when the core executes a **wait** instruction. The *wait* state is terminated and normal operation resumes when any asynchronous interrupt is ready to be taken by the core. When the *wait* state terminates, the core will take the interrupt, and the save/restore register indicating the address to return to after the interrupt is processed will point to the instruction following the **wait** instruction. Note that an external interrupt that is pending, but is not enabled by the core, will not cause the *wait* state to be terminated. The *wait* state is solely initiated by the core and as such does not participate in the protocol between the integrated device and the core with respect to *pm_halted* and *pm_stopped* core activity states. Because state is retained in the caches and core registers, and the caches continue to participate in snooping activities, software does not need to perform any specific actions prior to entering the *wait* state to ensure that coherent state is maintained.
doze	The *doze* state provides a similar level of power savings as the *wait* state, but is controlled by the integrated device and will terminate when an external asynchronous interrupt is pending, even if the core does not have that interrupt enabled. The core stops fetching and execution of instructions. All core clocks are active. Timebase continues to increment and timer functions are active. All state is retained and snooping activity for the caches and other broadcast CoreNet operations such as **msgsnd** and **tlbivax** are still active. The *doze* state is entered when the integrated device is programmed to signal the core to enter the *doze* state. To enter the *doze* state, the integrated device signals the core to enter the *pm_halted* activity state. The *doze* state is terminated and normal operation resumes when an asynchronous external interrupt to be signalled by the integrated device is pending. The *doze* state may also be terminated when one of the following internally generated asynchronous interrupts is pending: decrementer, fixed interval timer, watchdog timer, machine check, performance monitor, processor doorbell, processor doorbell critical, guest processor doorbell, guest processor doorbell critical, and guest processor doorbell machine check. When the *doze* state terminates the integrated device signals the core to exit the *pm_halted* activity state. The core resumes fetching and executing instructions from the point at which it stopped executing instructions. If the interrupt condition which caused the core to exit the *doze* state is enabled and the interrupt is still pending, the interrupt will immediately be taken and the save/restore register indicating the address to return to after the interrupt is processed will point to the instruction which would have executed next after the core entered the *pm_halted* activity state. Because state is retained in the caches and core registers, and the caches continue to participate in snooping activities, software does not need to perform any specific actions prior to entering the *doze* state to ensure that coherent state is maintained.

Table 8-1. e500mc Power Management States (continued)

State	Description
nap	The core stops fetching and execution of instructions. Core clocks are turned off by the integrated device, except for the timebase. The core retains all its state, however with clocks off the core will not receive and process transactions from CoreNet. Operations such as snoops, acceptance of messages from a **msgsnd** operation, and TLB invalidations from **tlbivax** operations will not be seen by the core and will be lost with respect to the core. The *nap* state is entered when the integrated device is programmed to signal the core to enter the *nap* state. To enter the *nap* state, the integrated device signals the core to enter the *pm_halted* activity state and then signals the core to enter the *pm_stopped* state. The *nap* state is terminated and normal operation resumes when an asynchronous external interrupt to be signalled by the integrated device is pending. The *nap* state may also be terminated when one of the following internally generated asynchronous interrupts is pending: decrementer, fixed interval timer, watchdog timer, machine check, and performance monitor. When the *nap* state terminates the integrated device signals the core to transition from the *pm_stopped* activity state to the *pm_halted* activity state, then exits the *core_halted* activity state. The core resumes fetching and executing instructions from the point at which it stopped executing instructions. If the interrupt condition which caused the core to exit the *nap* state is enabled and the interrupt is still pending, the interrupt will immediately be taken and the save/restore register indicating the address to return to after the interrupt is processed will point to the instruction which would have executed next after the core entered the *pm_halted* activity state. Because state is retained in the caches and core registers, but the caches no longer continue to participate in snooping activities, software should always flush, then invalidate the caches prior to initiating *nap* state to ensure that any modified data is written out to backing store. Upon exit from *nap* state, software must update any TLB entries that may have changed due to invalidations that were missed while the core was in the *pm_stopped* activity state. In general, this will require the flushing of any dynamic TLB entries and reloading them from the software page table. Because the core must flush its caches immediately prior to entering the *nap* state, the *nap* state will generally only be initiated by writing the appropriate integrated device registers by the specific core which will enter the *nap* state (that is, a core will generally *nap* itself, not another core).
sleep	The *sleep* state is the same as the *nap* state, except that the timebase functions are also turned off. All software activities required of the *nap* state are also required by the *sleep* state. In addition, since the timebase is also turned off during *sleep*, upon exit from *sleep* state, software will have to reload the timebase from some source external to the core. During sleep, the core will not wake from internally generated asynchronous interrupts because the core is not processing any events that might cause a wakeup condition to be noted.

The notion of *nap*, *doze*, and *sleep* modes (or states) pertains to, and are defined by, the integrated device as a whole. As shown in Figure 8-1, an integrated device may define the terms *nap*, *doze*, and *sleep* to mean different things. However, the integrated device controls the core power management by requesting the core to enter the core activity states *pm_halted*, *pm_stopped*, and by manipulating the timebase enable (*tben*) signal.

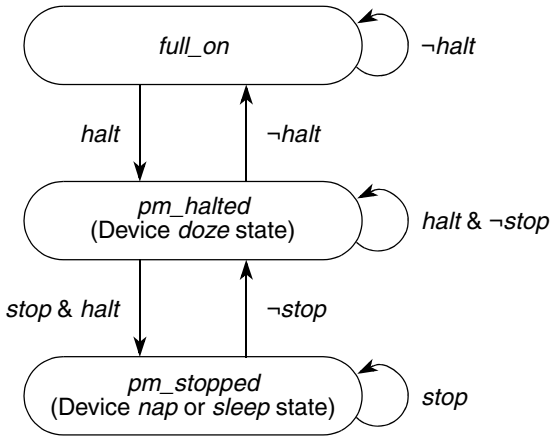


Figure 8-1. Core Activity State Diagram

In addition to the power-management states, dynamic power management automatically stops clocking individual internal functional units whenever they are idle.

Table 8-2 describes the core activity states.

Table 8-2. Core Activity States

State	Descriptions
full_on (default)	Default. All internal units are operating at the full clock speed defined at power-up. Dynamic power management automatically stops clocking individual internal functional units that are idle.
pm_halted	Initiated by asserting the *halt* input. The e500mc responds by stopping instruction execution. It then it asserts the *halted* output to indicate that it is in the *core_halted* state. Core clocks continue running, and snooping continues to maintain cache coherency. As Figure 8-1 shows, the e500mc is in *pm_halted* state when the integrated device is in *doze* state. The following occur once the core is in *core_halted* state: • Suspend instruction fetching. • Complete all previously fetched instructions and associated data transactions.
pm_stopped	Initiated when *stop* is asserted to the core while it is in *pm_halted* state. The core responds by inhibiting clock distribution to most of its functional units (after the CoreNet interface idles), and then asserting the *stopped* output.
tben	Disabling the timebase facilities. Additional power reduction is achieved by negating the time base enable (*tben*) input, which stops timebase operations. Note that *tben* controls the timebase in all power management states. Timer operation is independent of power management except for software considerations required for processing timer interrupts that occur during *pm_stopped* state. For example, if the timer facility is stopped, software ordinarily uses an external time reference to update the various timing counters upon restart.

8.3 Power Management Signals

Table 8-3 summarizes the power management signals of the e500mc. Power management signals cause the core to transition to different power management states and core activity states. Power management states are shown in Table 8-1 and core activity states are shown in Table 8-2.

Table 8-3. Power Management Signals

Signal	I/O	Description
halt	I	Asserted by the integrated device to initiate actions that cause the core to enter *halted* state.
halted	O	Asserted by the core when it reaches *pm_halted* state.
stop	I	Asserted by the integrated device to initiate the required actions that cause the core to go from *pm_halted* into *pm_stopped* state (as described in Table 8-2). Negating *stop* returns the core to *pm_halted* state.
stopped	O	Asserted by the core anytime the internal functional clocks of the e500mc are stopped (for example after integrated device asserts *stop*).
tben	I	Asserted by the integrated device to enable the timebase.
wake_req	O	Asserted when the core detects an internally generated asynchronous interrupt is enabled and pending. This prompts the integrated device to bring the core to a *full_on* activity state to service the interrupt. The interrupts that can assert *wake_req* are: decrementer, fixed interval timer, watchdog timer, machine check, performance monitor, processor doorbell, processor doorbell critical, guest processor doorbell, guest processor doorbell critical, and guest processor doorbell machine check.

8.4 Power Management Protocol

The e500mc responds to signals driven by the platform that command the core to transition from *full_on* state to *pm_halted* or *pm_stopped* state by driving the *halt* and *stop* signals. When the core has reached the requested state, it outputs the *halted* or *stopped* signals to inform the platform that the state transition is complete.

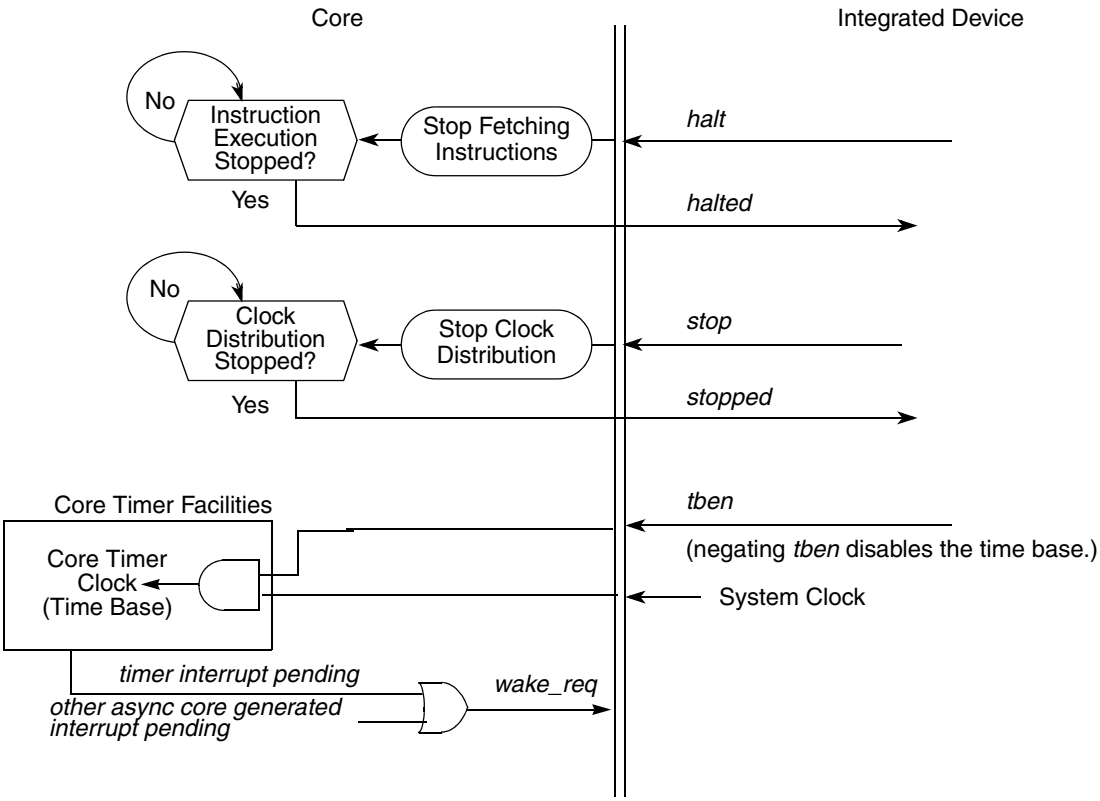


Figure 8-2. Core Power Management Handshaking

8.5 Interrupts and Power Management

In *pm_halted* or *pm_stopped* activity state, the core does not recognize external interrupt requests from the integrated device. The power management logic of the integrated device must monitor all external interrupt requests (as well as the e500mc *wake_req* output) to detect interrupt requests. Upon sensing an interrupt request, the integrated device ordinarily negates *stop* and *halt* to restore the core to *full_on* activity state, allowing it to service the interrupt request.

The control of power management state changes is done completely through the integrated device, including current state and previous state status. Consult the user manual of the integrated device for information on the programming model.

Chapter 9
Debug and Performance Monitor Facilities

9.1 Debug and Performance Monitor Facilities Overview

The e500mc core provides hardware support for the following:

- Software debuggers that run natively on the e500mc processor
- External software debuggers that run on external hardware that is attached to the e500mc processor

The architecture defines a set of debug features that support software debuggers that run natively on the processor. These features include trace facilities and instruction and data address breakpoints. The architecture specifies that these features trigger debug interrupts when they are enabled for internal debug mode (IDM). The e500mc implements these features as they are described in the debug chapter of the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

The e500mc provides the capability for an external debugger to leverage the architecture-defined debug mechanisms and use them when they are enabled for external debug mode (EDM). In EDM, the external debugger has control over these mechanisms, and software running on the e500mc cannot alter them. In EDM, the architecture-defined debug features can be used to cause the core to halt (stop fetching and executing instructions) instead of taking a debug interrupt. The e500mc has also implemented the **dnh** instruction, which when enabled, causes the processor to halt for the external debugger.

The performance monitor facility allows software running on the processor to collect information about events that occur in the processor. Software can configure counters to count events and later harvest those events to determine performance aspects of running software. The performance monitor facility is non-intrusive except when a performance monitor interrupt is configured. This can cause an interrupt if certain events, such as counter overflow, occur. The performance monitor facility is described in Section 9.11, "Performance Monitor."

9.1.1 Terminology

This chapter uses certain terminology that has the specific meanings defined in this section. This terminology is used elsewhere in this manual and in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* and has the same definition. Some of this terminology, such as 'debug event' appears in Power ISA™ and has a more limited scope, since Power ISA does not define external debug capabilities.

The term 'debug condition' indicates that a set of specific criteria have been met such that the corresponding debug event occurs in the absence of any gating or masking. The criteria for debug conditions are obtained from debug control registers.

The term ‘debug event’ means the setting of a bit in either the debug status register (DBSR) or external debug status register 0 (EDBSR0) upon the occurrence of the associated debug condition. However, a debug condition does not always result in a debug event. Conditions are prioritized with respect to exceptions. Except for some special UDE (Unconditional Debug Event) debug conditions, exceptions that have higher priority than a debug condition prevent the debug condition from being recorded as a debug event. In internal debug mode (IDM), debug events cause a debug interrupt if the debug enable bit is set (MSR[DE] = 1). It is possible that a UDE debug event can occur at the same time another debug event occurs.

The term ‘debug interrupt’ refers to the action of saving old context (machine state register and next instruction address) into the debug save/restore registers (DSRR0 and DSRR1) and beginning execution at a predetermined interrupt handler address. For additional information, see Section 4.9.16, “Debug Interrupt—IVOR15.”

In external debug mode (EDM) (EDBCR0[EDM] = 1), debug events cause the processor to halt.

9.2 Internal (Software) Debug Registers

Internal debug-related registers are accessible to software running on the processor. These registers are intended for use by special debug tools and debug software, and not by general application or operating system code. These registers are described in Section 2.17, “Internal Debug Registers.”

9.3 External Debug Registers

The external debug registers are used for controlling the core and reporting status while the e500mc is in external debug mode.

9.3.1 External Debug Control Register 0 (EDBCR0)

EDBCR0 is a control register that is accessible to an external debugger through the memory mapped interface. An external development tool can write to this register in order to enable EDM, to enable Debugger Notify Halt instructions (**dnh**), or disable certain asynchronous interrupts.

EDBCR0 is not accessible by software running on the e500mc core. However, the state of EDBCR0[EDM] is reflected as a read-only bit in DBCR0[EDM].

EDBCR0[EDM] takes precedence over DBCR0[IDM]. Whenever EDM = 1, debug events are enabled (whether IDM = 0 or IDM = 1), and any enabled debug event causes the processor to halt.

When EDBCR0 bits controlling asynchronous interrupt disables are set (EDMEO, EDCEO, EDEEO), normal asynchronous interrupt enabling conditions are overridden. The setting of these bits does not modify the state of the *wake_req* signal from the core to SoC power management logic. Nor does the setting of these bits affect the execution of the **wait** instruction. When **wait** is executed and asynchronous interrupt disables are set, the processor will wait until the interrupt disables are removed (through the external debugger) and an interrupt occurs before continuing execution.

If an external debugger wishes to use the EDMEO, EDCEO, or EDEEO bits to mask the taking of asynchronous interrupts, it should set these bits prior to changing any processor state after the processor

has been halted to prevent the processor from committing to an interrupt. For example, if after the processor is halted and the debugger jams a **mtmsr** instruction that sets MSR[EE] and the external input pin is signalling an external input interrupt is present, the core will be committed to take that interrupt and will take the interrupt when the processor is taken out of the halted state even if the processor sets the EDMEO, EDCEO, or EDEEO bits prior to resuming execution.

EDBCR0, shown in Figure 9-1, contains bits for enabling external debug features.

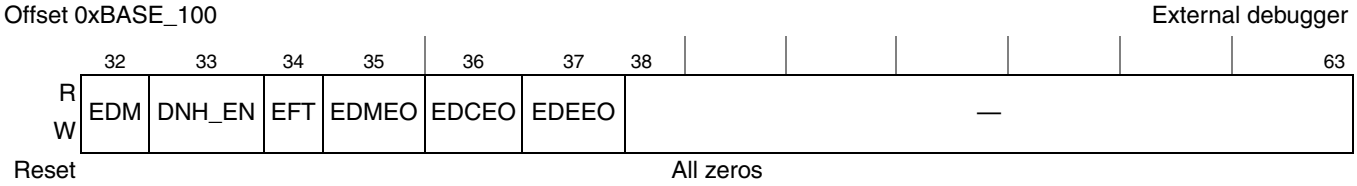


Figure 9-1. External Debug Control Register 0 (EDBCR0)

This table describes EDBCR0 fields.

Table 9-1. EDBCR0 Field Descriptions

Bits	Name	Description
32	EDM	External Debug Mode 0 = The core is not in external debug mode. Debug events do not cause the processor to halt. 1 = The core is in external debug mode. A qualified debug condition generates an external debug event which updates the corresponding EDBSR0 bit and causes the processor to halt.
33	DNH_EN	Debugger Notify Halt Enable 0 = A Debugger Notify Halt instruction (**dnh**) results in an illegal instruction exception 1 = **dnh** causes the processor to halt and update PRSR[DNHM].
34	EFT	(External) Freeze timers on debug halt 0 = Time base counters continue to run during debug halted state 1 = Time base counters freeze when entering debug halted state
35	EDMEO	Debugger Machine Check Interrupt Enable Override. When this bit is set, no asynchronous machine check interrupts will occur. Exception conditions for asynchronous machine check interrupts which occur will remain pending. This bit has no effect on error report interrupts, nor does it disable the NMI interrupt which is taken on the machine check level. 0 = Asynchronous machine check interrupts are enabled as described by the architecture. MSR[ME] and MSR[GS] are used to determine if an asynchronous machine check interrupt can be taken. 1 = Asynchronous machine check interrupts are disabled. MSR[ME] and MSR[GS] are not used to determine whether an asynchronous machine check interrupt can be taken. NMI interrupts are not affected by the setting of this bit. This bit should only be set when the processor is in External Debug mode by the external debugger. Architecturally, the behavior of this bit is undefined when the processor is not in EDM mode. For e500mc, this bit behaves the same regardless of whether the processor is in EDM mode.

Table 9-1. EDBCR0 Field Descriptions (continued)

Bits	Name	Description
36	EDCEO	Debugger Critical Interrupt Enable Override. When this bit is set, no asynchronous critical interrupts (critical input, processor doorbell critical, guest processor doorbell critical, guest processor doorbell machine check, or watchdog timer) will occur. Exception conditions for critical interrupts which occur will remain pending unless the pending condition is cleared. 0 = Critical interrupts are enabled as described by the architecture. MSR[CE] and MSR[GS] are used to determine if an asynchronous critical interrupt can be taken. 1 = Asynchronous machine check interrupts are disabled. MSR[CE] and MSR[GS] are not used to determine whether an asynchronous critical interrupt can be taken. This bit should only be set when the processor is in External Debug mode by the external debugger. Architecturally, the behavior of this bit is undefined when the processor is not in EDM mode. For e500mc, this bit behaves the same regardless of whether the processor is in EDM mode.
37	EDEEO	Debugger External Interrupt Enable Override. When this bit is set, no asynchronous external interrupts (external input, decrementer, fixed interval timer, performance monitor, processor doorbell, or guest processor doorbell) will occur. Exception conditions for external interrupts which occur will remain pending unless the pending condition is cleared. 0 External interrupts are enabled as described by the architecture. MSR[EE] and MSR[GS] are used to determine if an asynchronous external interrupt can be taken. 1 Asynchronous external interrupts are disabled. MSR[EE] and MSR[GS] are not used to determine whether an asynchronous external interrupt can be taken. This bit should only be set when the processor is in External Debug mode by the external debugger. Architecturally, the behavior of this bit is undefined when the processor is not in EDM mode. For e500mc, this bit behaves the same regardless of whether the processor is in EDM mode. **Note:** *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* allows implementations to consider a delayed floating-point enabled interrupt to be asynchronous, however the taking of delayed floating-point is not enabled by MSR[EE] and is unaffected by the setting of EDEEO.
38–63	—	Reserved

9.3.2 External Debug Status Register 0 (EDBSR0)

EDBSR0, shown in Figure 9-2, is a status register that is accessible to an external debugger through memory mapped access. If PRSR[DE_HALT] indicates that the core was halted by an enabled debug event when DBCR0[EDM] = 1 (see Section 9.3.6, "Processor Run Status Register (PRSR)"), the corresponding status bit is set within EDBSR0.

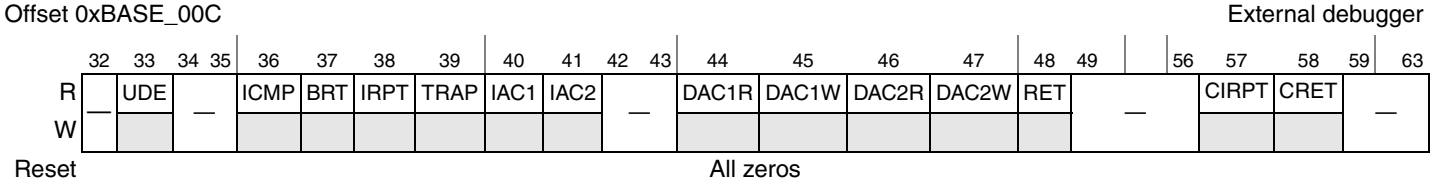


Figure 9-2. External Debug Status Register 0 (EDBSR0)

Table 9-2 describes EDBSR0 fields.

Table 9-2. EDBSR0 Field Descriptions

Bits	Name	Description
32	—	Reserved
33	UDE	Unconditional Debug Event Set if an unconditional debug condition occurred and DBCR0[EDM]=1
34–35	—	Reserved
36	ICMP	Instruction Complete Debug Event Set if an instruction complete debug condition occurred and DBCR0[EDM]=1
37	BRT	Branch Taken Debug Event Set if a branch taken debug condition occurred (DBCR0[BRT] = 1) and DBCR0[EDM]=1
38	IRPT	Interrupt Taken Debug Event Set if an interrupt taken debug condition occurred (DBCR0[IRPT] = 1) and DBCR0[EDM]=1
39	TRAP	Trap Instruction Debug Event Set if a trap instruction debug condition occurred (DBCR0[TRAP] = 1) and DBCR0[EDM]=1
40	IAC1	Instruction Address Compare 1 Debug Event Set if an IAC1 debug condition occurred (DBCR0[IAC1] = 1), and DBCR0[EDM]=1
41	IAC2	Instruction Address Compare 2 Debug Event Set if an IAC2 debug condition occurred (DBCR0[IAC2] = 1 && DBCR1[IAC12M] == 0), and DBCR0[EDM] = 1
42–43	—	Reserved
44	DAC1R	Data Address Compare 1 Read Debug Event Set if DBCR0[EDM] = 1 and a read-type DAC1 debug condition occurred (DBCR0[DAC1] = 10 or 11)
45	DAC1W	Data Address Compare 1 Write Debug Event Set if DBCR0[EDM] = 1 and a write-type DAC1 debug condition occurred (DBCR0[DAC1] = 01 or 11)
46	DAC2R	Data Address Compare 2 Read Debug Event Set if DBCR0[EDM] = 1 and a read-type DAC2 debug condition occurred (DBCR0[DAC1] = 10 or 11)
47	DAC2W	Data Address Compare 2 Write Debug Event Set if DBCR0[EDM] = 1 and a write-type DAC2 debug condition occurred (DBCR0[DAC1] = 01 or 11)
48	RET	Return Debug Event Set if a return debug condition occurred (DBCR0[RET] = 1) and DBCR0[EDM] = 1.
49–56	—	Reserved
57	CIRPT	Critical Interrupt Taken Debug Event Set if a critical interrupt debug condition occurred (DBCR0[CIRPT] = 1) and DBCR0[EDM] = 1
58	CRET	Critical Return Debug Event Set if a critical return debug condition occurred (DBCR0[CRET] = 1), and DBCR0[EDM] = 1
59–63	—	Reserved

Upon resuming, the status bit remains asserted until the next enabled debug event halts the core. This provides software visibility (read only) into the event that caused the most recent halt.

9.3.3 External Debug Status Register Mask 0 (EDBSRMSK0)

The external debug status register mask 0 (EDBSRMSK0) is used to mask debug events set in EDBSR0 from causing entry into debug halted mode. A “1” stored in any mask bit prevents debug HALT entry caused by the corresponding bit being set in EDBSR0. The mask has no effect on DBSR actions. EDBSRMSK0 may be used to allow debug events owned by hardware to be configured for watchpoint generation purposes without causing entry into a debug HALT state when the watchpoint occurs. EDBSRMSK0 is read and written via access by external development tools. No software access is provided. The EDBSRMSK0 register is shown in Figure 9-3.

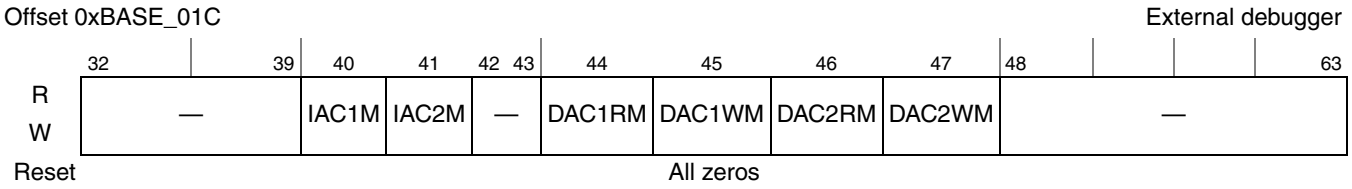


Figure 9-3. External Debug Status Register Mask 0 (EDBSRMSK0)

Table 9-3 provides bit definitions for external debug status register mask 0.

Table 9-3. EDBSRMSK0 Field Descriptions

Bits	Name	Description
32–39	—	Reserved
40	IAC1M	Instruction Address Compare 1 Debug Event Mask Set to 1 to mask entry into debug HALT state by EDBSR0[IAC1]
41	IAC2M	Instruction Address Compare 2 Debug Event Mask Set to 1 to mask entry into debug HALT state by EDBSR0[IAC2]
42–43	—	Reserved
44	DAC1RM	Data Address Compare 1 Read Debug Event Mask Set to 1 to mask entry into debug HALT state by EDBSR0[DAC1R]
45	DAC1WM	Data Address Compare 1 Write Debug Event Mask Set to 1 to mask entry into debug HALT state by EDBSR0[DAC1W]
46	DAC2RM	Data Address Compare 2 Read Debug Event Mask Set to 1 to mask entry into debug HALT state by EDBSR0[DAC2R]
47	DAC2WM	Data Address Compare 2 Write Debug Event Mask Set to 1 to mask entry into debug HALT state by EDBSR0[DAC2W]
48–63	—	Reserved

9.3.4 External Debug Status Register 1 (EDBSR1)

EDBSR1, shown in Figure 9-4, is a status register that is accessible to an external debugger through memory mapped access. It provides status information related to instruction jamming errors. The contents of EDBSR1 are only valid after an IJAM.

If a jammed instruction causes an exception, EDBSR1 indicates the presence of the exception, and which exception was signaled.

Jammed instructions do not take interrupts. In other words, the NIA (Next Instruction Address) is not altered to point to an interrupt handler, save/restore registers are not updated, and the MSR is not updated. Instead, EDBSR1[IJEE] is set, and the IVOR number for the exception is recorded in EDBSR1[IVOR].

EDBSR1[IJAE] indicates that an IJAM access error occurred, and EDBSR1[IJBUSY] indicates busy status on the IJAM access.

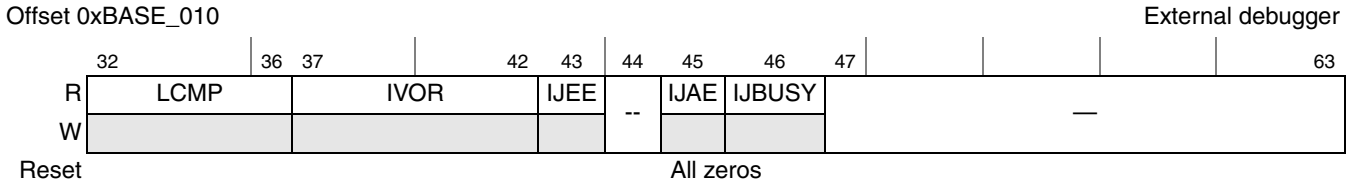


Figure 9-4. External Debug Status Register 1 (EDBSR1)

Table 9-4 describes EDBSR1 fields.

Table 9-4. EDBSR1 Field Descriptions

Bits	Name	Description
32–36	LCMP	Length Completed (instructions completed without error)[1] 00000 = No instructions completed without error 00001 = One instruction completed without error All other encodings are reserved.
37–42	IVOR	Interrupt Vector Offset Register Number If an exception occurs during an instruction jamming operation, the corresponding IVOR number is logged in this field. IVOR is valid only if IJEE is set. For example, if a program exception is recognized during an instruction jamming operation, IVOR would be set to 0x6 because program interrupts use IVOR6.
43	IJEE	Instruction Jamming Exception Error 0 = No exception occurred while executing the last instruction 1 = An exception occurred while executing the last instruction. EDBSR1[LCMP] indicates how many instructions completed prior to the exception while the IVOR field indicates what type of exception occurred. Note that exceptions that occur during instruction jamming operations do not cause interrupts.
44	—	Reserved
45	IJAE	Instruction Jamming Access Error[2] 0 = Most recent IJAM access completed without error 1 = An access error occurred during the IJAM operation

Table 9-4. EDBSR1 Field Descriptions (continued)

Bits	Name	Description
46	IJBUSY	Instruction Jamming Busy Status[3] 0 = IJAM access idle or completed (not busy) 1 = IJAM access not completed (busy)
47–63	—	Reserved

[1] The e500mc only supports jamming one instruction at a time

[2] EDBSR1[IJAE] is also available at the SoC. Refer to the SoC reference manual for details on external polling of this bit.

[3] EDBSR1[IJBUSY] is also available at the SoC. Refer to the SoC reference manual for details on external polling of this bit.

9.3.5 External Debug Exception Syndrome Register (EDESR)

The EDESR provides a syndrome to differentiate between the different kinds of exceptions that generate the same interrupt type. If an exception occurs during an instruction jamming operation, the syndrome information is captured in the EDESR instead of the ESR. EDESR fields are identical to those specified for the ESR, as described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* with e500mc-specific details listed in Section 2.9.6, "(Guest) Exception Syndrome Register (ESR/GESR)." EDESR is read only and is accessible through memory mapped access.

9.3.6 Processor Run Status Register (PRSR)

The PRSR, shown in Figure 9-5, provides status information for processor halt and stop. Halt requests are posted to PRSR as soon as they are recognized. When the processor is halted in response to a halt request, PRSR[HALTED] is set to indicate that the processor has reached the halted state. The latency between the posting of a halt request and the posting of the halted state depends on what the processor was doing at the time of the halt request.

The core remains halted or stopped as long as any of the halt or stop conditions exist. The power management conditions are cleared, and the corresponding PRSR bits are cleared, when the processor exits the halt or stop states. All other halt and stop conditions must be explicitly cleared by clearing the corresponding DBSR bit. These bits are cleared by writing a 1 to the bit. As long as PRSR indicates that any halted or stopped condition is active, the core remains halted or stopped.

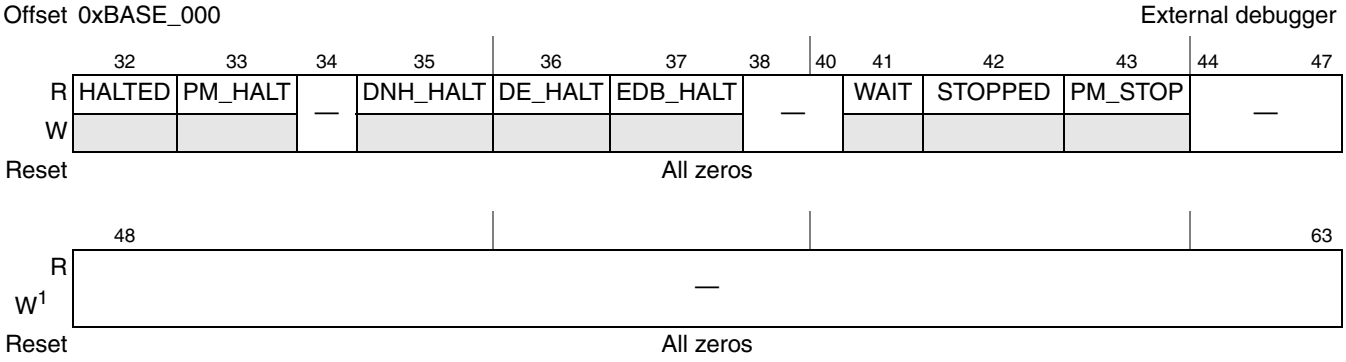


Figure 9-5. Processor Run Status Register (PRSR)

[1] The writable bits of this register support a write-1-to-clear functionality. Writing zeros has no effect.

Table 9-5 describes PRSR fields.

Table 9-5. PRSR Field Descriptions

Bits	Name	Description
32	HALTED	Halted state. Set whenever the core is halted. Cleared whenever the core resumes program execution.
33	PM_HALT	Power management halt. Set whenever the processor is halted in response to a power management request from the system. This is a non-debug halt.
34	—	Reserved
35	DNH_HALT	Debugger notify halt event. Set whenever the processor is halted in response to the **dnh** instruction. This bit should be cleared by the debugger prior to issuing a resume command.
36	DE_HALT	Debug event halt. Set whenever the processor is halted due to the occurrence of an enabled debug condition in EDM. This bit should be cleared by the debugger prior to issuing a RESUME command.
37	EDB_HALT	External debug halt request event. Set whenever the processor core receives a debug halt request from the system. This bit should be cleared by the debugger prior to issuing a resume command.
38–40	—	Reserved
41	WAIT	Processor is in a WAIT state caused by execution of a WAIT instruction.
42	STOPPED	Stopped state. Set whenever the core is stopped. Cleared whenever the core resumes program execution.
43	PM_STOP	Power management stop. Set whenever the processor is stopped in response to a power management stop request from the system. This is a non-debug stop.
44–58	—	Reserved
59–63	DNHM	Debugger notify halt message contains the additional information provided by the **dnh** instruction. The information is derived from the DUI operand of the **dnh** instruction.

9.3.7 Extended External Debug Control Register 0 (EEDCR0)

The EEDCR0, shown in Figure 9-6, provides extended controls not normally used in external debug operations.

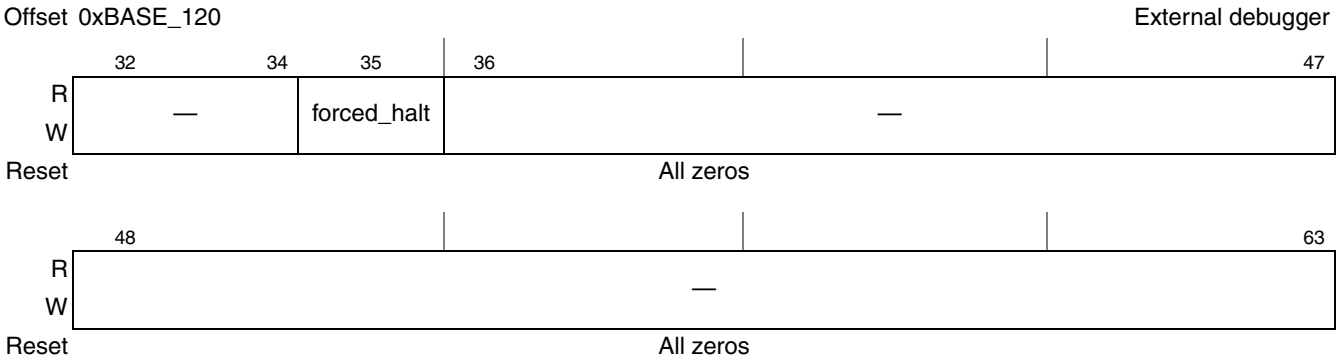


Figure 9-6. Extended External Debug Control Register 0 (EEDCR0)

Table 9-6 describes EEDCR0 fields.

Bits	Name	Description
32–34	—	Reserved
35	forced_halt	Forced halt. If a debug halt has been requested, but has not completed, writing a 1 to this field will force the processor to halt. When halting the processor using this mechanism, the processor may not be put back into a run state unless the entire integrated device is reset. If this bit is set and a debug halt is not in progress, the request will be ignored. When read, this field will always return 0. Forcing the processor to halt using this control should only be used when a normal halt command does not complete. The normal halt mechanism may fail to complete if there are problems in the CoreNet fabric whereby transactions are not completed. The processor in this case will fail to halt because part of the protocol for halting the core is to force all queued memory transaction to complete and wait until CoreNet has fully accepted those transactions. If the CoreNet fabric does not acknowledge the transactions, the halt sequence will hang. This control could then be used to force the processor into the halt state to examine the state of the processor. After a forced_halt is commanded, the external debugger should not take any action or jam any instructions which would cause the processor to attempt a transaction on the CoreNet interface, as doing so will likely cause the processor to hang. This field is not present on e500mc Rev 1.x or e500mc Rev 2.x.
36–63	—	Reserved

Table 9-6. EEDCR0 **Field Descriptions**

9.4 Nexus Registers

The Nexus control registers provide a mechanism to enable the various tracing features that are supported by the e500mc Nexus module. These registers are accessible through the NSPC and NSPD registers, described in Section 2.17.9, "Nexus SPR Access Registers."

9.4.1 Nexus Development Control Register 1 (DC1)

The DC1 provides basic trace enable controls for the core Nexus module.

Offset 0xBASE_408[1] External debugger

	32	38	39	40	48	49	50	51	52	53	54	55	56	57	58	63
R W	—		OTS	—		POTD	TSEN		EOC		EIC		—		TM	
Reset	All zeros															

Figure 9-7. Nexus Development Control Register 1 (DC1)

[1] Also accessible through NSPC/D. See Section 9.9.4.2, "Special-Purpose Register Access (Nexus Only)."

Table 9-7 describes DC1 fields.

Bits	Name	Description
32–38	—	Reserved
39	OTS	Ownership Trace PID Select 0 = PID0 data is transmitted within Ownership Trace Messages 1 = Nexus PID Register (NPIDR) data is transmitted within Ownership Trace Messages
40–48	—	Reserved
49	POTD	Periodic Ownership Trace Disable 0 = Periodic Ownership Trace message events are enabled 1 = Periodic Ownership Trace message events are disabled
50–51	TSEN	Timestamp Enable 0x = Timestamp is disabled 10 = Timestamp is enabled for all messages (timestamp is applied to all messages) 11 = Coarse timestamp is enabled (timestamp is periodically applied every 32 messages)
52–53	EOC	Event Out Control 00 = EVTO0 asserted upon occurrence of any watchpoints selected by DC2[EWC0] 01–11 = Reserved, EVTO0 behaves as disabled
54–55	EIC	Event In Control[1] 00 = EVTI0 assertion causes Program Trace Sync message (trigger use of uncompressed address information on next message) 01 = Reserved 10 = EVTI0 disabled for this module 11 = Reserved (should not be used to ensure future compatibility)
56–57	—	Reserved
58–63	TM	Trace Mode[2] 000000 = All trace disabled xxxxx1 = Ownership Trace enabled xxxx1x = Data Trace enabled xxx1xx = Program Trace enabled xx1xxx = Watchpoint Trace enabled x1xxxx = Reserved (writing x1xxxx may not read back the same at this bit position - software should not set TM_1 to a non-zero value) 1xxxxx = Data Acquisition Trace enabled

Table 9-7. DC1 Field Descriptions

1 EVTI may be used as a watchpoint condition independent of the settings of DC1[EIC]. SeeTable 9-43 for information on how events are mapped to watchpoints.

2 TM may be updated by hardware in response to watchpoint triggering. Writes to this field take precedence over hardware updates in the event of a collision. Refer to Section 9.4.5, “Nexus Watchpoint Trigger Control Register 1 (WT1),” for more information on watchpoint triggering.

9.4.2 Nexus Development Control Register 2 (DC2)

DC2, shown in Figure 9-8, provides controls for the Event Out signals EVTO[0:4], which are trigger outputs from the core. The functions performed by EVTO*n* assertion are integrated device-specific.

NOTE

Further details decribing the full debug functionality of this device are beyond the scope of this document. The debugging and performance monitoring capability enabled by the device hardware coexists within a debug ecosystem that offers a rich variety of tools at different levels of the hardware/software stack. Software development and debug tools from Freescale (Codewarrior), as well as third-party vendors, provide a rich set of options for configuring, controlling and analyzing debug- and performance-related events.

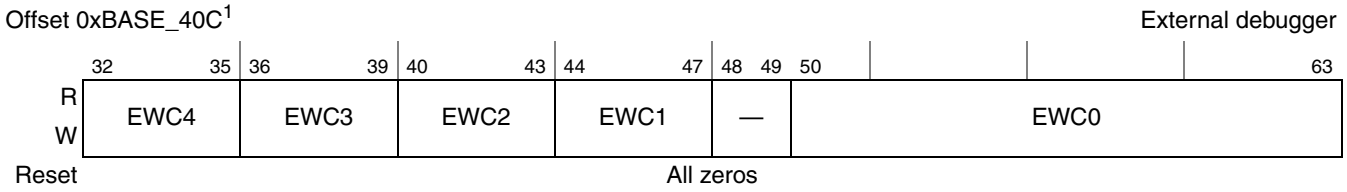


Figure 9-8. Nexus Development Control Register 2 (DC2)

[1] Also accessible through NSPC/D. See Section 9.9.4.2, "Special-Purpose Register Access (Nexus Only)."

Table 9-8 describes DC2 fields.

Table 9-8. DC2 Field Descriptions

Bits	Name	Description
32–35	EWC4	Event Out Watchpoint Control 4 0000 = No watchpoints trigger EVTO4 0001–1110 = Watchpoint #1–#14 (respectively) triggers EVTO4[1] 1111 = Reserved
36–39	EWC3	Event Out Watchpoint Control 3 0000 = No watchpoints trigger EVTO3 0001–1110 = Watchpoint #1–#14 (respectively) triggers EVTO3[1] 1111 = Reserved
40–43	EWC2	Event Out Watchpoint Control 2 0000 = No watchpoints trigger EVTO2 0001–1110 = Watchpoint #1–#14 (respectively) triggers EVTO2[1] 1111 = Reserved
44–47	EWC1	Event Out Watchpoint Control 1 0000 = No watchpoints trigger EVTO1 0001–1110 = Watchpoint #1–#14 (respectively) triggers EVTO1[1] 1111 = Reserved

Table 9-8. DC2 Field Descriptions (continued)

Bits	Name	Description
48-49	—	Reserved
50–63	EWC0	Event Out Watchpoint Control 0[1] 00000000000000 = No watchpoints trigger EVTO0 xxxxxxxxxxxxx1 = Watchpoint #1 triggers EVTO0 xxxxxxxxxxxx1x = Watchpoint #2 triggers EVTO0 xxxxxxxxxxx1xx = Watchpoint #3 triggers EVTO0 xxxxxxxxxx1xxx = Watchpoint #4 triggers EVTO0 xxxxxxxxx1xxxx = Watchpoint #5 triggers EVTO0 xxxxxxxx1xxxxx = Watchpoint #6 triggers EVTO0 xxxxxxx1xxxxxx = Watchpoint #7 triggers EVTO0 xxxxxx1xxxxxxx = Watchpoint #8 triggers EVTO0 xxxxx1xxxxxxxx = Watchpoint #9 triggers EVTO0 xxxx1xxxxxxxxx = Watchpoint #10 triggers EVTO0 xxx1xxxxxxxxxx = Watchpoint #11 triggers EVTO0 xx1xxxxxxxxxxx = Watchpoint #12 triggers EVTO0 x1xxxxxxxxxxxx = Watchpoint #13 triggers EVTO0 1xxxxxxxxxxxxx = Watchpoint #14 triggers EVTO0

1 Refer to Table 9-43 for information on the events that are mapped to these watchpoints.

9.4.3 Nexus Development Control Register 3 (DC3)

DC3 is not implemented on the e500mc.

9.4.4 Nexus Development Control Register 4 (DC4)

DC4, shown in Figure 9-9, provides additional control of Nexus debug features, Specifically, this register controls the masking of events which initiate Program Correlation Messages (PCM) as well as trace filters based on MSR state.

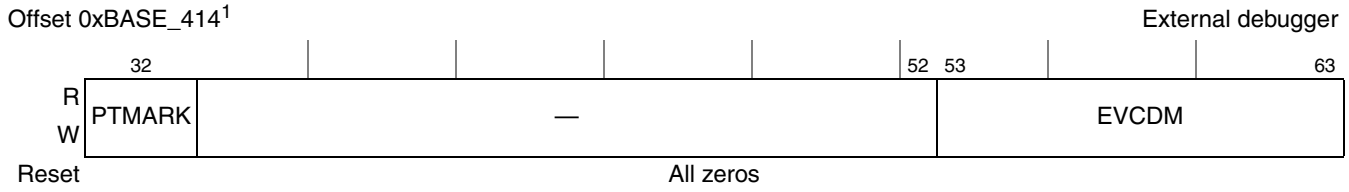


Figure 9-9. Nexus Development Control Register 4 (DC4)

1 Also accessible through NSPC/D. See Section 9.9.4.2, “Special-Purpose Register Access (Nexus Only).”

Table 9-9 describes DC4 fields.

Table 9-9. DC4 Field Descriptions

Bits	Name	Description
32	PTMARK	Program Trace Mark 0 = Ignore MSR[PMM] for masking program trace 1 = Mask program trace when MSR[PMM] = 0, unmask program trace when MSR[PMM] = 1. MSR[PMM] is the performance monitor mark bit. System software can set PMM to mark which execution contexts enable performance monitor statistics to be gathered. This bit is used for Nexus to provide execution context filtering. See Section 2.7.1, "Machine State Register (MSR)."
33–52	—	Reserved
53–63	EVCDM	Event Code (EVCODE) Mask[1] 00000000000 = No EVCODEs masked for Program Correlation Messages xxxxxxxxx1 = EVCODE #1 is masked for Program Correlation Messages xxxxxxxx1x = EVCODE #2 is masked for Program Correlation Messages xxxxxxx1xx–xxxxxx1xxx = Reserved xxxxx1xxxx = EVCODE #5 is masked for Program Correlation Messages xxxx1xxxxx–xx1xxxxxxx = Reserved x1xxxxxxxx = EVCODE #10 is masked for Program Correlation Messages 1xxxxxxxxx = EVCODE #11 is masked for Program Correlation Messages

[1] Refer to Table 9-35 for implemented EVCODEs

9.4.5 Nexus Watchpoint Trigger Control Register 1 (WT1)

WT1, shown in Figure 9-10, provides controls for watchpoint triggers that can be used to start and stop tracing, producing a temporal window of trace information.

Whenever a start trigger is detected, the designated trace features are enabled, and the corresponding DC1 enable bits are set. Whenever a stop trigger is detected, the designated trace features are disabled, and the corresponding enable DC1 bits are cleared. If the same trigger condition is used for both start and stop triggering, the designated trace features toggle between being enabled and disabled at each occurrence of the trigger condition. Similarly, if start and stop triggers for a trace feature occur simultaneously, the designated trace feature toggle between enabled and disabled depending on the enable state at the time of the trigger events. For example, if tracing is enabled, and start and stop triggers occur simultaneously, tracing is disabled. Direct writes of DC1 take precedence over any trace feature enable state that is derived from watchpoint triggering.

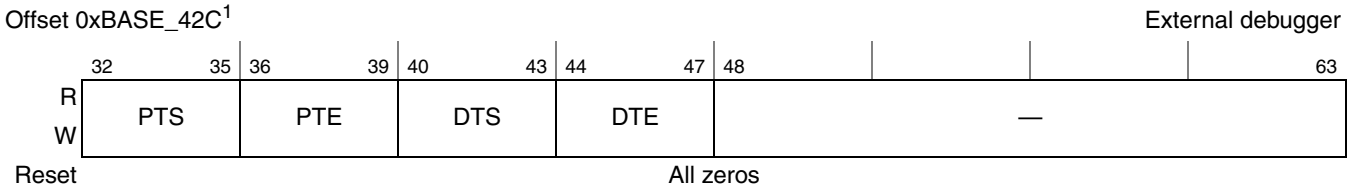


Figure 9-10. Nexus Watchpoint Trigger Register 1

[1] Also accessible through NSPC/D. See Section 9.9.4.2, "Special-Purpose Register Access (Nexus Only)."

Table 9-10 describes WT1 fields.

Bits	Name	Description
32–35	PTS	Program Trace Start 0000 = Trigger disabled 0001–1110 = Start program trace on Watchpoints 1–14 respectively (see Table 9-43) 1111 = Reserved
36–39	PTE	Program Trace End 0000 = Trigger disabled 0001–1110 = Stop program trace on Watchpoints 1–14 respectively (see Table 9-43) 1111 = Reserved
40–43	DTS	Data Trace Start 0000 = Trigger disabled 0001–1110 = Start data trace on Watchpoints 1–14 respectively (see Table 9-43) 1111 = Reserved
44–47	DTE	Data Trace End 0000 = Trigger disabled 0001–1110 = Stop data trace on Watchpoints 1–14 respectively (see Table 9-43) 1111 = Reserved
48–63	—	Reserved

Table 9-10. WT1 Field Descriptions

Note: Using start and stop triggers for data trace may preclude the ability to correlate data trace and program trace if watchpoint messages are used with DACs as a means to try and correlate data trace messages to the appropriate region of code (that is, the program trace).

9.4.6 Nexus Watchpoint Mask Register (WMSK)

WMSK, shown in Figure 9-11, controls which watchpoint events are enabled to produce Watchpoint Trace Messages (DC1[TM] must also be programmed to generate Watchpoint Trace Messages).

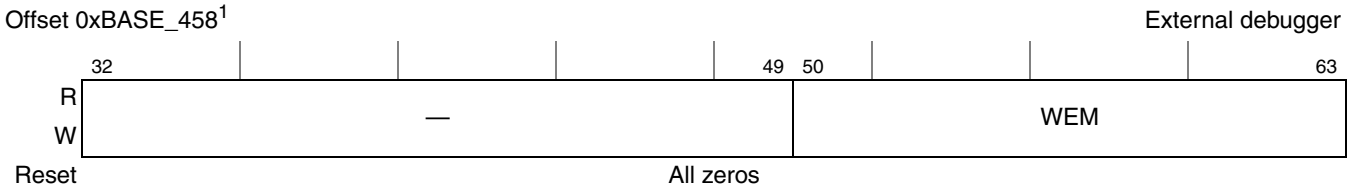


Figure 9-11. Nexus Watchpoint Mask Register

[1] Also accessible through NSPC/D. See Section 9.9.4.2, “Special-Purpose Register Access (Nexus Only).”

Table 9-11 describes WMSK fields.

Bits	Name	Description
32–49	—	Reserved
50–63	WEM	Watchpoint Enable for Messaging[1] 00000000000000 = No Watchpoints enabled for Watchpoint Trace Messaging xxxxxxxxxxxxx1 = Watchpoint #1 is enabled for Watchpoint Trace Messaging xxxxxxxxxxxx1x = Watchpoint #2 is enabled for Watchpoint Trace Messaging xxxxxxxxxxx1xx = Watchpoint #3 is enabled for Watchpoint Trace Messaging xxxxxxxxxx1xxx = Watchpoint #4 is enabled for Watchpoint Trace Messaging xxxxxxxxx1xxxx = Watchpoint #5 is enabled for Watchpoint Trace Messaging xxxxxxxx1xxxxx = Watchpoint #6 is enabled for Watchpoint Trace Messaging xxxxxxx1xxxxxx = Watchpoint #7 is enabled for Watchpoint Trace Messaging xxxxxx1xxxxxxx = Watchpoint #8 is enabled for Watchpoint Trace Messaging xxxxx1xxxxxxxx = Watchpoint #9 is enabled for Watchpoint Trace Messaging xxxx1xxxxxxxxx = Watchpoint #10 is enabled for Watchpoint Trace Messaging xxx1xxxxxxxxxx = Watchpoint #11 is enabled for Watchpoint Trace Messaging xx1xxxxxxxxxxx = Watchpoint #12 is enabled for Watchpoint Trace Messaging x1xxxxxxxxxxxx = Watchpoint #13 is enabled for Watchpoint Trace Messaging 1xxxxxxxxxxxxx = Watchpoint #14 is enabled for Watchpoint Trace Messaging

Table 9-11. WMSK Field Descriptions

[1] Refer to Table 9-43 for information on the events that are mapped to these watchpoints.

9.4.7 Nexus Overrun Control Register (OVCR)

The OVCR, shown in Figure 9-12, controls Nexus behavior as the internal message queues fill up. Response behavior options include suppressing selected message types, stalling the processor's instruction completion and stopping the processor clocks. Refer to Section 9.10.6, "Nexus Message Queues," for more information regarding the internal message queues.

Offset 0xBASE_45C[1] External debugger

	32 33	34 35	36 41 42 47	48 49	50 51	52 63
R W	—	SPTHOLD	—	SPEN	—	STTHOLD	—	STEN
Reset	All zeros					

Figure 9-12. Nexus Overrun Control Register

[1] Also accessible through NSPC/D. See Section 9.9.4.2, "Special-Purpose Register Access (Nexus Only)."

Table 9-12 describes OVCR fields.

Table 9-12. OVCR Field Descriptions

Bits	Name	Description
32–33	—	Reserved
34–35	SPTHOLD	Suppression Threshold 00 = Suppression threshold is when message queues are 1/4 full 01 = Suppression threshold is when message queues are 1/2 full 10 = Suppression threshold is when message queues are 3/4 full 11 = Reserved
36–41	—	Reserved
42–47	SPEN	Suppression Enable 000000 = Suppression is disabled xxxxx1 = Ownership Trace message suppression is enabled xxxx1x = Data Trace message suppression is enabled xxx1xx = Program Trace message suppression is enabled xx1xxx = Watchpoint Trace message suppression is enabled x1xxxx = Reserved 1xxxxx = Data Acquisition message suppression is enabled
48–49	—	Reserved
50–51	STTHOLD	Stall Threshold 00 = Stall threshold is when message queues are 1/4 full 01 = Stall threshold is when message queues are 1/2 full 10 = Stall threshold is when message queues are 3/4 full 11 = Reserved
52–62	—	Reserved
63	STEN	Stall Enable 0 = Processor stalling is disabled 1 = Processor stalling is enabled

9.5 Instruction Jamming (IJAM) Registers

This section discusses the following IJAM registers:

- Section 9.5.1, “IJAM Configuration Register (IJCFG)”
- Section 9.5.2, “IJAM Instruction Register (IJIR)”
- Section 9.5.3, “IJAM Data Registers 0 and 1 (IJDATA0, IJDATA1)”

9.5.1 IJAM Configuration Register (IJCFG)

IJCFG, shown in Figure 9-13, controls the basic settings for jamming instructions into the e500mc. It includes page attributes, addressing modes, and target storage space (memory or debug) for load/store instructions and other controls.

Offset 0xBASE_608 External debugger

	32–51	52–55	56	57	58–62	63
R W	—	IJRA	IJER	—	WIMGE	IJMODE
Reset	All zeros					

Figure 9-13. IJAM Configuration Register

Table 9-13 describes IJCFG fields.

Table 9-13. IJCFG Field Descriptions

Bits	Name	Description
32–51	—	Reserved
52–55	IJRA	Real Address (bits 28:31). If the jammed instruction is a load or store instruction, and IJER = 1, these 4 bits are prepended to the 32-bit effective address to form a 36-bit physical address (PA[28:63] = IJRA[0:3] || EA[32:63]). This field is only used when the jammed instruction is a load or store instruction.
56	IJER	Instruction Jamming Load/store Effective/Real Addressing Mode 0 = Load/store instruction (current access) uses effective addressing mode (MMU translation) 1 = Load/store instruction (current access) uses real addressing mode (no MMU translation)
57	—	Reserved
58–62	WIMGE	Page attributes for any storage access instruction (current access) when IJCFG[IJER] = 1. The meaning of these attributes is the same as defined when the processor is executing storage accesses through normal instruction execution. The definition of the WIMGE attributes can be found in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* -Cache and MMU Architecture.
63	IJMODE	Instruction Jamming Mode Control 0 = Load/store instructions (current access) target memory storage space. 1 = Load/store instructions (current access) target debug storage space. Changing the value of this field (whether load/store instructions target memory storage space or debug storage space) requires that a **sync 0** instruction be jammed and completed immediately prior to changing this field. This ensures that prior stores that may have been jammed are performed to the proper storage space.

9.5.2 IJAM Instruction Register (IJIR)

IJIR, shown in Figure 9-14, contains the instruction that is to be jammed into the e500mc core. Table 9-26 lists instructions supported for IJAM and the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors lists the* Instruction Set.

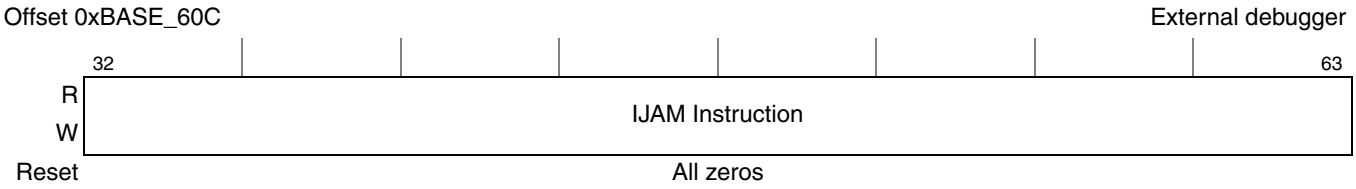


Figure 9-14. IJAM Instruction Register (IJIR)

9.5.3 IJAM Data Registers 0 and 1 (IJDATA0, IJDATA1)

IJDATA0 and IJDATA1, shown in Figure 9-15, contain data associated with load/store instructions. Data is written to this register when the jammed instruction requires associated write data (for example, load instructions from debug space). Data is read from this register when the jammed instruction has associated result data (for example, stores to debug space). These registers should be used when IJCFG[IJMODE] = 0’b1.

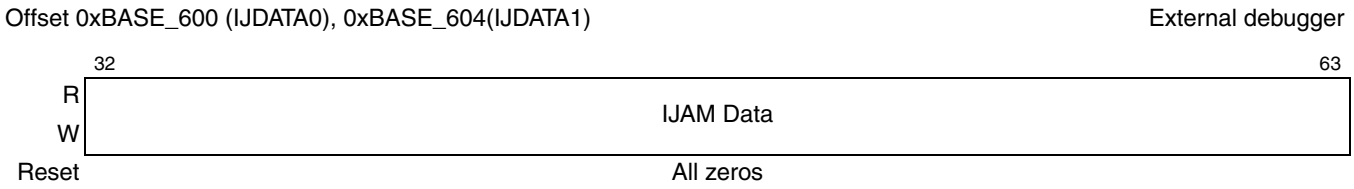


Figure 9-15. IJAM Data Registers (IJDATA0–IJDATA1)

9.6 Performance Monitor Registers (PMRs)

The performance monitor provides a set of PMRs for defining, enabling, and counting conditions that trigger the performance interrupt. PMRs for e500mc are described in Section 2.18, “Performance Monitor Registers (PMRs).”

9.7 Capture Registers

Capture registers are shadow registers that are used to capture a snapshot of an another register when requested. The capture register can then be accessed to determine the value at the time the snapshot occurred.

9.7.1 Performance Monitor Counter Capture Registers (PMCC0–PMCC3)

The performance monitor counter capture registers (PMCC0–PMCC3), shown in Figure 9-16, are 32-bit registers which capture the PMC*n* counter values based on the EVTO4 trigger signal. Detail on the performance monitor capture feature can be found in Section 9.11.4.2, “Core Performance Monitor & PC Capture Function.”

Offset PMCC 0xBASE_030, PMCC1- 0xBASE_034, PMCC1- 0xBASE_038, PMCC1- 0xBASE_03C — External debugger

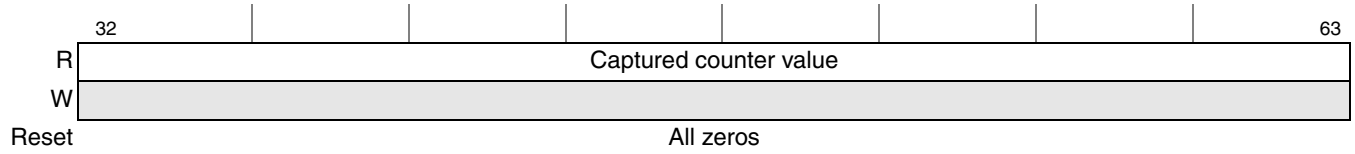


Figure 9-16. Performance Monitor Counter Capture Registers (PMCC0–PMCC3)

Table 9-14 describes PMCC register field.

Table 9-14. PMCC0–PMCC3 Field Descriptions

Bits	Name	Description
32–63	Counter Value	Value of the PMC*n* counter upon occurrence of the EVTO[4] trigger

9.7.1.1 Program Counter Capture Register (PCC)

The program counter capture register (PCC), shown in Figure 9-17, is a 32-bit register which captures the micro-architected program counter value based on the EVTO4 trigger signal. For e500mc, the program counter is accurate to within two instruction windows from when the signal is detected by the core and the two instructions at the bottom of the completion queue. Detail on the performance monitor capture feature can be found in Section 9.11.4.2, "Core Performance Monitor & PC Capture Function".

Offset 0xBASE_02C — External debugger

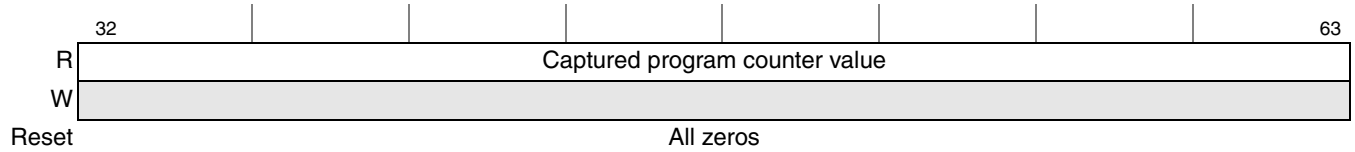


Figure 9-17. Program Counter Capture Register (PCC)

Table 9-15 describes PCC register field.

Table 9-15. PCC Field Descriptions

Bits	Name	Description
32–63	Captured program counter value	Value of the program counter upon occurrence of the EVTO[4] trigger

9.8 Debug Conditions

Debug events and debug interrupts are implemented (with a few exceptions) as defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. Conditions specific to the e500mc are described in the following sections.

9.8.1 Embedded Hypervisor

In the presence of hypervisor software, debug events are modified to be suppressed when debug capabilities are enabled in guest state. This prevents debug events from being recorded (and subsequent

debug interrupts from occurring) when executing in embedded hypervisor state when the guest operating system is using the debug facility.

When EPCR[DUVD] = 1 and MSR[GS] = 0, all debug events and associated exceptions do not occur except for the unconditional debug event, and no debug events are posted in the DBSR. Refer to the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* for more details on the embedded hypervisor.

9.8.2 Internal and External Debug Modes

The *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* and Power ISA specifies how the processor behaves in IDM. This is when DBCR0[EDM] = 0 and DBCR0[IDM] = 1.

The architecture allows implementation-dependent behavior when in EDM (DBCR0[EDM] = 1). In EDM, the core behaves as follows:

- **A mtspr** that attempt to change DBCRs, IACs, DACs, NSPC, NSPD, DBSR, or DBSRWR behaves as a NOP. The only exception is that jamming an **mtspr** instruction alters these registers.
- DBSR is not updated when a debug event occurs.
- When a debug event occurs, the core immediately halts. The debug interrupt is not taken. DSRR0, DSRR1, MSR, and ESR are not updated before halting. The NIA (Next Instruction Address) is not redirected to the first instruction of the debug interrupt handler.
- Upon halting for a debug event, the NIA contains the value that would otherwise have been placed in DSRR0 if the processor was in IDM.
- PRSR[DE_HALT] is set, and EDBSR0 indicates which debug events caused the core to halt.
- Delayed debug interrupts do not halt the core.

Note that, in IDM, BRT and ICMP events are not recognized unless debug interrupts are enabled (MSR[DE] = 1). therefore, BRT and ICMP can not cause imprecise debug events. However, in EDM, BRT and ICMP events are always recognized and cause the processor to halt, even if MSR[DE] = 0.

Because EDM usurps control of the architecture-defined debug control and status registers, simultaneous use of an internal and an external debugger is problematic. In particular, it is difficult to use an external debugger to debug an internal debugger. The **dnh** instruction can provide some assistance, though.

9.8.3 Debug Event Response Tables

The following tables identify the masking and gating functions for each condition and indicate the responses. Table 9-16 lists responses for instruction complete (ICMP) and branch taken (BRT) conditions.

NOTE

All debug conditions are gated by DBCR0

Table 9-16. Response—ICMP, BRT

HALTED	DBCR0		MSR[DE]	Action
	EDM	IDM		
1	x	x	x	No action
0	1	x	x	Halt. Set EDBSR0[x].
0	0	x	0	No action
0	0	0	x	No action
0	0	1	1	Issue debug interrupt. Set DBSR[x].

Table 9-17 lists responses for unconditional debug event (UDE), interrupt taken (IRPT), trap (TRAP), return from interrupt (RET), critical interrupt taken (CIRPT), and critical return from interrupt (CRET) conditions.

Table 9-17. Response—UDE, IRPT, TRAP, RET, CIRPT, CRET

HALTED	DBCR0		MSR[DE]	Action
	EDM	IDM		
1	x	x	x	No action.
0	1	x	x	Halt. Set EDBSR0[x]. Generate watchpoint (if enabled).
0	0	0	x	No action
0	0	1	0	Set DBSR[x], DBSR[IDE]. More than one DBSR bit may be set for imprecise debug events for various operations. Table 9-20 lists possible combinations of concurrent imprecise debug events for all operations.
0	0	1	1	Issue debug interrupt. Set DBSR[x]

Table 9-18 lists responses for instruction address compare 1 and 2 conditions.

Table 9-18. Response—IAC1, IAC2

HALTED	DBCR0		MSR[DE]	Action
	EDM	IDM		
1	x	x	x	No action.
0	1	x	x	Halt. Set EDBSR0[IAC1/2]. Generate IAC1/2 watchpoint
0	0	0	x	Generate IAC1/2 watchpoint. No other action.
0	0	1	0	Set DBSR[IAC1/2], DBSR[IDE]. Generate IAC1/2 watchpoint. More than one DBSR bit may be set for imprecise debug events for various operations. Table 9-20 lists possible combinations of concurrent imprecise debug events for all operations
0	0	1	1	Issue debug interrupt. Set DBSR[IAC1/2]. Generate IAC1/2 watchpoint

Table 9-19 lists responses for debug address compare 1 and 2 conditions.

Table 9-19. Response—DAC1, DAC2

HALTED	DBCR0		MSR[DE]	Another Debug Event on Same Instruction	Action
	EDM	IDM			
1	x	x	x	x	No action.
0	1	x	x	0	Halt. Set EDBSR0[DAC1/2]. Generate DAC1/2 watchpoint
0	1	x	x	1	Halt. Set EDBSR0[IAC1/2]. Generate DAC1/2 watchpoint
0	0	0	x	x	Generate DAC1/2 watchpoint. No additional action.
0	0	1	0	x	Set DBSR[DAC1/2], DBSR[IDE]. Generate DAC1/2 watchpoint. Multiple DBSR bits may be set for imprecise debug events for various operations. Table 9-20 lists possible combinations of concurrent imprecise debug events for all operations.
0	0	1	1	x	Issue debug interrupt. Set DBSR[DAC1/2]. Generate DAC1/2 watchpoint

Table 9-20 lists the combinations of debug events that may be simultaneously recorded in the DBSR (DBCR0[EDM] = 0, DBCR0[IDM] = 1) when debug interrupts are disabled (MSR[DE] = 0) for every operation type. Multiple debug events can be recorded as a result of a single operation if the DBSR[IDE] is set for that operation.

Table 9-20. Recording of Imprecise Debug Events (IDEs)

Operation That Occurs when IDE Events are Possible (DBCR0[EDM] = 0, DBCR0[IDM] = 1, MSR[DE] = 0)	Possible Simultaneous Imprecise Debug Events [1]							
	UDE	IRPT	TRAP	IAC1/2	DAC1/2	RET	CIRPT	CRET
Interrupt Operations								
Machine check interrupt	—	—	—	—	—	—	—	—
Unconditional debug event	Yes	—	—	—	—	—	—	—
Asynchronous critical input interrupt	—	—	—	—	—	—	Yes	—
Watchdog timer interrupt	—	—	—	—	—	—	Yes	—
Asynchronous external input interrupt	—	Yes	—	—	—	—	—	—
Instruction TLB error interrupt	—	Yes	—	Yes	—	—	—	—
Instruction storage (ISI) interrupt	—	Yes	—	Yes	—	—	—	—
Floating point unavailable interrupt	—	Yes	—	Yes	—	—	—	—
Program Interrupt	—	Yes	Yes[2]	Yes	—	—	—	—
Data TLB error interrupt	—	Yes	—	Yes	—	—	—	—
Data storage (DSI) interrupt	—	Yes	—	Yes	—	—	—	—
Alignment interrupt	—	Yes	—	Yes	—	—	—	—
System call interrupt	—	Yes	—	Yes	—	—	—	—
Fixed interval timer interrupt	—	Yes	—	—	—	—	—	—

Table 9-20. Recording of Imprecise Debug Events (IDEs) (continued)

Operation That Occurs when IDE Events are Possible (DBCR0[EDM] = 0, DBCR0[IDM] = 1, MSR[DE] = 0)	Possible Simultaneous Imprecise Debug Events [1]							
	UDE	IRPT	TRAP	IAC1/2	DAC1/2	RET	CIRPT	CRET
Performance monitor interrupt	—	Yes	—	—	—	—	—	—
Decrementer interrupt	—	Yes	—	—	—	—	—	—
Instruction Complete Operations								
rfi instruction	—	—	—	Yes	—	Yes	—	—
rfci instruction	—	—	—	Yes	—	—	—	Yes
Load, store, or cache management instruction	—	—	—	Yes	Yes	—	—	—
All other instructions	—	—	—	Yes	—	—	—	—

[1] Because debug exceptions do not occur when DE = 0, multiple debug events can be recorded in the DBSR as a result of a operation (along with the setting of DBSR[IDE]). All debug events marked “yes” are recorded if they occur for that operation.

[2] TRAP is set if the program interrupt is caused by a trap exception.

9.8.4 Delayed Debug Interrupts

Delayed debug interrupts on the e500mc core can be taken under one of the following circumstances:

- An **mtmsr** instruction that sets MSR[DE] = 1 and any DBSR bit is a one (including the IDE field, but excluding the MRR field). In this case, DSRR0 holds the address of the instruction following the **mtmsr**.
- Any return from interrupt class (**rfi**, **rfci, rfdi**, **rfmci)** instruction sets MSR[DE] = 1 and any DBSR bit is a one (including the IDE field, but excluding the MRR field). In this case, DSRR0 holds the address of the target of the return from interrupt instruction.

The e500mc uses DBCR0[IDM] to enable/disable recognition of debug events, and it uses MSR[DE] to enable/disable taking debug interrupts when debug events are recognized. When a debug event is recognized, the event is logged in DBSR and, if debug interrupts are enabled, a debug interrupt also occurs.

A delayed debug interrupt is a delayed response to a previously logged event. Although DBCR0[IDM] is a condition for recognizing and logging a debug event, it is not a condition for taking a delayed debug interrupt. This is different from some previous versions of e500, which required IDM = 1 in order to take a delayed debug interrupt.

9.8.5 Instruction Address Compare Debug Events

The core implements IAC debug events as described in the architecture, with the following exceptions and clarifications:

- Only IAC1 and IAC2 are supported. IAC3 and IAC4 are not supported.
- Real Mode comparisons (DBCR1[IAC1ER] = 01 and DBCR1[IAC2ER] = 01) are not supported.

One or more instruction address compare debug conditions (IAC1, IAC2) occur if they are enabled and execution is attempted of an instruction at an address that meets the criteria specified in DBCR0, DBCR1,

IAC1, and IAC2. These debug conditions cause debug events to be recorded in DBSR if MSR[DE] = 1 and no higher priority exception exists or according to Table 9-20 if MSR[DE] = 0. When MSR[DE] = 1, the IAC debug conditions are logged when the debug IAC1/2 interrupt is taken. When MSR[DE] = 0, IAC debug conditions are recorded in DBSR when an instruction marked with an IAC1/2 condition takes an interrupt or completes, according to Table 9-20. MSR[DE] has no effect on the updates to EDBSR0.

Instruction address compares may specify user/supervisor mode and instruction space (MSR[IS]), along with an effective address, masked effective address, or range of effective addresses for comparison. Refer to Section 2.17.3, "Debug Control Register 1 (DBCR1)," for details on the controls for the various IAC event modes.

IAC conditions are masked from generating IAC events if DBCR2[DACLINK1/2] are set. The IAC fields of DBSR and EDBSR0 are not updated. In this case, a DAC event occurs if an instruction generates both a DAC condition and an IAC condition and no exceptions of higher priority are present.

In EDM, an unmasked IAC debug condition is recorded as a debug event in EDBSR0[IAC1, IAC2], the execution of the instruction causing the debug event is suppressed, the processor halts, and NIA is set to the address of the excepting instruction.

In IDM, an unmasked IAC debug condition is recorded as a debug event in DBSR[IAC1, IAC2] if MSR[DE] = 1 and no higher priority exception exists, or according to Table 9-20, if MSR[DE] = 0. More than one bit will be set in DBSR if the instruction address compare mode is not exact address compare mode as DBSR bits corresponding to IAC1 and IAC2 will be set.

If debug interrupts are enabled (MSR[DE] = 1) and the debug event is recorded, a debug interrupt is generated, the execution of the instruction causing the debug event is suppressed, and DSRR0 is set to the address of the excepting instruction.

If debug interrupts are disabled (MSR[DE] = 0), the IAC event is conditionally recorded in the DBSR according to the type of operation associated with the event. See Table 9-20 for a complete list of operations and their effect on the recording of imprecise debug events as well as which imprecise debug events can be simultaneously recorded for a given operation. If the IAC event is recorded in the DBSR, DBSR[IDE] is also set to indicate that the debug interrupt (if later enabled) is an imprecise event. In the case of a delayed debug interrupt, the DSRR0 contains the address of the instruction following the one that enabled debug interrupts. Software in the debug interrupt handler can use the DBSR[IDE] information to determine how to interpret the contents of the DSRR0.

9.8.6 Data Address Compare Debug Events

e500mc implements DAC debug events as described in the architecture, with the following exceptions and clarifications.

- Real address comparisons (DBCR2[DAC1ER] = 01 and DBCR2[DAC2ER] = 01) are not supported.
- All load instructions are considered reads with respect to debug conditions, while all store instructions are considered writes with respect to debug conditions.
- When MSR[GS] = 0, the value of EPCR[DUVD] is used to suppress debug DAC events when external PID instructions are used, even if the external PID instructions target a context a context

where GS = 1. Refer to the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors* for details.

One or more data address compare debug conditions (DAC1R, DAC1W, DAC2R, DAC2W) occur if they are enabled, execution is attempted of a data storage access instruction, and the type and address of the data storage access meet the criteria specified in the DBCR0, DBCR2, DAC1, and DAC2. These conditions cause debug events to be recorded in DBSR if MSR[DE] = 1 and no higher priority exception exists, or according to Table 9-20, if MSR[DE] = 0. MSR[DE] has no effect on the updates to EDBSR0.

Data address compares may specify user/supervisor mode and data space (MSR[DS]), along with an effective address, masked effective address, or range of effective addresses for comparison. Refer to Section 2.17.4, "Debug Control Register 2 (DBCR2)," for details on the controls for the various DAC event modes.

DBCR0[DAC1] determines whether DAC1 comparisons are performed on read-type accesses, write-type accesses, or both. Similarly, DBCR0[DAC2] determines if DAC2 comparisons are performed on read-type accesses, write-type accesses, or both.

All load instructions are considered reads with respect to debug conditions, while all store instructions are considered writes with respect to debug conditions. In addition, the cache management instructions and certain special cases are handled as follows:

dcbt[**ls**], **dcbtst**, **dcbtep**, **dcbtstep, icbt**[**ls**], **icbi**, **icbiep**, and **icblc** are all considered reads with respect to debug events. Note that **dcbt**[**ep**], **dcbtst**[**ep**], and **icbt** are treated as NOPs when they report data storage or data TLB miss exceptions, instead of being allowed to cause interrupts. However, these instructions cause debug interrupts, even when they would otherwise have been NOPed due to a data storage or data TLB miss exception.

dcbz[**ep**], **dcbi**, **dcbf**[**ep**], **dcba**, **dcbst**[**ep**], **dcbtstls**, and **dcblc** are all considered writes with respect to debug events. Note that **dcbf** and **dcbst** are considered reads with respect to data storage exceptions, because they do not actually change the data at a given address. However, because execution of these instructions may result in write activity on the processor's data bus, they are treated as writes with respect to debug events. See Table 4-2 for the list of exceptions for all load, store, and cache management instructions.

lmw or **stmw** operations may partially complete if a DAC event occurs after the initial transfer has started. DAC events may be further qualified by requiring an IAC condition on the corresponding data storage access instruction by setting DBCR2[DACLINK1/2]. When DACs are linked to IACs in this way, a DAC event occurs only if an instruction generates both a DAC condition and an IAC condition (IAC1 or IAC2 debug condition). These linked events are recorded in DBSR[DAC1,DAC2], according to which DAC comparator generated the debug condition. For e500mc, a DACLINK1/2 event will only occur if the DAC condition matches the first word of a **lmw** or **stmw** instruction.

In EDM, if no higher priority exception is associated with the instruction, a DAC debug condition is recorded as a debug event in EDBSR0[DAC1R, DAC1W, DAC2R, DAC2W], the execution of the instruction causing the debug event is suppressed, the processor halts, and NIA is set to the address of the excepting instruction.

In IDM, a DAC debug condition is recorded as a debug event in DBSR[DAC1R,DAC1W,DAC2R, DAC2W] if MSR[DE] = 1 and no higher priority exception exists, or according to Table 9-20, if

MSR[DE] = 0. More than one bit will be set in DBSR if the data address compare mode is not exact address compare mode as DBSR bits corresponding to DAC1 and DAC2 will be set.

If debug interrupts are enabled (MSR[DE] = 1) and the debug event is recorded, a debug interrupt is generated, the execution of the instruction causing the debug condition is suppressed, and DSRR0 is set to the address of the excepting instruction.

If debug interrupts are disabled (MSR[DE] = 0), the DAC event is conditionally recorded in the DBSR according to the type of operation associated with the event. See Table 9-20 for a complete list of operations and their effect on the recording of imprecise debug events as well as which imprecise debug events can be simultaneously recorded for given operation. If the DAC event is recorded in the DBSR, DBSR[IDE] is also set to indicate that the debug interrupt (if later enabled) is an imprecise event. In the case of a delayed debug interrupt, the DSRR0 contains the address of the instruction following the one that enabled debug interrupts. Software in the debug interrupt handler can use the DBSR[IDE] information to determine how to interpret the contents of the DSRR0.

9.8.7 Trap Debug Event

A trap debug condition occurs if trap debug conditions are enabled (DBCR0[TRAP] = 1), a Trap instruction (**tw**, **twi**) is executed, and the conditions specified by the instruction for the trap are met. This condition causes the corresponding debug event to be recorded in DBSR if MSR[DE] = 1 and no higher priority exception exists, or according to Table 9-20, if MSR[DE] = 0. When MSR[DE] = 1, the trap debug condition is recorded in DBSR when the debug trap interrupt is taken. When MSR[DE] = 0, the trap debug condition is recorded in DBSR when the program trap interrupt is taken. MSR[DE] has no effect on the updates to EDBSR0.

In EDM, if no higher priority exception exists, a trap debug condition is recorded as a debug event in EDBSR0[TRAP], the execution of the trap instruction is suppressed, the processor halts, and NIA (Next Instruction Address) is set to the address of the trap instruction.

In IDM, a trap debug condition is recorded as a debug event in DBSR[TRAP] if MSR[DE] = 1 and no higher priority exception exists, or according to Table 9-20, if MSR[DE] = 0.

If debug interrupts are enabled (MSR[DE] = 1) and the debug event is recorded, a debug interrupt is generated, the execution of the trap instruction is suppressed, and DSRR0 is set to the address of the trap instruction.

If debug interrupts are disabled (MSR[DE] = 0), the trap debug event is conditionally recorded in the DBSR according to the type of operation associated with the event. See Table 9-20 for a complete list of operations and their effect on the recording of imprecise debug events as well as which imprecise debug events can be simultaneously recorded for given operation. If the trap debug event is recorded in the DBSR, DBSR[IDE] is also set to indicate that the debug interrupt (if later enabled) is an imprecise event. In the case of a delayed debug interrupt, the DSRR0 contains the address of the instruction following the one that enabled debug interrupts. Software in the debug interrupt handler can use the DBSR[IDE] information to determine how to interpret the contents of the DSRR0.

9.8.8 Branch Taken Debug Event

A branch taken debug condition occurs if branch taken debug conditions are enabled (DBCR0[BRT] = 1) and execution is attempted of a branch instruction which is taken (either an unconditional branch, or a conditional branch whose branch condition is true).

In EDM, a branch taken debug condition is recorded as a debug event in EDBSR0[BRT], the execution of the branch instruction is suppressed, the processor halts, and NIA (Next Instruction Address) is set to the address of the branch instruction.

In IDM, a branch taken debug condition records a debug event in DBSR[BRT] if MSR[DE] = 1. A debug interrupt is generated, the execution of the branch instruction is suppressed, and DSRR0 is set to the address of the branch instruction. Branch taken debug events are not recognized if MSR[DE] = 0 at the time of execution of the branch instruction and thus DBSR[IDE] cannot be set by a branch taken debug event

9.8.9 Instruction Complete Debug Event

An instruction complete debug condition occurs if instruction complete debug conditions are enabled (DBCR0[ICMP] = 1) and execution of any instruction is completed.

If execution of an instruction is suppressed due to the instruction causing some other exception that is enabled to generate an interrupt, then the attempted execution of that instruction does not cause an instruction complete debug condition. The **sc** instruction does not fall into the category of an instruction whose execution is suppressed, because the instruction actually executes and then generates a system call interrupt. In this case, the instruction complete debug event is also set. If a debug interrupt does occur in this case, DSRR0 points to the first instruction in the system call interrupt handler. Note that, in general, instruction complete debug conditions do not occur for any instruction whose execution causes a exception whose interrupt would save the address of that instruction in the appropriate save/restore register 0. For example, a trap instruction which causes a trap exception would not create an instruction complete debug condition.

In EDM, an instruction complete debug condition is recorded as a debug event in EDBSR0[ICMP], the processor halts, and NIA (Next Instruction Address) is set to the address of the next instruction to be executed.

In IDM, an instruction complete debug condition records a debug event in DBSR[ICMP] if MSR[DE] = 1. Instruction complete debug events are not recognized if MSR[DE] = 0 at the time of execution of the instruction and thus DBSR[IDE] cannot be set by an instruction complete debug event.

Special consideration is given to instructions which enable or disable instruction complete debug events. If MSR[DE] = 1, DBCR0[IDM] = 1, DBCR0[EDM] = 0, and DBCR0[ICMP] = 1, and an **mtmsr** instruction which clears MSR[DE] completes, no instruction complete debug event (or interrupt) occurs. Conversely, if MSR[DE] = 0, DBCR0[IDM] = 1, DBCR0[ICMP] = 1, and an **mtmsr** instruction which sets MSR[DE] completes, an instruction complete debug event still occurs for the **mtmsr** itself.

An **mtspr** instruction which enables or disables instruction complete debug events by changing the state of DBCR0[ICMP] or DBCR0[IDM] (when all other necessary conditions for enabling the event are present), causes the enable/disable operation to be applied to the **mtspr** itself. Return from interrupt class

instructions which enable or disable instruction complete debug events through the side effect of a change to MSR[DE] is not applied to the return instruction itself, but takes effect on the next instruction following the return.

When an instruction complete debug event is recorded in internal debug mode, a debug interrupt is generated and the address of the next instruction to be executed is recorded in DSRR0.

9.8.10 Interrupt Taken Debug Event

An interrupt taken debug condition occurs if interrupt taken debug conditions are enabled (DBCR0[IRPT] = 1) and a non-debug, noncritical, non-machine check interrupt occurs. Only non-debug, noncritical, non-machine check class interrupts cause an interrupt taken debug condition. This condition is recorded in DBSR if MSR[DE] = 1, or according to Table 9-20 if MSR[DE] = 0. MSR[DE] has no effect on the updates to EDBSR0.

In EDM, an interrupt taken debug condition is recorded as a debug event in EDBSR0[IRPT], the processor halts, and NIA (Next Instruction Address) is set to the address of the noncritical interrupt handler. No instructions at the noncritical interrupt handler will have executed.

In IDM, an interrupt taken debug condition is recorded as a debug event in DBSR[IRPT] if MSR[DE] = 1, or according to Table 9-20 if MSR[DE] = 0. If debug interrupts are enabled (MSR[DE] = 1), a debug interrupt is generated and the value saved in DSRR0 is the address of the noncritical interrupt handler. No instructions at the noncritical interrupt handler will have executed.

If debug interrupts are disabled (MSR[DE] = 0), the IRPT debug event is conditionally recorded in the DBSR according to the type of operation associated with the event. See Table 9-20 for a complete list of operations and their effect on the recording of imprecise debug events as well as which imprecise debug events can be simultaneously recorded for given operation. If the IRPT debug event is recorded in the DBSR, DBSR[IDE] is also set to indicate that the debug interrupt (if later enabled) is an imprecise event. In the case of a delayed debug interrupt, the DSRR0 contains the address of the instruction following the one that enabled debug interrupts. Software in the debug interrupt handler can use the DBSR[IDE] information to determine how to interpret the contents of the DSRR0.

9.8.11 Interrupt Return Debug Event

A return debug condition occurs if return debug conditions are enabled (DBCR0[RET] = 1) and an attempt is made to execute an **rfi** instruction and no other higher priority exception occurs executing the **rfi**. This condition causes the corresponding debug event to be recorded in DBSR if MSR[DE] = 1, or according to Table 9-20 if MSR[DE] = 0. MSR[DE] has no effect on the updates to EDBSR0.

In EDM, a return debug condition is recorded as a debug event in EDBSR0[RET], execution of the **rfi** is suppressed, the processor halts, and NIA (Next Instruction Address) is set to the address of the **rfi** instruction.

In IDM, a return debug condition is recorded as a debug event in DBSR[RET] if MSR[DE] = 1 and no higher priority exception exists, or according to Table 9-20 if MSR[DE] = 0. If debug interrupts are enabled (MSR[DE] = 1), a debug interrupt occurs provided there exists no higher priority exception which is enabled to cause an interrupt. The DSRR0 is set to the address of the **rfi** instruction.

If debug interrupts are disabled (MSR[DE] = 0) at the time of the execution of the **rfi** (that is, before the MSR is updated by the **rfi**), the RET event is conditionally recorded in the DBSR according to the type of operation associated with the event. See Table 9-20 for a complete list of operations and their effect on the recording of imprecise debug events as well as which imprecise debug events can be simultaneously recorded for given operation. If the RET debug event is recorded in the DBSR, DBSR[IDE] is also set to 1 to record the imprecise debug event. In the case of a delayed debug interrupt, the DSRR0 contains the address of the instruction following the one that enabled debug interrupts. Software in the debug interrupt handler can use the DBSR[IDE] information to determine how to interpret the contents of the DSRR0.

9.8.12 Unconditional Debug Event

An unconditional debug condition occurs when the unconditional debug event (UDE) input transitions to the asserted state, and either DBCR0[IDM] = 1 or DBCR0[EDM] = 1. The unconditional debug condition is the only debug condition which does not have a corresponding enable bit for the condition in DBCR0. This condition causes the corresponding debug event to be recorded in DBSR or EDBSR0 regardless of the setting of MSR[DE].

In EDM, upon the rising edge of the UDE input, an unconditional debug condition is recorded as a debug event in EDBSR0[UDE], the processor halts, and NIA (Next Instruction Address) is set to the address of the next instruction to be executed.

In IDM, upon the rising edge of the UDE input, an unconditional debug condition is recorded as a debug event in DBSR[UDE]. If debug interrupts are enabled (MSR[DE] = 1), a debug interrupt occurs in response to the unconditional debug event and the DSRR0 is set to the address of the instruction that would be executed next were it not for the occurrence of the debug interrupt. DBSR[IDE] is always set regardless of the state of MSR[DE] when an unconditional debug event occurs.

If MSR[DE] = 0 when an unconditional debug condition occurs, the condition is recorded as an event in the DBSR[UDE] In the case of a delayed debug interrupt, the DSRR0 contains the address of the instruction following the one that enabled debug interrupts.

9.8.13 Critical Interrupt Taken Debug Event

A critical interrupt taken debug condition occurs if critical interrupt taken debug conditions are enabled (DBCR0[CIRPT] = 1) and a critical interrupt occurs. Only critical class interrupts cause a critical interrupt taken debug condition. This condition causes the corresponding debug event to be recorded in DBSR if MSR[DE] = 1, or according to Table 9-20, if MSR[DE] = 0. MSR[DE] has no effect on the updates to EDBSR0.

In EDM, a critical interrupt taken debug condition is recorded as a debug event in EDBSR0, the processor halts, and NIA (Next Instruction Address) is set to the address of the critical interrupt handler. No instructions at the critical interrupt handler will have executed.

In IDM, a critical interrupt taken debug condition is recorded as a debug event in DBSR[CIRPT], if MSR[DE] = 1, or according to Table 9-20, if MSR[DE] = 0. If debug interrupts are enabled (MSR[DE] = 1), a debug interrupt occurs, provided there exists no higher priority exception which is enabled to cause an interrupt, and the value saved in DSRR0 is the address of the critical interrupt handler. No instructions at the critical interrupt handler will have executed.

If debug interrupts are disabled (MSR[DE] = 0), the CIRPT debug event is conditionally recorded in the DBSR according to the type of operation associated with the event. See Table 9-20 for a complete list of operations and their effect on the recording of imprecise debug events as well as which imprecise debug events can be simultaneously recorded for given operation. If the CIRPT debug event is recorded in the DBSR, DBSR[IDE] is also set to indicate that the debug interrupt (if later enabled) is an imprecise event. In the case of a delayed debug interrupt, the DSRR0 contains the address of the instruction following the one that enabled debug interrupts. Software in the debug interrupt handler can use the DBSR[IDE] information to determine how to interpret the contents of the DSRR0.

9.8.14 Critical Return Debug Event

A critical return debug condition occurs if critical return debug conditions are enabled (DBCR0[CRET] = 1) and an attempt is made to execute an **rfci** instruction and no other higher priority exception occurs executing the **rfci**. This condition causes the corresponding debug event to be recorded in DBSR if MSR[DE] = 1, or according to Table 9-20, if MSR[DE] = 0. MSR[DE] has no effect on the updates to EDBSR0.

In EDM, a critical return debug condition is recorded as a debug event in EDBSR0, execution of the **rfci** is suppressed, the processor halts, and NIA (Next Instruction Address) is set to the address of the **rfci** instruction.

In IDM, a critical return debug condition is recorded as a debug event in DBSR[CRET] if MSR[DE] = 1 and no higher priority exception exists, or according to Table 9-20 if MSR[DE] = 0. If debug interrupts are enabled (MSR[DE] = 1), a debug interrupt occurs, provided there exists no higher priority exception which is enabled to cause an interrupt. The DSRR0 is set to the address of the **rfci** instruction.

If debug interrupts are disabled (MSR[DE] = 0) at the time of the execution of the **rfci** (that is, before the MSR is updated by the **rfci**), the CRET event is conditionally recorded in the DBSR according to the type of operation associated with the event. See Table 9-20 for a complete list of operations and their effect on the recording of imprecise debug events as well as which imprecise debug events can be simultaneously recorded for given operation. If the CRET debug event is recorded in the DBSR, then DBSR[IDE] is also set to 1 to record the imprecise debug event. In the case of a delayed debug interrupt, the DSRR0 contains the address of the instruction following the one that enabled debug interrupts. Software in the debug interrupt handler can use the DBSR[IDE] information to determine how to interpret the DSRR0 contents.

9.9 External Debug Interface

External debug support is supplied through a memory mapped interface which allows access to internal cpu registers, arrays and other system state while the core is halted. EDM provides the ability to enter the halt state when a debug event occurs. This capability can be used to perform singlestep operations from the external debug tool.

9.9.1 Processor Run States

This section discusses the following processor run states:

- Section 9.9.1.1, “Halt”

- Section 9.9.1.2, “Stop (Freeze)”
- Section 9.9.1.3, “Wait”
- Section 9.9.1.4, “Entering/Exiting Processor Run States”

9.9.1.1 Halt

When the e500mc is in the halted state, the clocks are still running, but the core is not fetching or executing instructions. While in this state, an external debugger can jam instructions into the pipeline, and they are executed. The core also continues to snoop the core complex bus and maintains cache coherency.

Assertion of *pm_halt* causes the core to enter the halted state. PRSR[PM_HALT] is asserted to indicate that *pm_halt* has been asserted, and PRSR[HALTED] indicates that the core is in the halted state. When *pm_halt* is deasserted, PRSR[PM_HALT] transitions to zero and, if the processor has not also been halted for a halt condition in the debug class, the core resumes immediately.

There are several mechanisms that halt the core. These are described in Table 9-21.

Table 9-21. Methods for Halting the Core

Halt Condition	Classification	Enable	Documentation
Assertion of *pm_halt*	Power Management	Always enabled	—
Assertion of *core_dbg_halt*	Debug	Always enabled	—
DNH	Debug	EDBCR0[DNH_EN]	Section 9.9.3, “Debugger Notify Halt (dnh) Instruction”
DE	Debug	EDBCR0[EDM]	Section 9.8.2, “Internal and External Debug Modes”

Most external debug operations can only be performed when the processor is halted. Note that if the core is halted only because *pm_halt* is asserted (that is, no other halt requests are active in PRSR), it resumes immediately if *pm_halt* is deasserted. therefore, the core should always be halted with some other debug mechanism (for example, setting a system debug event halt) before accessing the contents of the core.

The Processor Run Status Register (PRSR) indicates whether or not the core is halted for debug.

9.9.1.2 Stop (Freeze)

When the e500mc is in the stopped state, the clocks are stopped. The caches are not snooped. If the clocks are stopped while the caches contain modified data, coherency may be lost because other processors (or other bus masters) do not see the modified data. Coherency may also be lost if the clocks are stopped while the caches contain shared or exclusive data, then restarted. In this case, other processors may have changed the data, but the stopped processor retains the stale data, which may be used when the processor is restarted.

Assertion of *core_stop* causes to core to enter the stopped state. PRSR[PM_STOP] is asserted to indicate that *core_stop* has been asserted, and PRSR[STOPPED] indicates that the core is in the stopped state. When *core_stop* is deasserted, PRSR[PM_STOP] transitions to zero and, if the processor has not also been stopped for a stop condition in the debug class, the core transitions immediately to the appropriate halted or running state. The mechanism that can stop the core.

Table 9-22. Methods for Stopping the Core

Stop Condition	Classification	Enable	Documentation
Assertion of *core_stop*	Power Management	Always enabled	Section 8.3, "Power Management Signals

9.9.1.3 Wait

When the processor executes the **wait** instruction, it discontinues fetching and executing instructions, and waits for an asynchronous interrupt. This is the program wait state. This state does not have any effect on the processor while it is in the debug halted state, but affects resuming from the halted state. If the processor is in the program wait state when the *core_resume* signal is asserted to exit the halted state, the core does not fetch or execute any instructions until an asynchronous interrupt occurs. Otherwise, it begins fetching and executing instructions immediately.

If the processor is in the program wait state when the debug halted state is entered, the processor remains in the program wait state. Jamming an **mtspr** to the NIA causes the processor to exit the program wait state. Jamming a **wait** instruction causes the processor to enter the program wait state.

The debugger can examine PRSR[WAIT] to determine whether or not the processor is in the program wait state.

9.9.1.4 Entering/Exiting Processor Run States

The e500mc core classifies halt and stop conditions into two categories: power management and debug. These categories are distinguished by the steps that are required to exit the halted or stopped state. This is done to avoid undesired interactions that could occur when *pm_halt* or *core_stop* is toggled while the processor is under control of a debugger.

Debug operations should not be performed while the core is halted/stopped only due to power management. If the core has been halted or stopped only for power management, the debugger should assert *core_dbg_halt* before executing debug operations.

When the core is running, the SoC should use the following sequence to enter the power management stopped state:

1. Assert *pm_halt.*
2. Wait for *core_halted* to be asserted by the core.
3. Assert *core_stop.*

This ensures that the core is left in a recoverable state when the clocks are stopped. For the e500mc, when *pm_halt* and *core_stop* are asserted simultaneously, the processor first halts, and then stops.

The processor can transition directly from any of the three possible states (running, halted, or stopped) to any other of the three states.

Assume that the processor has been halted by one of the halt conditions in the debug class. To resume from this state, the debugger must:

1. Clear all of the bits in PRSR that correspond to halt requests in the debug class.

2. Assert *core_resume*.

Similarly, assume that the processor has been stopped by one of the stop conditions in the debug class. To resume from this state, the debugger must:

1. Clear all of the bits in PRSR that correspond to stop requests in the debug class,
2. Assert *core_resume*.

Normally, when the processor has been halted for power management by asserting *pm_halt*, the processor resumes execution when *pm_halt* is deasserted. Similarly, the processor normally exits the power management stopped state whenever *core_stop* is deasserted. However, if the core has been halted or stopped for a halt or stop condition in the debug class, deassertion of *pm_halt* or *core_stop* do not cause the processor to resume until *core_resume* is asserted.

If *core_resume* is asserted while *pm_halt* or *core_stop* is asserted, the core remains in the halted or stopped for power management state.

If any of the debug related halt status bits are set in the PRSR indicating whether or not the core has been halted or stopped for a debug condition, *core_resume* must be asserted before the core resumes execution.

If the core has been halted or stopped only by assertion of *pm_halt* or *core_stop*, simply releasing *pm_halt* or *core_stop* allows the processor to resume execution.

If the core is in the stopped state, and some halt requests are active in PRSR, then an attempt to resume causes the processor to go directly from the stopped to the halted state. If no halt requests are active, the processor goes directly from the stopped to the running state.

In order to be able to resume from a stopped state, special steps must be taken when stopping the core. These steps are:

1. Flush the caches so that they do not contain any modified data. This prevents coherency problems.
2. Discontinue any snoop traffic.
3. Halt the core
4. Stop the core

9.9.2 Singlestep

An external development tool can singlestep through code using the instruction complete (ICMP), interrupt taken (IRPT) and critical interrupt taken (CIRPT) debug events in EDM. If a resume command is issued while the ICMP, IRPT, and CIRPT events are enabled in EDM, the processor does one of the following:

- Execute and complete one instruction, then halt before executing the next instruction.
- Execute one instruction and take a synchronous interrupt, then halt before executing the first instruction of the interrupt handler.
- Immediately take an asynchronous interrupt and halt on the first instruction of the interrupt handler.

therefore, to single step, set ICMP and IRPT and CIRPT, set EDM, clear PRSR and resume. Note that PRSR must be cleared prior to each resume command.

9.9.3 Debugger Notify Halt (dnh) Instruction

The **dnh** instruction (see Section 3.5, "Debug Instruction Model") provides a mechanism to halt the processor independent of the state of DBCR0[EDM]. It is enabled by setting EDBCR0[DNH_EN] = 1. This instruction is not privileged and may be executed while the processor is at any privilege level. It may be compiled into code during debug, or an external debugger may substitute **dnh** for another instruction at a location where a breakpoint is desired.

Execution of this instruction when EDBCR0[DNH_EN] = 1 causes the processor to halt. The NIA is set to the address of the **dnh** instruction, the 5-bit DUI operand is captured into PRSR[DNHM] and PRSR[DNH_HALT] is set. PRSR[DNHM] can provide an external debugger information about the breakpoint that was hit. For example, it could uniquely identify which breakpoint was hit.

The 10-bit DUIS field is an extension to the DUI field. It is not captured into any architecture-defined register and can only be acquired by reading the opcode. When the processor halts due to execution of a **dnh**, the NIA can be used to locate instruction and read the DUIS field.

Execution of the **dnh** instruction when EDBCR0[DNH_EN] = 0 causes an illegal instruction exception.

Software may be instrumented to include **dnh** instructions in order to transfer control to an external development tool at designated points for interactive debugging. The **dnh** instruction is useful for debugging debug interrupt service routines (IVOR15). Without the **dnh** instruction, it is difficult to halt within the debug interrupt routines, because the machine must be in internal debug mode to enter the routine but must be in EDM to halt on a debug exception. Because **dnh** is enabled with EDBCR0[DNH_EN] instead of DBCR0[EDM], it provides a way to halt within the debug interrupt service routine.

9.9.4 Resource Access

Memory mapped access is provided for debug resources. In addition, a subset of these resources (Nexus Trace) is accessible via software SPRs (using **mtspr/mfspr** instructions). The resources access methods areas follows:

- Instruction jamming (memory mapped)
  - — Access to architecture-defined registers, including GPRs, SPRs, and PMRs.
  - — Access to memory-mapped resources with and without MMU translations
- Storage access through memory mapped interface
  - — Direct access to a few architecture-defined registers
  - — Implementation-dependent access to arrays within the core
  - — Direct access to memory

9.9.4.1 Memory Mapped Access

Addressing the debug/expert resources through the memory mapped interface entails driving a base address for the e500mc core (BASE), a functional group select (GID) and register index for a specific register. The functional group select determines what class of resource is to be accessed, while the register index determines which resource within the group to access.

Refer to the SoC reference manual for specifics on accessing the internal memory mapped resources via the external JTAG interface as well as the Aurora high speed serial interface.

Figure 9-18 shows the address bit fields used in accessing debug/expert resources and Table 9-23 summarizes the debug/expert resource memory map.

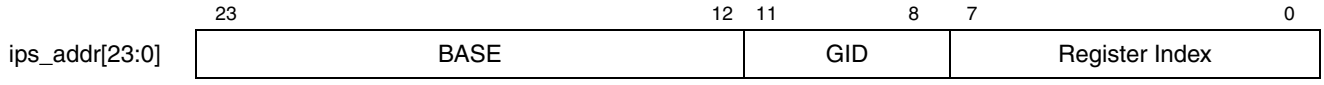


Figure 9-18. Debug/Expert Resource Access

Table 9-23. Debug/Expert Resource Address Map

Functional Group ID ips_addr[11:8]	Register Index ips_addr[7:0]	Resource	Access Type	Access Restrictions[1]	Service Data Width	Reset Source
0x0 Debug Status	0x00	Processor Run Status Register (PRSR)	R/W[2]	E	32	POR
	0x04	Reserved				
	0x08	Machine State Register (MSR)	R/W	E, H	32	HRESET
	0x0c	External Debug Status Register (EDBSR0)	R	E	32	POR
	0x10	External Debug Status Register (EDBSR1)	R	E	32	POR
	0x14	External Debug Exception Syndrome Register (EDESR)	R	E	32	POR
	0x18	Processor Version Register (PVR)	R		32	N/A
	0x1c	External Debug Status Register Mask 0 (EDBSRMSK0)	R/W	E	32	POR
	0x20 - 0x28	Reserved				
	0x2c	Program Counter Capture Register (PCC)	R		32	POR
	0x30	Perfmon Capture Count Register 0 (PMCC0)	R		32	POR
	0x34	Perfmon Capture Count Register 1 (PMCC1)	R		32	POR
	0x38	Perfmon Capture Count Register 2 (PMCC2)	R		32	POR
	0x3c	Perfmon Capture Count Register 3 (PMCC3)	R		32	POR
	0x40-0xfc	Reserved				
0x1 Debug Control	0x00	External Debug Control Register 0 (EDBCR0)	R/W	E	32	POR
	0x04 - 0x1c	Reserved				
	0x24 - 0xfc	Reserved				

Table 9-23. Debug/Expert Resource Address Map (continued)

Functional Group ID ips_addr[11:8]	Register Index ips_addr[7:0]	Resource	Access Type	Access Restrictions[1]	Service Data Width	Reset Source
0x3 Clock Control/Status	0x00	PLL Control Register 0 (PLL_CTRL0)	R/W	?	32	POR
	0x04	PLL Control Register 1 (PLL_CTRL1)	R/W	?	32	POR
	0x08 - 0x2c	Reserved				
	0x30	WAITR Configuration Register 0 (WAITR_CFG0)	R/W	?	32	POR
	0x34	WAITR_CFG1	R/W	?	32	POR
	0x38	WAITF_CFG0	R/W	?	32	POR
	0x3c	WAITF_CFG1	R/W	?	32	POR
	0x40	Test Mode Configuration Register	R/W	E?, H?	32	POR
	0x44 - 0x5c	Reserved				
	0x60	Silicon Debug Control Register 0 (SILCN_DBG_CTRL0)	R/W		32	POR
	0x64	Silicon Clock Control Register 0 (SILCN_CLK_CTRL0)	R/W		32	POR
	0x68	Silicon Clock Control Register 1 (SILCN_CLK_CTRL1)	R/W		32	POR
	0x6c	Silicon Clock Control Register 2 (SILCN_CLK_CTRL2)	R/W		32	POR
	0x70 - 0xfc	Reserved				
0x4 Nexus	0x00 - 0x04	Reserved				
	0x08	Nexus Development Control Register 1 (DC1)	R/W		32	POR
	0x0c	Nexus Development Control Register 2 (DC2)	R/W		32	POR
	0x10	Reserved				
	0x14	Nexus Development Control Register 4 (DC4)	R/W		32	POR
	0x18 - 0x28	Reserved				
	0x2c	Watchpoint Trigger Register 1 (WT1)	R/W		32	POR
	0x30 - 0x54	Reserved				
	0x58	Watchpoint Mask Register (WMSK)	R/W		32	POR
	0x5c	Nexus Overrun Control Register (OVCR)	R/W		32	POR
	0x60 - 0xfc	Reserved				
0x5	0x00 - 0xfc	Reserved				
0x6 Instruction Jamming	0x00	Instruction Jamming Data Register 0 (IJDATA0)	R/W	E, H	32	POR
	0x04	Instruction Jamming Data Register 1 (IJDATA1)	R/W	E, H	32	POR
	0x08	Instruction Jamming Configuration Register (IJCFG)	R/W	E, H	32	POR
	0x0c	Instruction Jamming Instruction Register (IJIR)	R/W	E, H	32	POR
	0x10 - 0xfc	Reserved				

Table 9-23. Debug/Expert Resource Address Map (continued)

Functional Group ID ips_addr[11:8]	Register Index ips_addr[7:0]	Resource	Access Type	Access Restrictions[1]	Service Data Width	Reset Source
0x7 - 0x9	Reserved	Reserved				
0xb - 0xd	Reserved					
0xe - 0xf	Reserved	Reserved for LBIST (not currently implemented)				

[1] If there are no restrictions listed, then software can access the corresponding resource through the memory mapped interface

E = External debugger access only

H = Access allowed when core is halted only

[2] Portions of PRSR support write-1-to-clear. All other fields are read-only.

9.9.4.2 Special-Purpose Register Access (Nexus Only)

Nexus trace resources can also be accessed through e500mc SPRs—specifically the Nexus SPR Access Configuration Register (NSPC) and Nexus SPR Access Data Register (NSPD).

Both read and write accesses are initiated by writing to NSPC via an **mtspr** instruction with the appropriate settings for the desired register index. The register index is identical to that used in accessing the resources through the memory map. For information about access, refer to Section 2.17.9, "Nexus SPR Access Registers."

Once the specific Nexus resource has been selected, software can then access the Nexus SPR Access Data Register (NSPD) by executing an **mtspr** instruction (for register writes) or an **mfspr** (for register reads).

Most registers require the processor to be halted in order to perform an access. Some registers may be accessed while the processor is running, but may also be updated by hardware. The values of these registers continues to change after their values are acquired by the debugger.

9.9.5 Instruction Jamming

Instruction jamming provides a generalized mechanism to perform debug operations using the existing facilities of the processor. When the processor is in a halted state, a development tool can jam instructions into the execution pipeline for the processor to execute.

Instruction jamming is useful for observing and altering the state of the machine whenever the processor is halted. Typical instruction jams include:

- **mfspr**—observe the value of an SPR
- **mtspr**—alter the value in an SPR
- load—observe the value of a memory location
- store—alter the value of a memory location
- Load or store from debug space—alter or observe the value of a GPR (see Section 9.9.5.1, "Debug Storage Space (IJCFG[IJMODE] = 1)")

Jammed instructions have no instruction address. therefore, they do not require translation of an instruction address, and there is no way to have an ITLB miss or ISI. Furthermore, a jammed instruction does not increment the NIA.

Jammed instructions can have undesired effects, particularly if the jammed instruction causes an exception. The processor provides some facilities that reduce the number of architectural registers that are affected by a jammed instruction that causes an exception. See Section 9.9.5.5, "Exception Conditions and Affected Architectural Registers," for details.

NOTE

Instruction jamming operations require the processor to be halted. Instruction jamming may change architecture-defined processor state. It is the responsibility of the external debug facility to save and restore any critical state.

9.9.5.1 Debug Storage Space (IJCFG[IJMODE] = 1)

Debug storage space is the conduit through which data is passed between an external debugger and the processor. From an external debugger's point of view, debug storage space is just part of the IJAM input or IJAM output, which are accessible through memory mapped access. From the processor's point of view, debug storage space is an alternate space that can be used as the source for loads or the destination for stores.

Debug storage space is accessed by load/store instructions when the IJMODE bit within the IJCFG register is 1. See Section 9.9.5.2, "Instruction Jamming Input," for a description of the IJAM input.

A debugger wishing to alter the value of a GPR would jam a load instruction. The debugger would place the desired load data in the IJAM IR register, and it set IJCFG[IJMODE] = 1 to specify that the load data should come from debug storage space (that is, from the IJAM input data in the IJDATA0/1 registers).

A debugger wishing to observe the value in a GPR would jam a store instruction. The debugger would set IJCFG[IJMODE] = 1 to specify that the store instruction should place its data into debug storage space (that is, send the IJAM output to the IJDATA0/1 registers). The debugger would then read the IJDATA0/1 registers to obtain the stored data.

Debug storage space is not part of the processor's memory address space. Although an effective address is calculated from the load or store instruction's operands, the address is not translated. therefore, there is no way to have a DTLB miss or DSI when jamming loads or stores to debug space. However, supplying operands that yield a nonzero effective address result in **unpredictable** results. therefore, the preferred form of loads and stores to debug space is the immediate form with RA = 0 and a displacement of 0x0.

The load/store instructions in Table 9-24 are supported when IJMODE = 1.

Table 9-24. Load/Store IJAM Transfers

Instruction	IJAM Transfer
lfd fr D, d(rA) **(lfs same syntax)**	IJDATA0[0–31] → FPR[0–31] IJDATA1[0–31] → FPR[32–63]
stfd fr S, d(rA) **(stfs same syntax)**	FPR[0–31] → IJDATA0[0–31] FPR[32–63] → IJDATA1[0–31]
stw rS,0(0)	GPR[32–63] → IJDATA1[0–31]
lwz rD,0(0)	IJDATA1[0–31] → GPR[32–63]
lhz rD,0(0)	IJDATA1[16–31] → GPR[48–63]; 16'b0 → GPR[32–47]
lbz rD,0(0)	IJDATA1[24–31] → GPR[56–63]; 24'b0 → GPR[32–55]

9.9.5.2 Instruction Jamming Input

Instructions to be jammed into the processor pipeline are transferred into the processor through accesses to memory mapped resources.

For all jammed instructions, the instruction jamming IR (IJIR) is required. This register contains the 32-bit Power instruction to be executed. When jamming load, store, or cache management (for example, **dcbf**) instructions, the IJCFG register is also required. This register drives the attributes of the load/store operation. When jamming load instructions, and IJCFG[IJMODE] indicates that the load data should be supplied from debug space, the IJDATA0/1 register(s) are required. They supply the data to be loaded. For more details refer '*Instruction Set Listings'* in '*EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors'*.

These registers can be accessed through a memory mapped access individually or through a block transfer. On e500mc, incorrect register settings result in **unpredictable** results. The IJAM register descriptions can be found in Section 9.5, "Instruction Jamming (IJAM) Registers."

The IJCFG register includes controls for jammed load and store instructions.

IJCFG[IJMODE] indicates whether jammed load/store instructions access memory or a special debug storage space. When IJMODE = 1, a jammed load instruction gets its data from IJDATA*n* registers, and a jammed store instruction writes its data to IJDATA0/1. When IJMODE = 0, load/store instructions access the cores memory address space as usual.

The effective/real bit, IJCFG[IJER] indicates whether load/store target addresses should be translated or not. Because debug storage space is not addressable, IJER is meaningful only when IJMODE = 0.

When IJCFG[IJER] = 1, load/store instructions do not have their effective addresses translated by the core's MMU. This means that the MMU does not supply a 36-bit physical address or page attributes (WIMGE bits) for the load/store instruction. Because the core only generates 32 bits of effective address, 4 more address bits are needed to form a 36-bit physical address. These additional 4 bits are supplied by

the IJRA field of IJCFG. The 36-bit address is formed by prepending the 4-bit IJRA field to the effective address calculated by the jammed load/store instruction (PA[28:63] = IJRA[0:3] || EA[32:63).

Because the WIMGE bits are not supplied by the MMU, they are supplied by the IJCFG[WIMGE] bits when IJER = 1. Care must be taken to specify the correct page attributes for a given real address so that cache paradoxes do not occur (that is, specifying a page attribute of cache-inhibited for a real address which has been previously accessed as cacheable may result in the load or store not accessing memory coherently with previous accesses or other processors or agents in the system).

When IJCFG[IJER] = 0, a data TLB miss error occurs if the MMU does not contain an entry that matches the virtual address. However, in real addressing mode, MMU translation is not performed and TLB miss errors do not occur.

Table 9-25. Instruction Jamming Addressing Modes

IJCFG[IJMODE]	IJCFG[IJER]	Page Attributes (LWIMGE)
0	0	Attributes taken from MMU
0	1	Attributes taken from IJCFG[WIMGE]
1	x	Don’t care. WIMGE attributes have no meaning when IJMODE=1

9.9.5.3 Supported Instruction Jamming Instructions

Table 9-26 lists instructions which are supported for instruction jamming (when the processor is in halt state). These instructions are executed in the same manner as if the processor were not halted when IJCFG is equal to 32’b0.

The table also includes all instructions that are capable of using options in IJCFG. All other instructions are not supported and will have unpredictable (UNPR) results if jammed. In addition, any instruction jammed with non zero values in IJCFG other than those explicitly listed as supporting them result in unpredictable outcomes.

Table 9-26. Implemented IJAM Instructions when the Processor Is Halted

Mnemonic	Description
dcbf	Data Cache Block Flush
dcbi	Data Cache Block Invalidate
dcblc	Data Cache Block Lock Clear
dcbst	Data Cache Block Store
dcbtls	Data Cache Block Touch and Lock Set
dcbtstls	Data Cache Block Touch for Store and Lock Set
dcbz	Data Cache Block Set to Zero
dcbzl	Data Cache Block Set to Zero
icbi	Instruction Cache Block Invalidate
icblc	Instruction Cache Block Lock Clear

Table 9-26. Implemented IJAM Instructions when the Processor Is Halted (continued)

Mnemonic	Description
icbtls	Instruction Cache Block Touch and Lock Set
lbz	Load Byte and Zero
lfd	Load Floating-Point Double
lfs	Load Floating-Point Single
lha	Load Halfword Algebraic
lhbrx	Load Halfword Byte-Reversed Indexed
lhz	Load Halfword and Zero
lwbrx	Load Word Byte-Reversed Indexed
lwz	Load Word and Zero
mfcr	Move from Condition Register
mffs[**.**]	Move from FPSCR
mfmsr	Move from Machine State Register
mfpmr	Move from PMR
mfspr	Move from SPR
mtcrf	Move to Condition Register Fields
mtfsf[**.**]	Move to FPSCR Fields
mtfsfi[**.**]	Move to FPSCR Field Immediate
mtmsr	Move to Machine State Register
mtpmr	Move to PMR
mtspr	Move to SPR
stb	Store Byte
stfd	Store Floating-Point Double
stfs	Store Floating-Point Single
sth	Store Halfword
sthbrx	Store Halfword Byte-Reversed Indexed
stw	Store Word
stwbrx	Store Word Byte-Reverse
sync	Sync. Only the form with **sync** L=0 is supported.
tlbre	TLB Read Entry
tlbsx	TLB Search Indexed
tlbwe	TLB Write Entry
wait	Wait

9.9.5.4 Instructions Supported only during Instruction Jamming

Table 9-27 lists instructions which are only supported when the processor is halted. These instructions produce an illegal instruction exception if attempted when the processor is not halted.

Table 9-27. Instructions Supported Only when the Processor is Halted

Mnemonic	SPRN	Behavior when Not Halted (Regardless of MSR[PR])	Behavior when Halted (Regardless of MSR[PR])	Comment
mfspr NIA	SPRN = 559	Illegal instruction exception	Executed	Move from SPR, NIA
mtspr NIA		Illegal instruction exception	Executed	Move to SPR, NIA

9.9.5.5 Exception Conditions and Affected Architectural Registers

Generally, jammed instructions are allowed to modify any architecture-defined register (such as GPRs, SPRs, MSR) in the processor. However, jamming an instruction that causes an exception condition can have undesired side effects. The processor has provided several special facilities to reduce these side effects. This reduces the debugger's burden to save and restore architectural state just in case an unanticipated exception occurs.

As previously mentioned, the NIA is not incremented when jammed instructions are executed. Furthermore, it is not updated to point to an interrupt vector if a jammed instruction causes an exception. therefore, the debugger does not have to save the state of the NIA when jamming instructions.

When in normal execution mode (that is, when not jamming), there are several cases when privileges must be observed or features must be enabled in order to avoid exception conditions. But when jamming instructions, the debugger is given full privileges so that it can avoid setting up architectural state necessary to execute a jammed instruction. In particular:

- MSR[PR] is effectively set to 0, giving the debugger access to all privileged instructions. therefore, program interrupt for privileged exceptions do not occur for jammed instructions.
- Read/Write privileges are enabled for all load/store instructions. therefore, data storage interrupts for read/write access control exceptions do not occur for jammed instructions. This is particularly useful when the debugger wishes to alter an instruction on a page and the translation attributes for that page do not include write access.
- DBCR0[IDM] is effectively cleared, preventing debug events from being recognized while jamming. therefore, DBSR is not updated and debug interrupts do not occur for jammed instructions.

In normal execution mode (that is, when not jamming), interrupts update save/restore registers and other various machine state. When jamming instructions, many of these registers are not updated if an exception

occurs. Table 9-28 lists some interesting architectural registers and indicates whether or not they are affected by an exception on a jammed instruction.

Table 9-28. Effect of Exceptions on Machine State

Register	Affected by Interrupt	Note
NIA	No	EDBSR1 indicates the IVOR number of the exception
SRR0, SRR1	No	—
CSRR0, CSRR1	No	—
DSRR0, DSRR1	No	—
MCSRR0, MCSRR1	No	—
ESR	No	EDESR contains the information usually captured in ESR
MSR	No	—
MCSR	Yes	—
MCAR	Yes	—
DEAR	No	—
MAS registers	No	—
DBSR	No	—

As Table 9-28 shows, the NIA is not updated when an exception occurs on a jammed instruction. Instead, EDBSR1 indicates the IVOR number of the exception that occurred. Similarly, the ESR is not updated, but the EDESR contains the information that would have been in the ESR if the exception had occurred in functional mode.

Data TLB misses are the most likely exceptions to occur on jammed instructions. They happen if no translation is available for a jammed load or store instruction. As can be seen in Table 9-28, the MAS registers and DEAR are not updated by a DTLB miss.

Asynchronous interrupts are always disabled when the processor is halted. therefore, asynchronous interrupts do not occur around the time that the processor is executing a jammed instruction.

9.9.5.6 Instruction Jamming Status

The status of instruction jamming operations is captured in the EDBSR1 register. In the event of an exception during instruction jamming, the instruction sequence is aborted.

The number of instructions that completed prior to the exception is recorded in EDBSR1[LCMP] (this is zero on e500mc). No interrupt is taken, but the IVOR number associated with the interrupt that would normally be taken is recorded in the EDBSR1[IVOR] and exception status is captured in External Debug Exception Syndrome Register (EDESR). EDESR is identical to its non-debug counterpart (the ESR) in terms of bit field definitions and provide information about the type of exception that occurred during instruction jamming.

Debug conditions are masked during instruction jamming and are not recorded. Effectively, DBCR0[IDM] = 0, so the DBSR does not log debug events.

The core should be halted for debug before jamming instructions. If an IJAM is performed while the core is not halted for debug, an internal bus error is generated. The IJAM may be performed, and the results are undefined.

If an access error occurs while jamming instructions, EDBSR1[IJAE] is set.

9.9.5.7 Special Note on Jamming Store Instructions

Under some conditions (for example, when the data cache is disabled), the effects of jamming a store instruction may not immediately become visible in the architectural state of the machine. For example, one might jam a store instruction then examine memory, expecting to find the stored data. However, the data may remain in non-architecture-defined registers within the core, and not yet be visible in memory. In these cases, jamming a **sync 0** instruction forces the data from the non-architecture-defined registers into some architecturally visible memory space.

Also note that jamming a **sync 0** instruction is required immediately prior to changing whether loads/stores are performed to memory storage space or to debug space. Since stores may take some time after completion to be performed, the **sync 0** ensures that the stores are initiated to the appropriate storage space prior to the **sync 0** instruction completing.

9.9.5.8 Instruction Jamming Output

Results from store instructions that target debug space that have been jammed into the processor pipeline are retrieved from the IJDATA0/1 registers. Store word instructions store their data into IJDATA1 Store double instructions store the upper word (bits 0–31) into IJDATA0 and the lower word (bits 32–63) into IJDATA1. The debugger can then perform register accesses to retrieve the data—it must access both registers in the 64-bit data case and only needs to access one of the data registers in the 32-bit case. It is expected that the development tool knows how much result data to expect from an instruction.

9.9.5.9 IJAM Procedure

A summary of the steps to perform various instruction jamming operations appears below:

The following procedure is used for instructions with associated data (input):

1. Confirm that the processor is halted. If not halted, issue a HALT command and wait until the processor is halted.
2. Write IJDATA0 with most significant word (if 64-bit data).
3. Write IJDATA1 with (least significant) word, halfword, or byte.
4. Write IJCFG[MODE] =1 to configure load/store operation.
5. Write IJIR to load instruction and run.
6. Check for IJAM completion status (one of two options):
   - Scan the SoC-level JTAG IR and capture the status that is shifted out in the process. If the status is IJAM not done, repeat this step.
   - Read EDBSR1[IJBUSY] to determine status.
7. On error, check EDBSR1 and EDESR.

NOTE

For 8-bit (byte) and 16-bit (halfword) writes, data should always be written to IJDATA1 right-justified (least significant) independent of the specific address accessed.

The following procedure is used for instructions with associated data (output):

1. Confirm that the processor is halted. If not halted, issue a HALT command and wait until the processor is halted.
2. Write [MODE] =1 to configure load/store operation.
3. Write IJIR to load instruction and run.
4. Check for IJAM completion status (one of two options):
   — Scan the SoC-level JTAG IR and capture the status that is shifted out in the process. If the status is IJAM not done, repeat this step.
   — Read EDBSR1[IJBUSY] to determine status.
5. On error, check EDBSR1 and EDESR.
6. If no error, read IJDATA0—most significant word (if 64-bit data).
7. If no error, read IJDATA1—least-significant word, halfword, or byte.

NOTE

For 8-bit (byte) and 16-bit (halfword) reads, data is always read from IJDATA1 right-justified (least significant) independent of the specific address accessed.

For instructions with no associated data, use the following procedure:

1. Confirm that the processor is halted. If not halted, issue a HALT command and wait until the processor is halted.
2. Write IJIR to load instruction and run
3. Check for IJAM completion status (one of two options):
   — Scan the SoC-level JTAG IR and capture the status that is shifted out in the process. If the status is IJAM not done, repeat this step.
   — Read EDBSR1[IJBUSY] to determine status
4. On error, check EDBSR1 and EDESR

9.9.5.10 Instruction Jamming Error Conditions

If a jammed instruction produces an exception, the instruction does not complete and no interrupt is taken. The exception status information is recorded in debug accessible registers for analysis. Exceptions on a jammed instruction produces the following side effects:

- EDBSR1[LCMP] = 0
- EDBSR1[IJEE] = 1
- EDBSR1[IVOR] = IVOR register number corresponding to the type of exception that occurred.

- EDESR = effective value of the ESR if the exception had been processed
- EDBSR1[IJAE] = 1

9.10 Nexus Trace

This specification defines the auxiliary port functions, transfer protocols and standard development features of the core Nexus module in compliance with IEEE-ISTO 5001. The development features supported are Program Trace, Data Trace, Data Acquisition Messaging, Watchpoint Messaging, and Ownership Trace. The e500mc Nexus module supports two Class 4 features: watchpoint triggering and processor overrun control.

A portion of the pin interface is also compliant with the IEEE 1149.1 JTAG standard. The IEEE-ISTO 5001 standard defines an extensible auxiliary port, which is used in conjunction with the JTAG port.

9.10.1 Nexus Features

The e500mc Nexus module is compliant with the IEEE-ISTO 5001 standard. The following features are implemented:

1. Program Trace via Branch Trace Messaging (BTM). Branch trace messaging displays program flow discontinuities (direct and indirect branches, exceptions, etc.), allowing the development tool to interpolate what transpires between the discontinuities. Thus static code may be traced.
2. Data Trace via Data Write Messaging (DWM). This provides the capability for the development tool to trace writes to (selected) internal memory-mapped resources.
3. Ownership Trace via Ownership Trace Messaging (OTM). OTM facilitates ownership trace by providing visibility of which process ID or operating system task is activated. An Ownership Trace Message is transmitted when a new process/task is activated, allowing the development tool to trace ownership flow.
4. Watchpoint Messaging for the following conditions
   — IAC and DAC events
   — taken interrupts
   — completion of return from interrupt class instructions
   — externally supplied events
   — Performance Monitor events
5. Data Acquisition Messaging (DQM) allows code to be instrumented to export customized information to the Nexus Auxiliary Output Port.
6. Watchpoint Trigger enable of Program Trace Messaging
7. Filtering of Program Trace Messaging based on process (indicated by MSR[PMM]).
8. Auxiliary interface for higher data input/output. This interface may be coupled to a high speed serial port on the device in order to push the information to a development tool.
   — Thirty MDO (Message Data Out) signals.
   — Two MSEO (Message Start/End Out) signals.
   — Five EVTO (Watchpoint Event) signals

— Two EVTI (Event In) signals

9. Registers for Program Trace, Data Trace, Ownership Trace, Data Acquisition, Watchpoint Messaging, and Watchpoint Trigger
10. All features controllable and configurable via a memory mapped interface which is accessible by development tools.
11. All features controllable and configurable via SPRs which are accessible by embedded software.
12. Timestamp capability on all message types.

9.10.2 Enabling Nexus Operations on the Core

By default, clocks for Nexus-related circuitry are inactive. These clocks must be enabled in order to use any of the Nexus features related to the processor. Nexus clocks are activated upon the first access to a Nexus register DC1/2/3/4, WT1, WMSK, OVCR.

Once the core Nexus clocks are active, the various features of the Nexus module can be enabled by programming the Nexus registers via the memory mapped or SPR registers.

If the Nexus module is disabled, no trace output is provided, and the Nexus registers are not accessible.

9.10.3 Modes of Operation

Nexus modes are described as follows:

- Reset

  The core Nexus block is placed in reset whenever the core reset input is asserted. While in reset, the following actions occur:

  — The auxiliary output port signals are deasserted.

  — Registers default back to their reset values and are not accessible until reset negates.
- Disabled

  For a graceful shutdown of Nexus functionality, all trace modes should be disabled first by clearing DC1. The message queues should also be allowed to drain prior to disabling the clocks. Alternatively, a reset can be applied to the core which also resets the Nexus state and disables clocks to the debug circuitry. Failure to shutdown the Nexus block gracefully may produce unpredictable results if the Nexus block is later enabled.

  While disabled, none of the Nexus features are accessible.
- Enabled

  When Nexus is enabled, the various Nexus features may be activated by programming the Nexus control registers, which are accessible via memory mapped or SPR access.

9.10.4 Supported TCODEs

The Nexus auxiliary port allows for flexible transfer operations via public messages. A TCODE defines the transfer format, the number and/or size of the packets to be transferred, and the purpose of each packet. The IEEE-ISTO 5001 standard defines a set of public messages and allocates additional TCODEs for

vendor-specific features outside the scope of the public messages. The Nexus block currently supports the public and vendor defined TCODEs shown in Table 9-29.

Table 9-29. Supported TCODEs

Message Name	Field Size (bits)[1]		Field Name	Field Type	Field Description
	Minimum	Maximum			
Debug Status	6	6	TCODE	fixed	TCODE number = 0
	6	6	SRC	fixed	source processor identifier
	16	16	STATUS	fixed	Debug Status information (from PRSR[32:47])
	0	28	TSTAMP	variable	Timestamp (optional)
Ownership Trace Message	6	6	TCODE	fixed	TCODE number = 2
	6	6	SRC	fixed	source processor identifier
	1	44	PROCESS	variable	Task/Process ID (Refer to Table 9-42 for more information about this field)
	0	28	TSTAMP	variable	Timestamp (optional)
Data Acquisition Message	6	6	TCODE	fixed	TCODE number = 7
	6	6	SRC	fixed	source processor identifier
	8	8	IDTAG	fixed	identification tag (DEVENT[32:39])
	1	32	DQDATA	variable	exported data taken from DDAM[32:63]
	0	28	TSTAMP	variable	Timestamp (optional)
Error Message	6	6	TCODE	fixed	TCODE number = 8
	6	6	SRC	fixed	source processor identifier
	4	4	ETYPE	fixed	error type (Refer to Table 9-32)
	8[2]	8[2]	ECODE	fixed	error code (Refer to Table 9-31)
	0	28	TSTAMP	variable	Timestamp (optional)
Program Trace - Synchronization Message	6	6	TCODE	fixed	TCODE number = 9
	6	6	SRC	fixed	source processor identifier
	1	1	MAP	fixed	instruction address space identifier (IS)
	1	1	I-CNT	variable	for e500mc implementations, this field is set to “0”
	1	32	FADDR	variable	full current instruction address[3]
	0	28	TSTAMP	variable	Timestamp (optional)

Table 9-29. Supported TCODEs (continued)

Message Name	Field Size (bits)[1]		Field Name	Field Type	Field Description
	Minimum	Maximum			
Data Trace - Data Write Message w/ Sync	6	6	TCODE	fixed	TCODE number = 13
	6	6	SRC	fixed	source processor identifier
	4	4	DSZ	fixed	data size (Refer to Table 9-30)
	1	13	F-ADDR	variable	full data write address (leading zeros truncated)
	1	64	DATA	variable	write data value
	0	28	TSTAMP	variable	Timestamp (optional)
Watchpoint Message	6	6	TCODE	fixed	TCODE number = 15
	6	6	SRC	fixed	source processor identifier
	1	16	WPHIT	variable	watchpoint source indicators
	0	28	TSTAMP	variable	Timestamp (optional)
Resource Full Message	6	6	TCODE	fixed	TCODE number = 27
	6	6	SRC	fixed	source processor identifier
	4	4	RCODE	fixed	resource code identifying the full resource (Refer to Table 9-33)
	1	30	RDATA	variable	resource data (Refer to Table 9-33)
	0	28	TSTAMP	variable	Timestamp (optional)
Program Trace - Indirect Branch History Message	6	6	TCODE	fixed	TCODE number = 28
	6	6	SRC	fixed	source processor identifier
	2	2	BTYPE	fixed	branch type (Refer to Table 9-34)
	1	8	I-CNT	variable	# of sequential instructions completed since the last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	U-ADDR	variable	unique portion of the indirect change of flow target address
	1	30	HIST	variable	direct branch / predicate instruction history information
	0	28	TSTAMP	variable	Timestamp (optional)

Table 9-29. Supported TCODEs (continued)

Message Name	Field Size (bits)[1]		Field Name	Field Type	Field Description
	Minimum	Maximum			
Program Trace - Indirect Branch History Message w/ Sync	6	6	TCODE	fixed	TCODE number = 29
	6	6	SRC	fixed	source processor identifier
	2	2	BTYPE	fixed	branch type (Refer to Table 9-34)
	1	8	I-CNT	variable	# of sequential instructions completed since the last predicate instruction, transmitted instruction count, or taken change of flow.
	1	32	F-ADDR	variable	full indirect change of flow target address[4]
	1	30	HIST	variable	direct branch / predicate instruction history information
	0	28	TSTAMP	variable	Timestamp (optional)
Program Trace - Program Correlation Message	6	6	TCODE	fixed	TCODE number = 33
	6	6	SRC	fixed	source processor identifier
	4	4	EVCODE	fixed	event code identifying the event to correlate with program flow (Refer to Table 9-35)
	1	8	I-CNT	variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	30	CDATA	variable	correlation data (branch history information)
	0	28	TSTAMP	variable	Timestamp (optional)

1 The number shown in this column indicates the minimum logical number of bits required in the field after any applicable compression has been employed. The actual minimum number of bits transferred by the implementation may be larger due to constraints of the auxiliary output port width (Nexus packets must be zero-padded out to a port boundary in accordance with IEEE-ISTO 5001).

2 Note: e500mc uses only 8 bit ECODE encodings, whereas other Nexus clients on the integrated device may use 12 bit ECODE encodings. Software decoding Nexus messages should account for this difference.

3 There will be microarchitected (implementation specific) amount of “skid” in terms of the specific instruction address that is transmitted relative to the sync condition. Subsequent program trace message fields (I-CNT / HIST) will be based from this messaged PC value maintaining a coherent trace flow.

4 Program Trace -Indirect Branch History Message w/ Sync is the message that is generated periodically with the F-ADDR

Table 9-30. Data Trace Size (DSZ) Encodings (TCODE = 13)

DSZ Encoding	Transfer Size	Description
0000	0 bytes[1]	0-bit
0001	1 bytes	8-bit
0010	2 bytes	16-bit/halfword
0011	3 bytes	24-bit/string
0100	4 bytes	32-bit/word

Table 9-30. Data Trace Size (DSZ) Encodings (TCODE = 13) (continued)

DSZ Encoding	Transfer Size	Description
0101	5 bytes	Misaligned access
0110	6 bytes	
0111	7 bytes	
1000	8 bytes	64-bit / double
1001	16 bytes	128-bit
1010	32 bytes[1]	256-bit
1011	64 bytes[1]	512-bit
1100-1111	Reserved	

1 Implied data instructions and cache management instructions utilize these encodings. Refer to Section 9.10.13.3, "Data Trace Size Field (DSZ).

Table 9-31. Error Code (ECODE) Encodings (TCODE = 8)

Error Code[1]	Description
xxxxxxx1	Watchpoint Trace Message(s) lost. Applies only to Error Type 0 (ETYPE = 0000)
xxxxxx1x	Data Trace Message(s) lost. Applies only to Error Type 0 (ETYPE = 0000)
xxxxx1xx	Program Trace Message(s) lost
xxxx1xxx	Ownership Trace Message(s) lost. Applies only to Error Type 0 (ETYPE = 0000)
xxx1xxxx	Status message(s) lost (Debug Status). Applies only to Error Type 0 (ETYPE = 0000)
xx1xxxxx	Data Acquisition Message(s) lost
x1xxxxxx	Reserved
1xxxxxxx	Reserved

1 Note: e500mc uses only 8 bit ECODE encodings, whereas other Nexus clients on the integrated device may use 12 bit ECODE encodings. Software decoding Nexus messages should account for this difference.

Table 9-32. Error Type (ETYPE) Encodings (TCODE = 8)

Error Type	Description
0000	Message queue overrun caused one or more messages to be lost
0001	Contention with higher priority messages caused one or more messages to be lost
0010–1111	Reserved

Table 9-33. Resource Code (RCODE) Encoding (TCODE = 27)

Resource Code	Description	RDATA
0000	Instruction counter	Maximum instruction count (0xFD or 0xFE)[1]
0001	Branch history buffer	Branch/predicate history buffer contents

Table 9-33. Resource Code (RCODE) Encoding (TCODE = 27) (continued)

Resource Code	Description	RDATA
0010–0111	Reserved	N/A
1000	Timestamp counter	Maximum timestamp count (0xFF_FFFF)
1001–1111	Reserved	N/A

[1] The e500mc can complete up to two (2) instructions per cycle. The RDATA is transmitted with a value of 0xFD or 0xFE to accurately indicate the maximum instruction count when the RFM is transmitted.

Table 9-34. Branch Type (B-TYPE) Encoding (TCODE = 28, 29)

Branch Type Code	Description
00	Branch instruction
01	Interrupt
1*x*	Reserved

Table 9-35. Event Code (EVCODE) Encoding (TCODE = 33)

Event Code	Mnemonic	Description
0000	EVCODE #1	Entry into halted state for debug
0001	EVCODE #2	Entry into halted or stopped state for power management
0010–0011	—	Reserved
0100	EVCODE #5	Program trace disabled
0101–1000	—	Reserved
1001	EVCODE #10	Begin masking of program trace due to MSR[PMM] = 0. This event applies only if DC4[PTMARK] = 1.
1010	EVCODE #11	Branch and link occurrence (direct branch function call)
1011–1111	—	Reserved

9.10.5 Nexus Message Fields

Nexus messages are comprised of fields. Each field is a distinct piece of information within a message, and a message may contain multiple fields. Messages are transferred in packets over the Auxiliary Output protocol. A packet is a collection of fields. A packet may contain any number of fixed length fields, but may contain at most one variable length field. The variable length field must be the last field in a packet. This section provides information on some of the fields that comprise the supported messages.

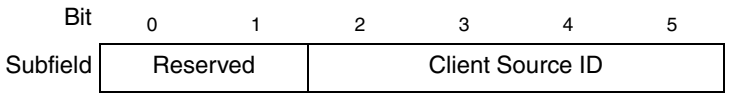
9.10.5.1 TCODE Field

The TCODE field is a 6-bit fixed length field that identifies the type of message and its format. The field encodings are assigned by IEEE-ISTO 5001.

9.10.5.2 Source ID Field (SRC)

Each Nexus module in a device is identified by a unique client source identification number. The number assigned to each Nexus module is determined by the Integrated device. The core implements a 6-bit fixed length source ID field consisting of 4-bit client source ID with the upper 2 bits reserved.

Figure 9-19. Source ID Field Structure



9.10.5.3 Relative Address Field (U-ADDR)

The non-sync forms of the program trace messages include addresses which are relative to the address that was transmitted in the previous synchronizing program trace message. The relative address format is compliant with IEEE-ISTO 5001 and is designed to reduce the number of bits transmitted for address fields.

The relative address is generated by XORing the new address with the previous, and then using only the results up to the most significant 1. To recreate the original address, the relative address is XORed with the previously decoded address.

The relative address of a program trace message is calculated with respect to the previous program trace message, regardless of any address information that may have been sent in any other trace messages in the interim between the two program trace messages.

9.10.5.4 Full Address Field (F-ADDR)

Program trace synchronization messages provide the full address associated with the trace event (leading zeros may be truncated) with the intent of providing a reference point for development tools to operate from when reconstructing relative addresses. Synchronization messages are generated at significant mode switches and are also generated periodically to ensure that development tools are guaranteed to have a reference address given a sufficiently large sample of trace messages. Program Trace -Indirect Branch History Message w/ Sync is the message that is generated periodically with the F-ADDR

9.10.5.5 Timestamp Field (TSTAMP)

The timestamp field is enabled by programming DC1[TSEN]. There are two supported timestamp modes: fine and coarse. When fine timestamping is enabled, the timestamp field is appended to all messages from the Nexus client and provides a time reference for the trace event. When coarse timestamping is enabled, the timestamp field is appended periodically, once every 32 messages.

The timestamp value is recorded at the time that the message enters the internal message queues. The timestamp value is constructed from a 24-bit counter operating at the processor frequency plus a 4-bit correction counter.

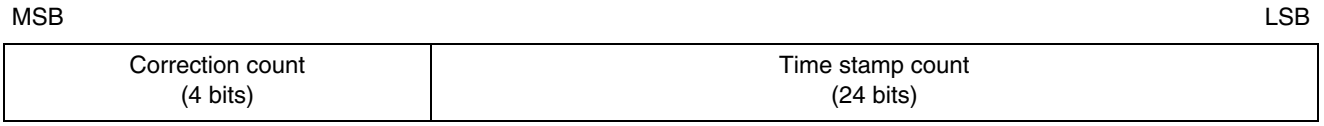


Figure 9-20. Time Stamp Field Components

When a message pends due to contention with other message types, a 4-bit counter is used to keep track of how long the message pends until it actually enters the message queues. This 4-bit correction value is concatenated with the 24-bit timestamp and can be used to correct the timestamp value for that pending latency by subtracting the correction value from the 24-bit timestamp value. If a message pends for 15 or more cycles, the timestamp correction indicates a value of 0xF. A timestamp correction value of 0xF should be taken to mean that the timestamp value for that message is unreliable.

Whenever the 24-bit timestamp counter overflows, a Resource Full Message (RFM) is generated with a resource code of 0x8 and an RDATA field of 0xFF_FFFF. The Timestamp Resource Full Messages caused by do not pend. Clearing DC1[TSEN] will disable the timestamp counter, preventing Resource Full Messages from being generated due to timestamp overflow.

9.10.6 Nexus Message Queues

The e500 Nexus block implements internal message queues capable of storing two messages per cycle. Messages that enter the queue are transmitted in the order in which they are received.

If more than two messages attempt to enter the queue in the same cycle, the two highest priority messages are stored and the remaining messages may pend and retry transmission to the queues on subsequent cycles. Refer to Section 9.10.7, “Nexus Message Priority,” for more information on message priorities.

The overrun control register (OVCR) controls the Nexus behavior as the message queue fills. The Nexus block may be programmed to do the following:

- Allow the queue to overflow, drain the contents, queue an overrun error message and resume tracing.
- Stall the processor instruction completion when the queue utilization reaches the selected threshold.
- Suppress selected message types when the queue utilization reaches the selected threshold.

The Nexus message queues may fill due to software application behavior, which in general are not detailed here.

9.10.6.1 Message Queue Overrun

In this mode, the message queue stops accepting messages when an overrun condition is detected. The contents of the queues are allowed to drain until empty. Incoming messages are discarded until the queue is emptied. At this point, an overrun error message is enqueued which contains information about the types of messages that were discarded due to the overrun condition.

9.10.6.2 CPU Stall

In CPU stall mode, instruction completion is stalled whenever the queue utilization reaches the selected threshold. PRSR[STALL_ST] is set whenever the trigger condition is reached and remains set until the stall condition is negated. The instruction completion is stalled long enough drop one threshold level below the level which triggered the stall. For example, if stalling the processor is triggered at 1/4 full, the stall stays in effect until the queue utilization drops to empty. There may be significant skid from the time that the stall request is made until the processor is able to stop completing instructions. This skid should be taken into consideration when programming the threshold. Refer to Section 9.4.7, "Nexus Overrun Control Register (OVCR)," for programming options.

9.10.6.3 Message Suppression

In this mode, the message queue disables selected message types whenever the queue utilization reaches the selected threshold. This allows lower bandwidth tracing to continue and possibly avoids an overrun condition. If an overrun condition occurs despite this message suppression, the queue responds according to the behavior described in Section 9.10.6.1, "Message Queue Overrun." As soon as it is triggered, message suppression remains in effect until queue utilization drops the threshold below the level selected to trigger the suppression.esse

9.10.7 Nexus Message Priority

Nexus messages may be lost due to contention with other message types under the following circumstances:

- A new message is generated for a message type already pending retry due to contention with other message types in the previous cycle. The pending message is kept and continues to arbitrate for entry into the message queues. The new message is discarded. See Table 9-36 for a listing of various message types and their relative priority.
- More than two messages within the program trace message type are generated in the same cycle.

Table 9-36 lists the various message types and their relative priority from highest to lowest. Note that program trace has been allocated two ports into the message buffer so that two messages can be generated in one cycle.

Up to two message requests can be queued into the message buffer in a given cycle. If more than two message requests exist in a given cycle, the two highest priority message classes are queued into the message buffer. Any remaining messages that did not successfully queue into the message buffer in that cycle generate subsequent responses, as described in Table 9-36.

Table 9-36. Message Type Priority and Message Dropped Responses

Priority	Message Type	Message	Pend and Retry on Arbitration Loss?	Message Dropped Response
0 (highest)	Error	Error	N/A	N/A
1	Watchpoint Trace	Watchpoint Message (WPM)	Y	None

Table 9-36. Message Type Priority and Message Dropped Responses (continued)

Priority	Message Type	Message	Pend and Retry on Arbitration Loss?	Message Dropped Response
2	Data Acquisition	Data Acquisition Message (DQM)	Y	DQM error message
	Ownership Trace	Ownership Trace Message (OTM)	Y	None
3	Program Trace (port 1)	Indirect Branch with History (IHM)	Y	BTM error message sync upgrade next IHM
		Resource Full Message (RFM) for instruction counter, history buffer and timestamp overflow	Y	
4	Program Trace (port 2)	Program Correlation Message (PCM)	Y	BTM error message sync upgrade next IHM
		Debug Status Message (DS)	Y	Sync upgrade next IHM
5 (lowest)	Data Trace	Data Trace Write Message (DTM)	Y	None

9.10.7.1 Data Acquisition Message Priority Loss Response and Retry

If a Data Acquisition Message (DQM) loses arbitration due to contention with higher priority messages, the DQM pends and retries on the subsequent cycle. If a new data acquisition event occurs while a DQM is pending, the new event is discarded. An error message is generated to indicate that a DQM has been lost due to contention.

9.10.7.2 Ownership Trace Message Priority Loss Response and Retry

If an Ownership Trace Message (OTM) loses arbitration due to contention with higher priority messages, the OTM pends and retries on the subsequent cycle. If a new ownership trace event occurs while an OTM is pending, then the new event generates a replacement message. Even if the pending OTM is a periodic update, software updates of the process ID information are more important than periodic refreshes of the process ID state and the new message is transmitted.

9.10.7.3 Program Trace Message Priority Loss Response and Retry

If a Program Trace Message (PTM) loses arbitration due to contention with higher priority messages, the PTM pends and retries on the subsequent cycle. If a new program trace event occurs while a PTM is pending, the new event is discarded. If the discarded PTM is a Program Correlation Message, a Resource Full message for instruction count, history buffer, timestamp overflow or an Indirect Branch with History message, then an Error message is generated to indicate that branch trace information has been lost.

Once the pending PTM is enqueued, if another PTM was discarded during the retry phase, then the next Indirect Branch with History Message is upgraded to a sync-type message.

9.10.8 Data Trace Message Priority Loss Response and Retry

If a Data Trace Message (DTM) loses arbitration due to contention with higher priority messages, the DTM pends and retries on the subsequent cycle. If a new data trace event occurs while a DTM is pending, the new event is discarded.

9.10.9 Debug Status Messages

Debug Status Messages are enabled whenever any Nexus trace modes are enabled (DC1[TM] is nonzero). A debug status message is generated whenever the processor state changes. Any transition between normal, halted and stopped states constitutes a processor state change for the purpose of generating debug status messages.

9.10.10 Error Messages

Error messages are enabled whenever the debug logic is enabled. There are two conditions that produce an error message, each receiving a separate error type designation:

- A message is discarded due to contention with other (higher priority) message types. Error messages have the highest priority - error messages generated if any other messages are discarded due to contention. Such errors have an Error Type value of 4’b0001.
- The message queue overruns. After the queue is drained, an error message is enqueued with an error code that indicates the types of messages discarded during that time. Such errors have an Error Type value of 4’b0000.

9.10.11 Resource Full Messages

Certain trace resources, such as counters and history buffers, have hardware limitations to their size. To avoid losing information when these resources become full, the e500mc is capable of generating Resource Full messages. The information from this message is added or concatenated with information from subsequent messages to interpret the full picture of what has transpired. For the e500mc, Resource Full messages are generated upon overflow of any one of three resources: instruction counter, history buffer and timestamp counter.

The instruction counter is capable of counting up to 255 sequential instructions before overflowing. If the instruction counter overflows, a Resource Full message is generated. Development tools can use this information to properly reconstruct program flow. Disabling program trace will disable the instruction counter, preventing Resource Full messages from being generated due to this resource.

The branch/predicate history buffer is capable of storing up to 30 bits (29 history events plus the stop bit). The history buffer is reset whenever the branch/predicate history information is transmitted in a message. If the history buffer becomes full, a Resource Full message is generated to transmit the contents of the history buffer. Development tools can concatenate this history information with history fields from other program trace messages to obtain the complete branch/predicate history. Disabling program trace will disable logging branch/predicate information in the history buffer, preventing Resource Full messages from being generated due to this resource.

The timestamp counter is a 24-bit resource which counts cycles at the e500mc processor frequency. When enabled, the value from this counter (along with a 4-bit correction value) is appended to trace messages as they enter the internal message queues. If the timestamp counter overflows, a Resource Full message is generated to transmit the maximum timestamp value (0xFF_FFFF). Development tools can append this value to the timestamp value transmitted within the next trace message to reconstruct the true timestamp value. Disabling the timestamp feature by clearing DC1[TSEN] will disable the timestamp counter, preventing Resource Full messages from being generated due to this resource.

The specific resource that has become full is indicated by the resource code (RCODE) within the Resource Full message. The data associated with the specific resource is captured in the resource data field (RDATA). These fields and their values are outlined in Table 9-33, "Resource Code (RCODE) Encoding (TCODE = 27)."

9.10.12 Program Trace

This section details the program trace mechanisms supported by the Nexus module included in the core. Program trace is implemented using Branch Trace Messaging (BTM) in accordance with IEEE-ISTO 5001 definitions.

Branch Trace Messaging facilitates program trace by providing the following types of information:

- The number of sequential instructions which have completed since the last predicate instruction, transmitted instruction count, or taken change of flow.
- Branch/predicate history indicating whether direct branches in the program flow were taken or not as well as indicating whether or not predicate instructions were executed.
- In the case of indirect changes of flow (including interrupts), the target address of the change of flow is provided.

9.10.12.1 Enabling and Disabling Program Trace

Program trace can be enabled in one of two ways:

- Setting the appropriate DCI[TM] bit (DC1[61]).
- Programming the PTS field of the WT1 register to enable program trace on the occurrence of a watchpoint condition.

Similarly, program trace may be disabled by the following:

- Clearing the appropriate DCI[TM] bit (DC1[61]). Note that resetting the Nexus module clears all Nexus registers, disabling program trace as a side effect.
- Programming WT1[PTE] to disable program trace on the occurrence of a watchpoint condition.
- Program trace can be filtered out (effectively disabled) when certain processes are active by programming DC4[PTMARK].

Program trace is effectively suppressed whenever the processor core is in the debug halted or debug stopped state. Instruction jamming operations do not produce any Program Trace Messages. Whenever the processor core leaves the debug halted state, program trace enable state reverts to the status of DC1[61].

9.10.12.2 Sequential Instruction Count Field

Most of the program trace messages include an instruction count field. This instruction count indicates the number of sequential instructions that have completed since the last predicate instruction, transmitted instruction count, or taken change of flow. Taken indirect branch instructions are included in this count. Instructions which produce branch/predicate history information are not included in this count. The instruction counter is reset every time the instruction count is transmitted in a message or whenever there is a branch/predicate history event.

9.10.12.3 Branch/Predicate History Events

The branch/predicate history buffer stores information about branch and predicate instruction execution. The buffer is implemented as a left-shifting register. The buffer is preloaded with a one (1) which acts as a stop bit (the most significant 1 in the history field is a termination bit for the field).

A value of one (1) is shifted into the history buffer for each taken direct branch (program counter relative branch) or predicate instruction whose condition evaluates to true. A value of zero (0) is shifted into the history buffer for each not-taken branch (including indirect branch instructions) or predicate instruction whose condition evaluates to false. The e500mc implements a 30-bit history buffer (29 history bits plus 1 stop bit).

This history buffer information is transmitted as part of an Indirect Branch with History Message, as part of a Program Correlation Message, or as part of a Resource Full Message if the history buffer becomes full.

Table 9-37. Branch/Predicate History Events

Branch/Predicate History Event	History Bit	Relevant Instructions	Notes
Not taken register indirect branches	0	**bcctr**, **bcctrl**, **bclr**, **bclrl**	—
Not taken direct branches	0	**b**, **ba**, **bc**, **bca**, **bla**, **bcla**, **bl**, **bcl**	—
Taken direct branches	1	**b**, **ba**, **bc**, **bca**, **bla**, **bcla**, **bl**, **bcl**	If EVCODE for direct branch function calls is not masked in DC4, taken **bl** and **bcl** instructions generate Program Correlation Messages and are not logged in the history buffer.
Predicated instructions	1	**isel**, **fsel**	—

9.10.12.4 Indirect Branch Message Events

An indirect branch event is a change of flow whose branch target cannot be inferred from the source code. This includes register indirect branch instructions, and interrupts. When an indirect branch event occurs and program trace is enabled, an Indirect Branch with History Message (TCODE 28) is generated.

The address field of this message is the target address of the change of flow. For interrupts, this is the interrupt vector.

Table 9-38. Indirect Branch Message Events

Indirect Branch Message Event	BTYPE	Relevant Instructions
Taken register indirect branches	00	**bcctr, bcctrl, bclr, bclrl**
Return from Interrupt	00	**rfi, rfci, rfdi, rfmci**
Interrupt taken	01	N/A Interrupts caused by **sc**, **tw**, and **twi** are messaged in the same way as any other taken interrupt event.
Reserved	11	N/A

9.10.12.5 Resource Full Events

Program trace can produce three types of Resource Full messages (TCODE 27): instruction counter, history buffer and timestamp.

The instruction counter is capable of counting up to 255 sequential instructions before overflowing. If the instruction counter overflows, a Resource Full message is generated. Development tools can use this information to properly reconstruct program flow.

The branch/predicate history buffer is capable of storing up to 30 bits (29 history events plus the stop bit). The history buffer is reset whenever the branch/predicate history information is transmitted in a message. If the history buffer becomes full, a Resource Full message is generated to transmit the contents of the history buffer. Development tools can concatenate this history information with history fields from other program trace messages to obtain the complete branch/predicate history.

9.10.12.6 Program Correlation Events

Program Correlation Messages (TCODE 33) are used to correlate processor events to instructions in the program flow. Program Correlation Messages provide branch/predicate history and sequential instruction count information at the time the event is detected. This information can be used by development tools to correlate the event to an instruction in the program flow. Each event can be independently masked by setting a bit in Nexus Development Control Register 4 (DC4).

The e500 Nexus module generates Program Correlation Messages for the following events when program trace is enabled and the event is not masked in DC4:

- The processor is halted for debug.
- The processor is halted for power management.
- Program trace becomes disabled (excluding disable by reset)
- Program trace becomes masked due to MSR[PMM] = 0 and DC4[PTMARK] = 1
- Branch and Link instructions (direct branch function call, **bl/bcl**)

9.10.12.7 Synchronization Conditions

By default, program trace messages perform XOR compression on the branch target address to produce the address field for the message. This compression is consistent with the specification in IEEE-ISTO 5001.

Under some conditions an uncompressed address is sent to provide development tools with a baseline reference address. The nature of these conditions determines the type of message transmitted. In cases where there is a discontinuity in program flow, a synchronization message is transmitted indicating a “hard” sync has occurred (TCODE 9). Subsequent Program Trace Messages will base their sequential

instruction count (I-CNT) and branch history (HIST) values starting from the program counter (PC) value transmitted within this message. Hard sync cases are outlined in Table 9-39.

Table 9-39. Hard Synchronization Conditions

Hard SYNC Condition	Description
$\overline{\text{EVTI}}$0 Assertion	The e500mc $\overline{\text{EVTI}}$0 pin has been asserted (high to low transition) and DC1[EIC] determines that $\overline{\text{EVTI}}$0 generates trace synchronization messages.
Exit from System Reset	The embedded processor has successfully exited system reset. For program trace messages, this is required to allow the *number of instruction units executed* packet in a subsequent BTM to be correctly interpreted by the tool.
Exit from Debug	The embedded processor has exited from the debug HALT state.
Program Trace Enable	Program Trace is enabled during normal execution of the embedded processor.
FIFO Overrun	An overrun condition had previously occurred in which one or more trace occurrences were discarded by the debug logic. To inform the tool that an overrun condition occurred, the target outputs an Error Message (TCODE = 8) prior to a *sync* message. The error message contains an ECODE value indicating the type(s) of messages lost due to the overrun condition.
Message Contention	One or more messages was lost due to contention with a higher priority message. To inform the tool that this condition occurred, the target outputs an Error Message (TCODE = 8) prior to a *sync* message. The error message contains an ECODE value indicating the type of message lost due to the contention.See Section 9.10.7, “Nexus Message Priority”
Exit from Power-down	The processor has exited from a power management state. For program trace messages, this is required to allow the *number of instruction units executed* packet in a subsequent BTM to be correctly interpreted by the tool.

Conditions which do not create a discontinuity are considered “soft” sync cases. These conditions cause the next branch trace message to use an uncompressed target address (TCODE 29). Soft sync cases are outlined in Table 9-40.

Table 9-40. Soft Synchronization Conditions

Soft SYNC Condition	Description
$\overline{\text{EVTI}}$1 Assertion	The e500mc $\overline{\text{EVTI}}$1 pin has been asserted (high to low transition).
Periodic Message Counter	The periodic trace message counter has expired indicating that there have been 255 program trace messages without an uncompressed address.This will upgrade the next program trace message to be a sync type. This insures that with a sufficiently large sample of trace information, there is guaranteed to be a reference address that can be used to meaningfully interpret the remainder of the program trace.

9.10.13 Data Trace

The e500mc supports limited data trace. The features of data trace are as follows:

- Only stores are traced.
- The data trace message is uncorrelated (meaning there is no corresponding Program Correlation Message (PCM)).

- Each address compare is limited to a maximum of 4 Kbytes on exact match (see Section 2.17.5, “Debug Control Register 4 (DBCR4),” for detail on programming extended DAC ranges).
- Misaligned stores are not combined, meaning that each half that has an associated DAC set is sent as an independent data trace message.
- Store multiple word instructions (**stmw**) produce a separate data trace message for each word stored that meets the trace criteria.

9.10.13.1 Enabling and Disabling Data Trace

The data trace features rely on the data address compare (DAC) resources in order to compress address information by implying upper order address bits from the DAC attribute. Consequently, data trace functionality requires DAC settings to be enabled in addition to enabling messaging.

To enable DACs for use by data trace, the following conditions are required:

- DBCR0[IDM] = DBCR0[EDM] = 0. This is required because comparator resources are shared with the architecture-defined DAC function. If DBCR0[IDM] or DBCR0[EDM] is set, the DAC function has precedence and Data Trace Messages are not generated.

NOTE

DAC conditions may also trigger Data Trace Messages if the corresponding store operation completes. In most cases, a DAC condition prevents the store operation from completing (by causing entry into debug halted state or generating a debug interrupt).

The case which does not prevent completion is when DBCR0[EDM] = 0, DBCR0[IDM] = 1, and MSR[DE] = 0. In this case, DAC conditions produce imprecise debug events which do not suppress the completion of the corresponding store operation and consequently produce Data Trace Messages when enabled.

- DAC1 and DAC2 should be programmed with the desired addresses for data tracing.
- DBCR0[DAC1, DAC2] should be programmed to enable the DAC condition to occur on store-type data storage accesses.
- DBCR4[DAC1XM] and DBCR4[DAC2XM] should be programmed to construct data trace address regions which do not exceed 4 Kbytes. If a DAC match region exceeds 4 Kbytes, the resulting data trace may be ambiguous as a result of address aliasing.
- Additional filtering of data trace according to privilege and/or address space may be applied by programming DBCR2[DAC1U,DAC1ER,DAC2US,DAC2ER].

Data Trace Messaging can be enabled in one of two ways:

- Setting the appropriate DCI[TM] bit (DC1[62]).
- Programming WT1[DTS] to enable data trace on the occurrence of a watchpoint condition.

Similarly, data trace may be disabled by one of the following:

- Disabling the DAC conditions for store-type accesses.

- Clearing the appropriate DCI[TM] bit (DC1[62]). Note that resetting the Nexus module clears all Nexus registers, disabling data trace as a side effect.
- Programming WT1[DTE] to disable data trace on the occurrence of a watchpoint condition.

NOTE

The latter two mechanisms defined above will disable additional stores from entering the e500mc store queue, but accesses which have already entered the queue (that is, accesses in flight) are messaged out before the DTMs are actually disabled.

Data trace is effectively suppressed whenever the processor core is in the debug halted or debug stopped state. Instruction jamming operations do not produce any Data Trace Messages. Whenever the processor core leaves the debug halted state, data trace enable state reverts to the status of DC1[62].

9.10.13.2 Data Trace Range Control

The data trace address range is limited to two 4-Kbyte ranges. These ranges are controlled by setting the effective addresses in DAC1 and DAC2 and the DAC configuration in DBCR0 and DBCR4. DBCR0[DAC1] and DBCR0[DAC2] must be enabled for store-type data storage accesses. DBCR4[DAC1XM] and DBCR4[DAC2XM] must be set to modes that ensure that the address range of each DAC does not exceed 4 Kbytes.

9.10.13.3 Data Trace Size Field (DSZ)

For normal data transfers, DSZ indicates the size (in bytes) of the store that is being traced, but there are two special cases which use unique DSZ values to indicate specific types of data transfers.

Certain cache management instructions (**dcba, dcbz, dcbal, dcbzl**) are treated as store-type data storage accesses and always have a data value of zero. For the **dcbz** and **dcba** instructions, a mode bit (L1CSR0[DCBZ32]) determines whether or not 32-bytes or 64-bytes are zeroed out. This is indicated with the respective DSZ value within the data trace message. Data Trace Messages for **dcbzl** and **dcbal** instructions will always include a DSZ value indicating 64-bytes (4’b1011). Refer to Section 3.4.10.1, “User-Level Cache Instructions” for more detail on cache management instructions.

The e500mc supports an additional instruction form, called decorated storage notify (**dsn**) which provides the ability to send an address along with a decoration, but does not include any data. This implied zero data trace message transmitted with a DSZ value of zero (4’b0000) indicating a zero byte data transfer. Refer to the integrated device reference manual for more detail on the **dsn** instruction.

9.10.13.4 Data Trace Address Field

The data trace address field consists of a 12-bit address offset and a 1-bit DAC tag identifier as follows:

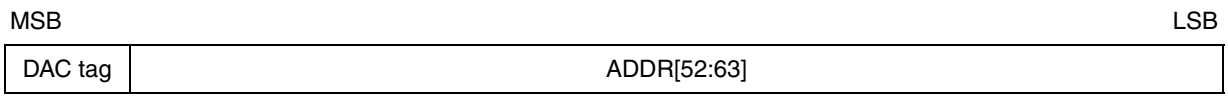


Figure 9-21. Data Trace Address Field Components

A value of 0 for the DAC tag indicates that this store matched only the DAC1 conditions or matched both the DAC1 and DAC2 conditions. A value of 1 for the DAC tag indicates that this store matched only the DAC2 conditions. The full effective address can be reconstructed by concatenating the DAC information with the data trace address field information as follows:

DAC1[32–51] or DAC2[32–51]	ADDR[52–63]

Figure 9-22. Data Trace Full Address Reconstruction

The upper address information should be selected from DAC1 or DAC2 according to the DAC tag bit in the data trace address field and the DAC settings. Note that setting the DAC conditions to include regions in excess of 4 Kbytes results in address aliasing making precise reconstruction of the full effective address impossible (without other implied restrictions or information that can remove the ambiguity).

9.10.13.5 Data Trace Data Field

The data trace data field contains the data that was written by a store operation that met the requirements for being traced. Leading zeros are truncated to an auxiliary output port boundary and not transferred in the message.

9.10.13.6 Data Trace Message Events

A data trace event is a store-type data storage access that is executed by the load/store unit and which meets the criteria set forth for a DAC condition. Additional filtering and triggering may be applied to control when data trace events are observed. When a qualified data trace event occurs and data trace is enabled, a Data Trace Write with Sync message (TCODE 13) is generated.

Table 9-41. Data Trace Message Events

Data Trace Message Event Source	Relevant Instructions	Notes
Cache management instructions that are treated as store-type data storage accesses	**dcba**, **dcbal**, **dcbz**, **dcbzl, dcbzep, dcbzl, dcbzlep**	Treated like data storage writes with write data value of zero (refer to Section 9.10.13.3, “Data Trace Size Field (DSZ)).
Store instructions that produce data storage write accesses	**stb**[**u**][**x**], **stbepx**, **stfd**[**u**][**x**], **stfdepx**, **stfiwx**, **stfs**[**u**][**x**], **sth**[**u**][**x**], **sthbrx**, **sthepx**, **stmw**, **stw**[**u**][**x**], **stwbrx, stwepx**	For **stmw**, a separate data trace message is generated for each word stored that meets the trace criteria.
Conditional store instruction	**stwcx.**	Only generates data trace messages if the associated store is successful (that is, the condition evaluates to true)

9.10.14 Ownership Trace

Ownership trace facilitates tracking the active operating system task by providing visibility to special purpose registers designated for use by the OS for process ID. All OS process ID changes which are reflected in the Nexus Process ID Register (NPIDR) or PID Register will generate Ownership Trace Messages. Changes in the logical partition ID (LPIDR) will also generate OTMs.

9.10.14.1 Enabling and Disabling Ownership Trace

Ownership trace can be enabled by the following:

- Setting the appropriate DCI[TM] bit (DC1[63]).

Similarly, ownership trace may be disabled by one of the following:

- Clearing the appropriate DCI[TM] bit (DC1[63]). Note that resetting the Nexus module clears all Nexus registers, disabling program trace as a side effect.
- Periodic ownership trace message events can be disabled by setting DC1[POTD]. Ownership trace message events due to **mtspr** instruction execution are unaffected by this control.

Ownership trace is effectively suppressed whenever the processor core is in the debug halted or debug stopped state. Instruction jamming operations do not produce any ownership trace messages. Whenever the processor core leaves the debug halted state, ownership trace enable state reverts to the status of DC1[63].

9.10.14.2 Ownership Trace Process Field

The process field of an Ownership Trace Message provides the contents of several pieces of process ID information. The PID value that is transmitted as part of the message is based on the type of ownership trace event as well as the value of DC1[OTS]. Refer Section Table 9-42., “OTM PROCESS Field Components”. The process field also consists of an index to identify which process ID values are being reported for a particular message. Refer to Section 9.10.14.3, “Standard Ownership Trace Message Events” and Section 9.10.14.4, ““Sync” Ownership Trace Message Events” for details on ownership trace events.

Table 9-42. OTM PROCESS Field Components

PID Index (4 Bits)	PID Index Description	PID Value (up to 40 bits)
0000	OS PID information	Process ID (PID)[1]
0001	Hypervisor PID information	Logical Partition ID (LPIDR)
0010	“Sync” PID information	{LPIDR, MSR[GS], PID[1], MSR[PR]}
0011–1111	Reserved, not used	N/A

[1] NPIDR[32:63] or PID0 (as programmed in DC1[OTS])

9.10.14.3 Standard Ownership Trace Message Events

The following two events generate standard Ownership Trace Messages when ownership trace is enabled:

- As programmed by DC1[OTS], a write to either (1) the NPIDR register or (2) the PID register is performed by executing an **mtspr** with the selected register as the target. The PROCESS field of the resulting ownership trace message indicates that the process ID changed with a PID index of 0000 and that the new value written to the selected register is conveyed in the PID value subfield.
- When the hypervisor changes LPIDR, an OTM message indicates that the logical partition ID changed with a PID index of 0001 and the new LPIDR is conveyed in the PID value subfield.

9.10.14.4 “Sync” Ownership Trace Message Events

The following three events generate “sync” OTM messages when ownership trace is enabled:

- Upon assertion of EVTI0 (when DC1[EIC] is programmed to initiate synchronization), all of the most recent process ID information is messaged out with a PID index of 0010. This effectively creates a “sync” OTM and the PROCESS field for this message reflects the current value in the NPIDR register (or PID0), the current privilege level (MSR[PR]), the current logical partition ID (LPID) as well as the current guest OS state (MSR[GS]).
- Upon a change in privilege level (MSR[PR]) or a change in guest state (MSR[GS]), an OTM will be generated with the same information as in the “sync” OTM case (PID index = 0010).
- Periodically, once every 256 messages, an OTM will also be generated with the same information as in the “sync” OTM case (PID index = 0010). These periodic ownership trace message events can be disabled by setting DC1[POTD].
- After draining the Nexus queues due to nexus overrun condition, an OTM “sync” message will be generated. If Ownership changed during the flush of the Nexus queues, this message along with the Hard Sync message synchronizes the trace tool again to the current program flow.

9.10.15 Data Acquisition

This section details the data acquisition mechanisms supported by the Nexus module included in the e500mc core. Data Acquisition Trace is implemented using Data Acquisition Trace Messages in accordance with IEEE-ISTO 5001 definitions. The control mechanism to export the data is different from the recommendations of the standard, however.

Data Acquisition Trace provides a convenient and flexible mechanism for the debugger to observe the architectural state of the machine through software instrumentation in either IDM or EDM mode.

9.10.15.1 Enable and Disable Data Acquisition Trace

Enabling and disabling data acquisition trace messaging is done as follows:

- Enabled by setting the appropriate DCI[TM] bit (DC1[58]).
- Disabled by clearing the appropriate DCI[TM] bit (DC1[58]).

Note that resetting the Nexus module clears all Nexus registers, disabling Data Acquisition Trace as a side effect.

Data Acquisition Trace is effectively suppressed whenever the processor core is in the debug halted or debug stopped state. Instruction jamming operations do not produce any Data Acquisition Trace messages. Whenever the processor core leaves the debug halted state, Data Acquisition Trace enable state reverts to the status of DC1[58].

9.10.15.2 Data Acquisition ID Tag Field

The ID tag field (IDTAG) is a 8-bit value specifying the complementary control or attribute information for the data included in the Data Acquisition Message. IDTAG is configured by accessing the DQM resources through the DEVENT and DDAM SPR registers.

IDTAG is sampled from DEVENT[32:39] when a write to DDAM is performed via **mtspr** operations.

The IDTAG is left to the discretion of the development tool to be used in whatever manner is deemed appropriate for the application.

9.10.15.3 Data Acquisition Data Field

The Data Acquisition data field (DQDATA) is the data captured from the DDAM write operation via mtspr operations. DQDATA is sampled from DDAM[32:63].

9.10.15.4 Data Acquisition Trace Event

For DQM, a dedicated SPR has been allocated (DDAM). It is expected that the general use case is to instrument the software and use mtspr operations to generate Data Acquisition Messages.

There is no explicit error response for failed accesses as a result of contention between an internal and external debugger. Refer to Section 9.8.2, "Internal and External Debug Modes" for more information regarding internal/external debugger contention of debug resources. Reads from the data acquisition message register do not generate a data acquisition event and return zeros for the read data.

9.10.16 Watchpoint Trace

This section details the watchpoint trace mechanisms supported by the Nexus module included in the core. Watchpoint Trace trace is implemented using Watchpoint Trace Messaging in accordance with IEEE-ISTO 5001 definitions.

Watchpoint Trace facilitates monitoring program execution for specific event occurrences.

9.10.16.1 Watchpoint Events

Table 9-43 lists all of the watchpoint events supported by the e500mc core. These watchpoint events may be used for one or more of the following functions:

- Triggers for enabling/disabling program trace according to the settings programmed in the WT1 register.
- Assert the debug event out signals (EVTO[4:0]) according to the settings in DC1[EOC] and DC2.
- Generate a Watchpoint Trace message according to the settings programmed in DC1 and WMSK registers.

Table 9-43. Core Debug Watchpoint Mappings

Core Watchpoints	Event Description
Watchpoint #1	IAC1 event
Watchpoint #2	IAC2 event
Watchpoint #3	Interrupt taken
Watchpoint #4	Return from interrupt class instruction completed
Watchpoint #5	DAC1 event

Table 9-43. Core Debug Watchpoint Mappings (continued)

Core Watchpoints	Event Description
Watchpoint #6	DAC2 event
Watchpoint #7	Event In (0) (EVTI0)[1]
Watchpoint #8	Event In (1) (EVTI1)
Watchpoint #9	Data Acquisition Event (0) (DVT0)
Watchpoint #10	Data Acquisition Event (1) (DVT1)
Watchpoint #11	Process ID Update (PID)
Watchpoint #12	Performance Monitor Watchpoint 0 (PMW0)[2]
Watchpoint #13	Performance Monitor Watchpoint 1 (PMW1)[2]
Watchpoint #14	Performance Monitor Watchpoint 2 (PMW2)[2]
Watchpoint #15	Reserved

[1] Assertion of EVTI0 produces a watchpoint independent of the settings of DC1[EIC]. That is, an EVTI0 assertion produces a watchpoint in addition to any functionality that is enabled in DC1[EIC]
[2] Configuration is controlled by the PMLCb registers. See Section 2.18.3, "Local Control B Registers (PMLCb0–PMLCb3)."

When the processor is in EDM, IACs, DACs return from interrupt. Return from critical interrupt debug conditions cause bits to be set in EDBSR0 and the processor to halt instead of taking a debug interrupt. In these cases, the watchpoint for the respective events will trigger on the update to EDBSR0 rather than the debug interrupt. For e500mc Rev 1.x and Rev 2.x, IACs and DACs that cause halts in EDM mode will not trigger watchpoints.

9.10.16.2 Enabling and Disabling Watchpoint Trace Messaging

Watchpoint trace messaging can be enabled by setting the appropriate DCI[TM] bit (DC1[60]) and enabling selected watchpoint events to produce a watchpoint trace message by programming WMSK. Note that except for interrupt taken, return from interrupt, and EVTI events, additional configuration is required to setup the individual watchpoint conditions. These additional configuration controls are specific to the event type.

Similarly, watchpoint trace may be disabled by the following:

- Clearing the appropriate DCI[TM] bit (DC1[60]). Note that resetting the Nexus module clears all Nexus registers, disabling watchpoint trace as a side effect.
- Clearing the WMSK register such that no watchpoint events are enabled to produce a watchpoint trace message.

Watchpoint trace is effectively suppressed whenever the processor core is in the debug halted or debug stopped state. Instruction jamming operations do not produce any watchpoint trace messages. Whenever the processor core leaves the debug halted state, watchpoint trace enable state reverts to the status of DC1[60].

9.10.16.3 Watchpoint Hit Field

The watchpoint hit field consists of fifteen bits with one bit per watchpoint event. Whenever a watchpoint trace message is generated, the watchpoint hit field of the message includes a 1 for each watchpoint event that occurred at that time and a zero for each event that did not occur. Only watchpoints that are enabled in WMSK may set a bit in the watchpoint hit field.

Most Significant Bit — Least Significant Bit

WP15	WP14	WP13	WP12	WP11	WP10	WP9	WP8	WP7	WP6	WP5	WP4	WP3	WP2	WP1

Figure 9-23. Watchpoint Hit Field

9.10.16.4 Watchpoint Trace Message Events

A watchpoint trace message is generated whenever watchpoint trace is enabled (DC1[60] is set) and a watchpoint event occurs which is enabled to produce a watchpoint trace message (the corresponding WMSK bit is set). If more than one enabled watchpoint occurs in a single cycle, only one watchpoint trace message is generated and multiple bits of the watchpoint hit field is set.

9.11 Performance Monitor

This section describes the performance monitor, which is defined by the architecture and described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. The primary function of the performance monitor is to count events pertaining to the performance of the core (for example, load/store and memory interface activity, cache activity, instructions fetched or executed, branches taken or not taken). Some features are defined by the implementation, in particular, the events that can be counted.

9.11.1 Overview

The performance monitor provides the ability to count predefined events and processor clocks associated with particular operations, for example cache misses, mispredicted branches, or the number of cycles an execution unit stalls. The count of such events can be used to trigger the performance monitor interrupt.

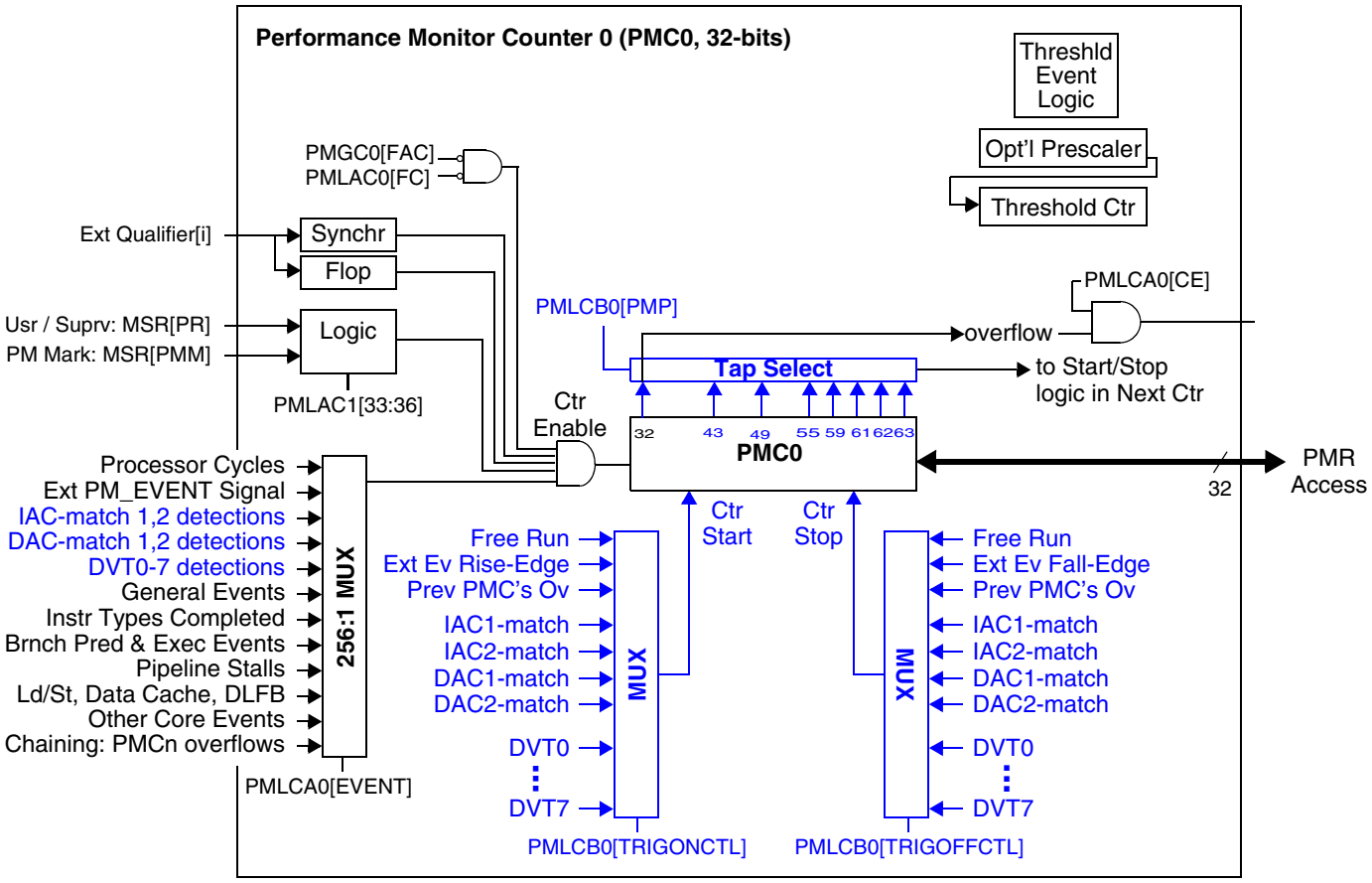
The performance monitor can be used to do the following:

- Improve system performance by monitoring software execution and then recoding algorithms for more efficiency. For example, memory hierarchy behavior can be monitored and analyzed to optimize task scheduling or data distribution algorithms.
- Characterize processors in environments not easily characterized by benchmarking.
- Help system developers bring up and debug their systems.

The performance monitor uses the following resources:

- The performance monitor mark bit, MSR[PMM], controls which programs are monitored.
- The move to/from performance monitor registers (PMR) instructions, **mtpmr** and **mfpmr**.
- The external input, *pm_event*.

Figure 9-24 shows a detailed view of one of the PMC counters available within the core performance monitor. Blue highlights special triggering controls that are available for e500mc.



Freeze *All* Counters via:

Software: PMGC0[FAC]
PMC Overflow: PMGC0[FCECE]
Time Base Hit: PMGC0[FCECE] and PMGC0[TBEE]

Freeze *Individual* Counters via:

Freeze Always: PMLCAn[FC]
Freeze in Supervisor / User Modes: PMLCAn[FCS], PMLCAn[FCU]
Freeze when Mask Set / Cleared: PMLCAn[FCM1], PMLCAn[FCM0]

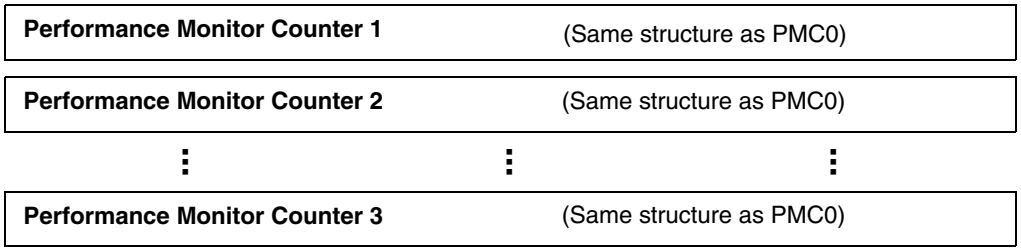


Figure 9-24. Detailed View: Core Performance Monitor Counters 0 through 3

- PMRs:
  - The performance monitor counter registers (PMC0–PMC3) Section 2.18.4, "Performance Monitor Counter Registers (PMC0–PMC3/UPMC0–UPMC3) are 32-bit counters used to

count software-selectable events. Each counter counts up to 256 events.UPMC0–UPMC3 provide user-level read access to these registers. Reference events are those events that are applicable to most microprocessor microarchitectures and are of general value. They are identified in Table 9-47.

— The performance monitor global control register (PMGC0) controls the counting of performance monitor events. It takes priority over all other performance monitor control registers. UPMGC0 provides user-level read access to PMGC0.

— The performance monitor local control registers (PMLCa0–PMLCa3, PMLCb0–PMLCb3) control each individual performance monitor counter. Each counter has a corresponding PMLCa and PMLCb register. UPMLCa0–UPMLCa3 and UPMLCb0–UPMLCb3 provide user-level read access to PMLCa0–PMLCa3, PMLCb0–PMLCb3).

The performance monitor interrupt follows the architecture-defined interrupt model and is briefly described in Section 4.9.17, "Performance Monitor Interrupt—IVOR35."

Software communication with the performance monitor is achieved through PMRs rather than SPRs. The PMRs are used for enabling conditions that can trigger a performance monitor interrupt.

9.11.2 Performance Monitor Instructions

Instructions for reading and writing the PMRs are shown in Table 9-44. These are described in detail in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors*.

Table 9-44. Performance Monitor Instructions

Name	Mnemonic	Syntax
Move from Performance Monitor Register	**mfpmr**	**r**D,PMRN
Move to Performance Monitor Register	**mtpmr**	PMRN,**r**S

9.11.3 Performance Monitor Interrupt

The performance monitor interrupt is triggered by an enabled condition or event. The only enabled condition or event defined for the e500 is the following:

- A PMC*n* overflow condition occurs when both of the following are true:
  — The counter's overflow condition is enabled; PMLCa*n*[CE] is set.
  — The counter indicates an overflow; PMC*n*[OV] is set.

If PMGC0[PMIE] is set, an enabled condition or event triggers the signaling of a performance monitor exception. If PMGC0[FCECE] is set, an enabled condition or event also triggers all performance monitor counters to freeze.

Although the performance monitor exception condition could occur with MSR[EE] cleared, the interrupt cannot be taken until MSR[EE] is set. If PMC*n* overflows, signals an exception (PMLCa*n*[CE] and PMGC0[PMIE] are set) while interrupts are disabled (MSR[EE] is clear), and freezing of the counters is not enabled (PMGC0[FCECE] is clear), PMC*n* can wrap around to all zeros again without the performance monitor interrupt being taken.

9.11.4 Event Counting

This section describes configurability and specific unconditional counting modes.

9.11.4.1 Processor Context Configurability

Counting can be enabled if conditions in the processor state match a software-specified condition. Because a software task scheduler may switch a processor's execution among multiple processes and because statistics on only a particular process may be of interest, a facility is provided to mark a process. The performance monitor mark bit, MSR[PMM], is used for this purpose. System software may set this bit when a marked process is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR[PR,PMM] together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified by PMLCa*n*[FCS,FCU,FCM1,FCM0], the state for which monitoring is enabled, counting is enabled for PMC*n*.

The processor states and the settings of the FCS, FCU, FCM1, and FCM0 fields in PMLCa*n* necessary to enable monitoring of each processor state are shown in Table 9-45.

Table 9-45. Processor States and PMLCa0—PMLCa3 Bit Settings

Processor State	FCS	FCU	FCM1	FCM0
Marked	0	0	0	1
Not marked	0	0	1	0
Supervisor	0	1	0	0
User	1	0	0	0
Marked and supervisor	0	1	0	1
Marked and user	1	0	0	1
Not marked and supervisor	0	1	1	0
Not mark and user	1	0	1	0
All	0	0	0	0
None	X	X	1	1
None	1	1	X	X

Two unconditional counting modes may be specified:

- Counting is unconditionally enabled regardless of the states of MSR[PMM] and MSR[PR]. This can be accomplished by clearing PMLCa*n*[FCS], PMLCa*n*[FCU], PMLCa*n*[FCM1], and PMLCa*n*[FCM0] for each counter control.
- Counting is unconditionally disabled regardless of the states of MSR[PMM] and MSR[PR]. This can be accomplished by setting PMGC0[FAC] or by setting PMLCa*n*[FC] for each counter control. Alternatively, this can be accomplished by setting PMLCa*n*[FCM1] and PMLCa*n*[FCM0] for each counter control or by setting PMLCa*n*[FCS] and PMLCa*n*[FCU] for each counter control.

9.11.4.2 Core Performance Monitor & PC Capture Function

For real-time debug, a capture function is available for the core performance counters, PMC0–PMC3 as well as the micro-architected program counter (PC). Whenever the core's EVTO4 signal is asserted, the PMC values of each counter and the current PC are captured in registers, which are then readable through the e500mc memory mapped interface.

One of the e500mc watchpoint signals, EVTO4, was selected for simultaneous capture of the PMC counters and micro-architected PC because it provided capture capability through programming of DC2[EWC4] for any of the e500mc watchpoint sources including the following:

- EVTI0 signal, providing a device trigger from the EPU through RCPM
- EVTI1 signal, providing a device trigger from the EPU through RCPM
- IAC match
- DAC match
- Other watchpoint sources

This allows for capture either on an external device event through the EVTI signals or on an event internal to the e500mc.

A high-level block diagram of the capture functionality is shown in Figure 9-25. The capture registers are written only from the PMCs on the capture signal and are readable only through the e500mc memory mapped interface. The location of these registers in the memory map are outlined in Table 9-23.

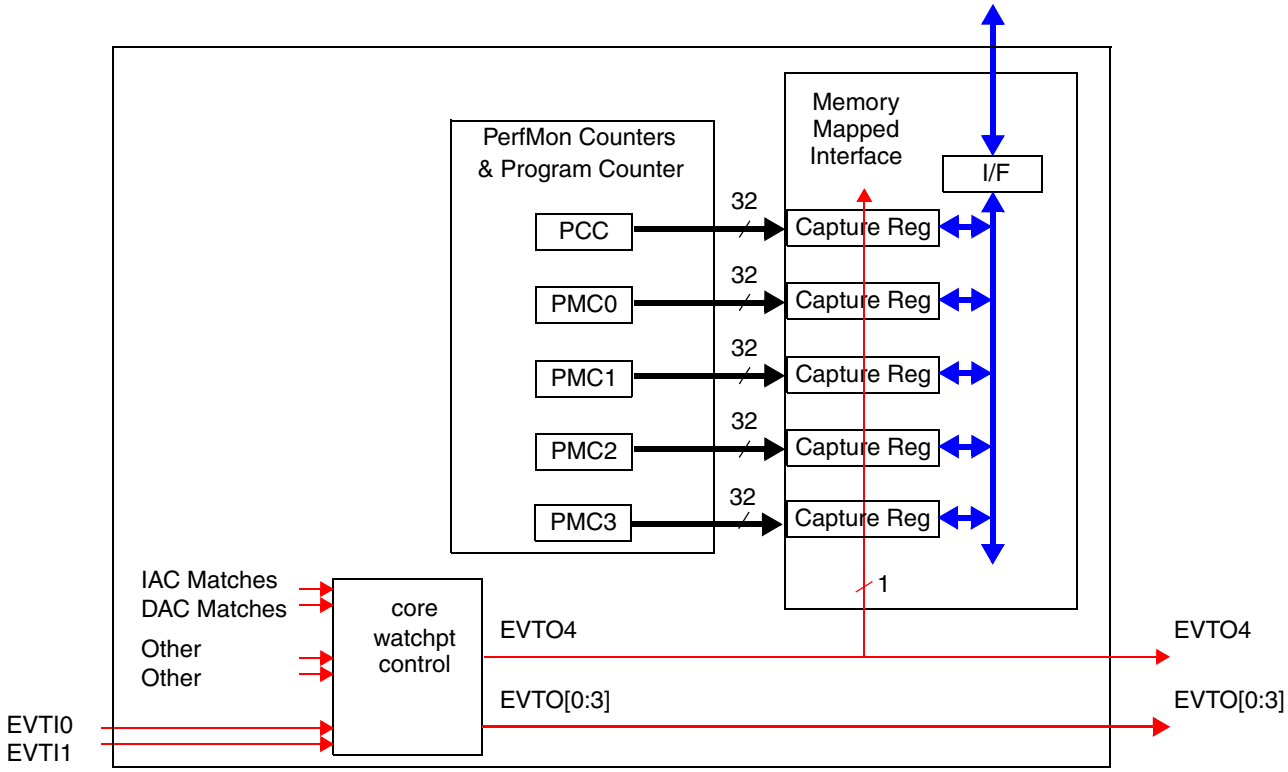


Figure 9-25. Core Performance Monitor Capture Capability

NOTE

Note that the EVTI signal, provided from the SoC, can be used to capture not only PMC counter (& program counter) values from a single core, but from all cores (or a subset of cores) on the SoC as well as the SoC-level performance counters located in the event processing unit (EPU).

9.11.5 Examples

The following sections provide examples of how to use the performance monitor facility.

9.11.5.1 Chaining Counters

The counter chaining feature can be used to decrease the processing pollution caused by performance monitor interrupts (such as cache contamination and pipeline effects) by allowing a higher event count than is possible with a single counter. Chaining two counters together effectively adds 32 bits to a counter register where the first counter's overflow event acts like a carry out feeding the second counter. By defining the event of interest to be another PMC's overflow generation, the chained counter increments each time the first counter rolls over to zero. Multiple counters may be chained together.

Because the entire chained value cannot be read in a single instruction, an overflow may occur between counter reads, producing an inaccurate value. A sequence like the following is necessary to read the complete chained value when it spans multiple counters and the counters are not frozen. The example shown is for a two-counter case.

```
loop:   mfpmr           Rx,pmctr1       #load from upper counter
        mfpmr           Ry,pmctr0       #load from lower counter
        mfpmr           Rz,pmctr1       #load from upper counter
        cmp             cr0,0,Rz,Rx     #see if 'old' = 'new'
        bc              4,2,loop        #loop if carry occurred between reads
```

The comparison and loop are necessary to ensure that a consistent set of values has been obtained. The above sequence is not necessary if the counters are frozen.

9.11.5.2 Thresholding

Threshold event measurement enables the counting of duration and usage events. For example, data line fill buffer (DLFB) load miss cycles (event C0:76 and C1:76) require a threshold value. A DLFB load miss cycles event is counted only when the number of cycles spent recovering from the miss is greater than the threshold. Because this event is counted on two counters and each counter has an individual threshold, one execution of a performance monitor program can sample two different threshold values. Measuring code performance with multiple concurrent thresholds expedites code profiling significantly.

9.11.6 Event Selection

Event selection is specified through the PMLCa*n* registers described in Section 2.18.2, "Local Control A Registers (PMLCa0–PMLCa3/UPMLCa0–UPMLCa3)." The event-select fields in PMLCa*n*[EVENT] are described in Table 9-47, which lists encodings for the selectable events to be monitored. Table 9-47

establishes a correlation between each counter, events to be traced and the pattern required for the desired selection.

For the purposes of event descriptions, the following definitions of micro-ops apply:

- A *micro-op* is defined to be:
  - 2 for load and store instructions that use an update form (such as **lwzu**);
  - 1 to 32 for load and store multiple instructions (**lmw**, **stmw**) depending on the number of registers processed;
  - 1 for all other instructions
- A *store micro-op* is defined to be:
  - 1 to 32 for store multiple instructions (**stmw**) depending on the number of registers processed;
  - 2 for any misaligned store that crosses a doubleword boundary;
  - 1 for all other store instructions including store with update forms;
  - 1 for all other instructions that are treated as a store or are processed as an entry in the store queue by the implementation:
    - **dcba***, **dcbf***, **dcbst***, **dcbz***;
    - **dcbt** (CT=1), **dcbtst** (CT=1);
    - **icbi***;
    - **icbt** (CT=1);
    - **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, **icblc**;
    - **msgsnd**, **mbar**, **sync**, **tlbivax**, **tlbilx**
  - **dcbt*** instructions that are processed as a NOP are not counted
- A *load micro-op* is defined to be:
  - 1 to 32 for load multiple instructions (**lmw**) depending on the number of registers processed;
  - 2 for any misaligned load that crosses a doubleword boundary;
  - 1 for all other load instructions including load with update forms;
  - 1 for all other instructions that are treated as a load by the implementation:
    - **dcbt** (CT=0), **dcbtst** (CT=0)
  - **dcbt*** instructions that are processed as a NOP are not counted
- A *cacheable store micro-op* is defined to be a store micro-op to an address that is marked with WIMGE = 0b00xxx (not write-through and not cacheing inhibited).
- A *cacheable load micro-op* is defined to be a load micro-op to an address that is marked with WIMGE = 0bx0xxx (not cacheing inhibited).

The Spec/Nonspec column indicates whether the event count includes any occurrences due to processing that was not architecturally required by the Power ISA sequential execution model (speculative processing).

- Speculative counts include speculative instructions that were later flushed.
- Nonspeculative counts do not include speculative operations, which are flushed.

Table 9-46 describes how event types are indicated in Table 9-47.

Table 9-46. Event Types

Event Type	Label	Description
Reference	Ref:#	Shared across counters PMC0—PMC3. Applicable to most microprocessors.
Common	Com:#	Shared across counters PMC0–PMC3. Fairly specific to e500 microarchitectures.
Counter-specific	C[0–3]:#	Counted only on one or more specific counters. The notation indicates the counter to which an event is assigned. For example, an event assigned to counter PMC2 is shown as C2:#.

Table 9-47 describes performance monitor events.

Table 9-47. Performance Monitor Event Selection

Number	Event	Spec/ Nonspec	Count Description
General Events			
Ref:0	Nothing	Nonspec	Register counter holds current value
Ref:1	Processor cycles	Nonspec	Every processor cycle
Ref:2	Instructions completed	Nonspec	Completed instructions. 0, 1, or 2 per cycle.
Com:3	Micro-ops completed	Nonspec	Completed micro-ops.
Com:4	Instructions fetched	Spec	Fetched instructions. 0, 1, 2, 3, or 4 per cycle. (instructions written to the IQ.)
Com:5	Micro-ops decoded	Spec	Micro-ops decoded.
Com:6	PM_EVENT transitions	Spec	0 to 1 transitions on the *pm_event* input.
Com:7	PM_EVENT cycles	Spec	Processor cycles that occur when the *pm_event* input is asserted.
Instruction Types Completed			
Com:8	Branch instructions completed	Nonspec	Completed branch instructions.
Com:9	Load micro-ops completed	Nonspec	Completed load micro-ops.
Com:10	Store micro-ops completed	Nonspec	Completed store micro-ops.
Com:11	Number of CQ redirects	Nonspec	Fetch redirects initiated from the completion unit. (for example, resulting from **sc**, **rfi**, **rfci**, **rfdi**, **rfmci**, **isync**, and interrupts)
Branch Prediction and Execution Events			
Com:12	Branches finished	Spec	Includes all branch instructions
Com:13	Taken branches finished	Spec	Includes all taken branch instructions
Com:14	Finished unconditional branches that miss the BTB	Spec	Includes all taken branch instructions not allocated in the BTB
Com:15	Branches mispredicted (for any reason)	Spec	Counts branch instructions mispredicted due to direction, target (for example if the CTR contents change), or IAB prediction. Does not count instructions that the branch predictor incorrectly predicted to be branches.
Com:16	Branches in the BTB mispredicted due to direction prediction.	Spec	Counts branch instructions mispredicted due to direction prediction.

Table 9-47. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:17	BTB hits and pseudo-hits	Spec	Branch instructions that hit in the BTB or miss in the BTB and are not-taken (a pseudo-hit). Characterizes upper bound on prediction rate.
Pipeline Stalls			
Com:18	Cycles decode stalled	Spec	Cycles the IQ is not empty but 0 instructions decoded
Com:19	Cycles issue stalled	Spec	Cycles the issue buffer is not empty but 0 instructions issued
Com:20	Cycles branch issue stalled	Spec	Cycles the branch buffer is not empty but 0 instructions issued
Com:21	Cycles SFX0 schedule stalled	Spec	Cycles SFX0 is not empty but 0 instructions scheduled
Com:22	Cycles SFX1 schedule stalled	Spec	Cycles SFX1 is not empty but 0 instructions scheduled
Com:23	Cycles MU schedule stalled	Spec	Cycles MU is not empty but 0 instructions scheduled
Com:24	Cycles ~~LRU~~ schedule stalled	Spec	Cycles ~~LRU~~ is not empty but 0 instructions scheduled
Com:25	Cycles BU schedule stalled	Spec	Cycles BU is not empty but 0 instructions scheduled
Load/Store, Data Cache, and Data Line Fill Buffer (DLFB) Events			
Com:26	Total translated	Spec	Total of load and store micro-ops that reach the second stage of the LSU [1]
Com:27	Loads translated	Spec	Cacheable load micro-ops translated.[1]
Com:28	Stores translated	Spec	Cacheable store micro-ops translated.[1]
Com:29	Touches translated	Spec	Cacheable **dcbt** and **dcbtst** instructions translated (CT = 0 only) (Doesn't count touches that are converted to NOPs)
Com:30	Cacheops translated	Spec	**dcba***, **dcbf***, **dcbi**, **dcbst***, and **dcbz*** instructions translated.
Com:31	Cache-inhibited accesses translated	Spec	Cache inhibited accesses translated
Com:32	Guarded loads translated	Spec	Guarded loads translated
Com:33	Write-through stores translated	Spec	Write-through stores translated
Com:34	Misaligned load or store accesses translated	Spec	Misaligned load or store accesses translated.
Com:35	Total allocated to DLFB	Spec	—
Com:36	Loads translated and allocated to DLFB	Spec	Applies to same class of instructions as loads translated.
Com:37	Stores completed and allocated to DLFB	Nonspec	Applies to same class of instructions as stores translated.
Com:38	Touches translated and allocated to DLFB	Spec	Applies to same class of instructions as touches translated.
Com:39	Stores performed	Nonspec	Cacheable store micro-ops completed
Com:40	Data L1 cache locks	Nonspec	Cache lines locked in the data L1 cache. (Counts a lock even if an overlock condition occurs.)

Table 9-47. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:41	Data L1 cache reloads	Spec	Counts cache reloads for any reason. Typically used to determine data cache miss rate (along with loads/stores completed).
Com:42	Data L1 cache castouts	Spec	Does not count castouts due to **dcbf***.
Data Side Replay Conditions: Times Detected			
Com:43	Load miss with DLFB full.	Spec	Counts number of stalls; Com:51 counts cycles stalled.
Com:44	Load miss with load queue full.	Spec	Counts number of stalls; Com:52 counts cycles stalled.
Com:45	Load guarded miss when the load is not yet at the bottom of the CQ.	Spec	Counts number of stalls; Com:53 counts cycles stalled.
Com:46	Translate a store when the store queue is full.	Spec	Counts number of stalls; Com:54 counts cycles stalled.
Com:47	Address collision.	Spec	Counts number of stalls; Com:55 counts cycles stalled.
Com:48	Data MMU miss.	Spec	Counts number of stalls; Com:56 counts cycles stalled.
Com:49	Data MMU busy.	Spec	Counts number of stalls; Com:57 counts cycles stalled.
Com:50	Second part of misaligned access when first part missed in cache.	Spec	Counts number of stalls; Com:58 counts cycles stalled.
Data Side Replay Conditions: Cycles Stalled			
Com:51	Load miss with DLFB full.	Spec	Counts cycles stalled; Com:43 counts number of stalls.
Com:52	Load miss with load queue full.	Spec	Counts cycles stalled; Com:44 counts number of stalls.
Com:53	Load guarded miss when the load is not yet at the bottom of the CQ.	Spec	Counts cycles stalled; Com:45 counts number of stalls.
Com:54	Translate a store when the store queue is full.	Spec	Counts cycles stalled; Com:46 counts number of stalls.
Com:55	Address collision.	Spec	Counts cycles stalled; Com:47 counts number of stalls.
Com:56	Data MMU miss.	Spec	Counts cycles stalled; Com:48 counts number of stalls.
Com:57	Data MMU busy.	Spec	Counts cycles stalled; Com:49 counts number of stalls.
Com:58	Second part of misaligned access when first part missed in cache.	Spec	Counts cycles stalled; Com:50 counts number of stalls.
Fetch, Instruction Cache, Instruction Line Fill Buffer (ILFB), and Instruction Prefetch Events			
Com:59	Instruction L1 cache locks	Nonspec	Counts cache lines locked in the instruction L1 cache. (Counts a lock even if an overlock occurs.)
Com:60	Instruction L1 cache reloads from fetch	Spec	Counts reloads due to demand fetch. Typically used to determine instruction cache miss rate (along with instructions completed)
Com:61	Number of fetches	Spec	Counts fetches that write at least one instruction to the IQ. (With instruction fetched (com:4), can used to compute instructions-per-fetch)
Instruction MMU, Data MMU and L2 MMU Events			

Table 9-47. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:62	Instruction MMU TLB4K reloads	Spec	Counts reloads in the level 1 instruction MMU TLB4K.þA reload in the level 2 MMU TLB4Kis not counted.
Com:63	Instruction MMU VSP reloads	Spec	Counts reloads in the level 1 instruction MMU VSP.þA reload in the level 2 MMU VSP is not counted.
Com:64	Data MMU TLB4K reloads	Spec	Counts reloads in the level 1 data MMU TLB4K.þA reload in the level 2 MMU TLB4K is not counted.
Com:65	Data MMU VSP reloads	Spec	Counts reloads in the level 1 data MMU VSP. A reload in the level 2 MMU VSP is not counted.
Com:66	L2MMU misses	Nonspec	Counts instruction TLB/data TLB error interrupts
BIU Interface Usage			
Com:67	BIU master requests	Spec	Master transaction starts (number of Aout sent to CoreNet)
Com:68	BIU master global requests	Spec	Master transaction starts that are global (M=1) (number of Aout with M=1 sent to CoreNet). For e500mc Rev 1.x and Rev 2.x this event is not supported.
Com:69	BIU master data-side requests	Spec	Master data-side transaction starts (number of D-side Aout sent to CoreNet)
Com:70	BIU number of stash requests received	Spec	Stash request on Ain matches stash IDs for the core and are sent to LFB. For e500mc Rev 1.x and Rev 2.x this event is not supported.
Com:71	BIU number of stash accepts	N/A	LFB signals snarf snoop response for ACRout for stash request. For e500mc Rev 1.x and Rev 2.x this event is not supported.
Snoop			
Com:72	Snoop requests	N/A	Externally generated snoop requests. (number of Ain from CoreNet not from self)
Com:73	Snoop hits	N/A	Snoop hits on all data-side resources regardless of the cache state (modified, shared, or exclusive)
Com:74	Snoop pushes	N/A	Snoop pushes from all data-side resources. (Number of ACRout to CoreNet for any snoop push). For e500mc Rev 1.x and Rev 2.x this event is not supported.
Com:75	Snoop sharing	N/A	Number of ACRout when the core retains a copy of the coherency granule. For e500mc Rev 1.x and Rev 2.x this event is not supported.
Threshold Events			
C0:76 C1:76	Data line fill buffer load miss cycles	Spec	Instances when the number of cycles between a load allocation in the data line fill buffer (entry 0) and write-back to the data L1 cache exceeds the threshold.
C0:77 C1:77	ILFB fetch miss cycles	Spec	Instances when the number of cycles between allocation in the ILFB (entry 0) and write-back to the instruction L1 cache exceeds the threshold.

Table 9-47. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
C0:78 C1:78	External input interrupt latency cycles	N/A	Instances when the number of cycles between request for interrupt (*int*) asserted (but possibly masked/disabled) and redirecting fetch to external interrupt vector exceeds threshold.
C0:79 C1:79	Critical input interrupt latency cycles	N/A	Instances when the number of cycles between request for critical interrupt (*cint*) is asserted (but possibly masked/disabled) and redirecting fetch to the critical interrupt vector exceeds threshold.
C0:80 C1:80	External input interrupt pending latency cycles	N/A	Instances when the number of cycles between external interrupt pending (enabled and pin asserted) and redirecting fetch to the external interrupt vector exceeds the threshold. Note that this and the next event may count multiple times for a single interrupt if the threshold is very small and the interrupt is masked a few cycles after it is asserted and later becomes unmasked.
C0:81 C1:81	Critical input interrupt pending latency cycles	N/A	Instances when the number of cycles between pin request for critical interrupt pending (enabled and pin asserted) and redirecting fetch to the critical interrupt vector exceeds the threshold. See note for previous event.
Chaining Events[2]			
Com:82	PMC0 overflow	N/A	PMC0[32] transitions from 1 to 0.
Com:83	PMC1 overflow	N/A	PMC1[32] transitions from 1 to 0.
Com:84	PMC2 overflow	N/A	PMC2[32] transitions from 1 to 0.
Com:85	PMC3 overflow	N/A	PMC3[32] transitioned from 1 to 0.
Interrupt Events			
Com:86	Interrupts taken	Nonspec	—
Com:87	External input interrupts taken	Nonspec	—
Com:88	Critical input interrupts taken	Nonspec	—
Com:89	System call and trap interrupts	Nonspec	—
Misc Events			
Com:90	Transitions of TBL bit selected by PMGC0[TBSEL].	Nonspec	Counts transitions of the TBL bit selected by PMGC0[TBSEL].
Com:91	L2 linefill buffer	—	Linefill requests to L2
Com:92	L2 Vs	—	Victim selects to L2
Com:93	Castouts released	—	Speculative reservations in castout buffer that are not needed
Com:94	INTV allocations	—	Allocations to INTV queue
Com:95	Store retries to MBAR	—	DLFB retries to MBAR
Com:96	Store retries due to misc	—	Retries to store queue, excluding MBAR case
Stashing Events			
Com:97	Stash L1 hit	N/A	Stash hits in L1

Table 9-47. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:98	Stash L2 hit	N/A	Stash hits in L2
Com:99	Stash busy 1	N/A	Cycles stash 1 DLFB busy
Com:100	Stash busy 2	N/A	Cycles stash 2 DLFB's busy
Com:101	Stash busy 3	N/A	Cycles stash 3 DLFB's busy
Com:102	Stash hit	N/A	Access hits on stash DLFB
Com:103	Stash hit DLFB	N/A	Stash hits to a DLFB
Com:106	Stash requests	N/A	Stash requests
Com:107	Stashes to L1 data cache	N/A	Stash requests to L1
Com:108	Stashes to backside L2	N/A	Stash requests to L2
Com:109	Stalls due to no CAQ or COB	—	Stalls due to no CAQ or COB
Backside L2 Events			
Com:110	L2 cache accesses	Spec	L2 cache accesses, which include the following: *load*, *store*, fetch, **dcba***, **dcbz***, **dcblc** CT = 1, **icblc** CT=1, **dcblc** CT=2, **icblc** CT=2, **dcbf***, **dcbst***, **dcbi**, CI store, **icbi***, **lwarx**, **stwcx.**, write-though store, CI **stwcx.**, **mbar**, **sync**, **tlbsync**, **tlbivax**, **tlbilx**, prefetch requests(**dcbt**, **dcbtst**, **dcbtls**, **dcbtstls** CT = 0,1,2 & **icbt** CT=1, 2 & **icbtls** CT = 0,1,2), L2 cache allocation
Com:111	L2 hit cache accesses	Spec	L2 cache accesses, which include the following: *load*, *store*, fetch, **dcba***, **dcbz***, **dcblc** CT = 1, **icblc** CT=1, **dcblc** CT=2, **icblc** CT=2, **dcbf***, **dcbst***, **dcbi**, CI store, **icbi***, **lwarx**, **stwcx.**, write-though store, CI **stwcx.**, **mbar**, **sync**, **tlbsync**, **tlbivax**, **tlbilx**, prefetch requests(**dcbt**, **dcbtst**, **dcbtls**, **dcbtstls** CT = 0,1,2 & **icbt** CT=1, 2 & **icbtls** CT = 0,1,2) && L2_hit
Com:112	L2 cache data accesses	Spec	L2 cache data accesses
Com:113	L2 cache data hits	Spec	L2 cache data hits
Com:114	L2 cache instruction accesses	Spec	L2 cache instruction accesses
Com:115	L2 cache instruction hits	Spec	L2 cache instruction hits
Com:116	L2 cache allocations	Spec	L2 cache allocations: castouts from L1 (clean or modified), **dcbt**, **dcbtst** CT=2, **dcbtls**, **dcbtstls** CT=2, **icbt** CT=2, **icbtls** CT=2, fetch
Com:117	L2 cache data allocations	Spec	L2 cache data allocations castouts from L1, **dcbt**/**dcbtst** CT=2, **dcbtls**/**dcbtstls** CT=2
Com:118	L2 cache dirty data allocations	Spec	L2 cache dirty data allocations
Com:119	L2 cache instruction allocations	Spec	L2 cache instruction allocations fetch, **icbtls** CT=2, **icbt** CT=2
Com:120	L2 cache Updates	Spec	L2 cache updates
Com:121	L2 cache clean updates	Spec	L2 cache clean updates
Com:122	L2 cache dirty updates	Spec	L2 cache dirty updates

Table 9-47. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:123	L2 cache clean redundant updates	Spec	L2 cache clean redundant updates
Com:124	L2 cache dirty redundant updates	Spec	L2 cache dirty redundant updates
Com:125	L2 cache locks	Spec	L2 cache locks
Com:126	L2 cache castouts	Spec	L2 cache castouts
Com:127	L2 cache data dirty hits	Spec	L2 cache data dirty hits
Com:128	Instruction lfb went high priority	Spec	Instruction lfb went high priority
Com:129	Snoop throttling turned on	Spec	Snoop throttling turned on
Com:130	L2 invalidation of clean lines	Spec	L2 invalidation of clean lines
Com:131	L2 invalidation of Incoherent line	Spec	L2 invalidation of Incoherent line
Com:132	L2 invalidation of coherent line	Spec	L2 invalidation of coherent line
Com:133	Coherent lookup miss due to valid but incoherent (address matches)	Spec	Coherent lookup miss due to valid but incoherent (address matches)
IAC, DAC Events			
Com: 140	IAC1s detected	Nonspec	Every valid IAC1 detection
Com: 141	IAC2s detected	Nonspec	Every valid IAC2 detection
Com: 142	(reserved)	Nonspec	—
Com: 143	(reserved)	Nonspec	—
Com: 144	DAC1s detected	Nonspec	Every valid DAC1 detection
Com: 145	DAC2s detected	Nonspec	Every valid DAC2 detection
Com: 146	(reserved)	Nonspec	—
Com: 147	(reserved)	Nonspec	—
DVT Events			
Com: 148	DVT0 detected	Nonspec	Detection of a write to DEVENT SPR with DVT0 set
Com: 149	DVT1 detected	Nonspec	Detection of a write to DEVENT SPR with DVT1 set
Com: 150	DVT2 detected	Nonspec	Detection of a write to DEVENT SPR with DVT2 set
Com: 151	DVT3 detected	Nonspec	Detection of a write to DEVENT SPR with DVT3 set
Com: 152	DVT4 detected	Nonspec	Detection of a write to DEVENT SPR with DVT4 set
Com: 153	DVT5 detected	Nonspec	Detection of a write to DEVENT SPR with DVT5 set
Com: 154	DVT6 detected	Nonspec	Detection of a write to DEVENT SPR with DVT6 set
Com: 155	DVT7 detected	Nonspec	Detection of a write to DEVENT SPR with DVT7 set
Com: 156	Cycles completion stalled (Nexus)	Nonspec	Number of completion cycles stalled due to Nexus FIFO full
FPU Events			

Table 9-47. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
Com: 160	FPU double pump	Spec	Double pump penalized ops finished through the pipe. Counts once for every multiply family double pump operation
Com: 161	FPU finish	Spec	—
Com: 162	FPU divide cycles	Spec	Counts once for every cycle of divide execution. (**fdivs** and **fdiv**)
Com: 163	FPU denorm input	Spec	Counts extra cycles delay due to denormalized inputs. If there is one, this is incremented 4 times, Two operands increments it 5 times. This shows the real penalty due to denorms, not just how often they occur.
Com: 164	FPU result stall	Spec	Counts extra cycles due to denorm results, overflow, mass cancellation, zero results, carry-in mispredict, exponent range check.
Com: 165	FPU FPSCR full stall	Spec	—
Com: 166	FPU pipe sync stall	Spec	Synchronization-op stalls: count once for each cycle that a “break-before” FPU is in the RS/issue stage but cannotissue. Also count once for each cycle that an FPU op is in the RS/issue stage but cannot issue due to “break-after”: of an FPU op currently in progress.
Com: 167	FPU input data stall	Spec	FPU data-ready stall: cycles in which there is an op in the RS/issue stage that cannot issue because one or more of its operands is not yet available.
Extended Load Store Events			
Com: 176	Decorated loads	Nonspec	Number of decorated loads to cache inhibited memory performed
Com: 177	Decorated stores	Nonspec	Number of decorated stores to cache inhibited memory performed
Com: 178	Load Retries	Nonspec	Number of load retries
Com: 179	**stwcx.** successes	Nonspec	Number of successful **stwcx.** instructions
Com: 180	**stwcx.** unsuccessful	Nonspec	Number of unsuccessful **stwcx.** instructions

1 For load/store events, a micro-op is described as translated when the micro-op has successfully translated and is in the second stage of the load/store translate pipeline.

2 For chaining events, if a counter is configured to count its own overflow bit, that counter does not increment. For example, if PMC2 is selected to count PMC2 overflow events, PMC2 does not increment.

Chapter 10
Execution Timing

This chapter provides an overview of how the e500mc core performs operations defined by instructions and how it reports the results of instruction execution. It provides a high-level description about how the core execution units work and how these units interact with other parts of the processor, such as the instruction fetching mechanism, cache register files, and other architected registers. It includes tables that identify the unit that executes each instruction implemented on the core, the latency for each instruction, and other information useful to assembly language programmers.

10.1 Terminology and Conventions

This section provides an alphabetical list of terms used in this chapter. These definitions offer a review of commonly used terms and point out specific ways in which these terms are used in this chapter.

NOTE

Please read this list carefully. Some definitions differ slightly from those used to describe previous processors and, in particular, with respect to dispatch, issue, finishing, retirement, and write back.

Branch prediction	The process of predicting the direction and target of a branch. Branch direction prediction involves guessing whether a branch will be taken. Branch target prediction involves guessing the target address of a branch. The e500mc does not use the architecture-defined hint bits in the BO operand for static prediction. Clearing BUCSR[BPEN] disables dynamic branch prediction; in this case the e500mc predicts every branch as not taken.
Branch resolution	The determination of whether a branch prediction is correct. If it is, instructions following the predicted branch that may have been speculatively executed can complete (*see* Complete). If it is incorrect, the processor redirects fetching to the proper path and marks instructions on the mispredicted path (and any of their results) for purging when the mispredicted branch completes.
Complete	An instruction is eligible to complete after it finishes executing and makes its results available for subsequent instructions. Instructions must complete in order from the bottom two entries of the completion queue (CQ). The completion unit coordinates how instructions (which may have executed out of order) affect architected registers to ensure the appearance of serial execution. This guarantees that the completed instruction and all previous instructions can cause no exceptions. An instruction completes when it is retired, that is, deleted from the CQ.
Decode	The decode stage determines the issue queue to which each instruction is dispatched (see Dispatch) and determines whether the required space is available

	in both that issue queue and the completion queue. If space is available, it decodes instructions supplied by the instruction queue, renames any source/target operands, and dispatches them to the appropriate issue queues.
Dispatch	Dispatch is the event at the end of the decode stage during which instructions are passed to the issue queues and tracking of program order is passed to the completion queue.
Fetch	The process of bringing instructions from memory (such as a cache or system memory) into the instruction queue.
Finish	An executed instruction finishes by signaling the completion queue that execution has concluded. An instruction is said to be finished (but not complete) when the execution results have been saved in rename registers and made available to subsequent instructions, but the completion unit has not yet updated the architected registers.
Issue	The stage responsible for reading source operands from rename registers and register files. This stage also assigns instructions to the proper execution unit.
Latency	The number of clock cycles necessary to execute an instruction and make the results of that execution available to subsequent instructions.
Pipeline	In the context of instruction timing, this term refers to interconnected stages. The events necessary to process an instruction are broken into several cycle-length tasks to allow work to be performed on several instructions simultaneously—analogous to an assembly line. As an instruction is processed, it passes from one stage to the next. When work at one stage is done and the instruction passes to the next stage, another instruction can begin work in the vacated stage. Although an individual instruction may have multiple-cycle latency, pipelining makes it possible to overlap processing so the number of instructions processed per clock cycle (throughput) is greater than if pipelining were not implemented.
Program order	The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache.
Rename registers	Temporary buffers for holding results of instructions that have finished execution but have not completed. The ability to forward results to rename registers allows subsequent instructions to access the new values before they have been written back to the architectural registers.
Reservation station	A buffer between the issue and execute stages that allows instructions to be issued even though resources necessary for execution or results of other instructions on which the issued instruction may depend are not yet available.
Retirement	Removal of a completed instruction from the completion queue at the end of the completion stage. (In other documents, this is often called deallocation.)
Speculative instruction	Any instruction that is currently behind an older branch instruction that has not been resolved.

Stage	Used in two different senses, depending on whether the pipeline is being discussed as a physical entity or a sequence of events. As a physical entity, a stage can be viewed as the hardware that handles operations on an instruction in that part of the pipeline. When viewing the pipeline as a sequence of events, a stage is an element in the pipeline during which certain actions are performed, such as decoding the instruction, performing an arithmetic operation, or writing back the results. Typically, the latency of a stage is one processor clock cycle. Some events, such as dispatch, write-back, and completion, happen instantaneously and may be thought to occur at the end of a stage. An instruction can spend multiple cycles in one stage; for example, a divide takes multiple cycles in the execute stage. An instruction can also be represented in more than one stage simultaneously, especially in the sense that a stage can be seen as a physical resource. For example, when instructions are dispatched, they are assigned a place in the CQ at the same time they are passed to the issue queues.
Stall	An occurrence when an instruction cannot proceed to the next stage. Such a delay is initiated to resolve a data or resource hazard, that is, a situation in which a planned instruction cannot execute in the proper clock cycle because data or resources needed to process the instruction are not yet available.
Superscalar	A superscalar processor is one that can issue multiple instructions concurrently from a conventional linear instruction stream. In a superscalar implementation, multiple instructions can execute in parallel at the same time.
Throughput	The number of instructions processed per cycle. In particular, throughput describes the performance of a multiple-stage pipeline where a sequence of instructions may pass through with a throughput that is much faster than the latency of an individual instruction.
Write-back	Write-back (in the context of instruction handling) occurs when a result is written into the architecture-defined registers (typically the GPRs). On the e500mc, write-back occurs in the clock cycle after the completion stage. Results in the write-back buffer cannot be flushed. If an exception occurs, results from previous instructions must write back before the exception is taken.

10.2 Instruction Timing Overview

The e500mc design minimizes the number of clock cycles it takes to fetch, decode, dispatch, issue, and execute instructions and to make the results available for a subsequent instruction. To improve throughput, the e500mc implements pipelining, superscalar instruction issue, and multiple execution units that operate independently and in parallel.

Some instructions, such as loads and stores, access memory and require additional clock cycles between the execute and write-back phases. Latencies may be greater if the access is to noncacheable memory, causes a TLB miss, misses in the L1 cache, generates a write-back to memory, causes a snoop hit from another device that generates additional activity, or encounters other conditions that affect memory accesses.

The e500mc can complete as many as two instructions on each clock cycle.

The instruction pipeline stages are described as follows:

- Instruction fetch—Includes the clock cycles necessary to request an instruction and the time the memory system takes to respond to the request. Fetched instructions are latched into the instruction queue (IQ) for consideration by the dispatcher.

  The fetcher tries to initiate a fetch in every cycle in which it is guaranteed that the IQ has room for fetched instructions. Instructions are typically fetched from the L1 instruction cache; if caching is disabled or the fetch misses in the cache, instructions are fetched from the instruction line fill buffer (ILFB). Likewise, on a cache miss, as many as four instructions can be forwarded to the fetch unit from the ILFB as the cache line is passed to the instruction cache.

  Fetch timing is affected by many things, such as whether an instruction is in the on-chip instruction cache or an L2 cache. Those factors increase when it is necessary to fetch instructions from system memory and include the processor-to-bus clock ratio, the amount of bus traffic, and whether any cache coherency operations are required.

  Fetch timing is also affected by whether effective address translation is available in a TLB, as described in Section 10.3.1.1, "L1 and L2 TLB Access Times."
- The decode/dispatch stage fully decodes each instruction; most instructions are dispatched to the issue queues, but **isync**, **rfi**, **rfgi**, **rfci**, **rfdi**, **rfmci**, **sc**, **ehpriv**, **dnh**, **wait**, and **nop**s are not. Every dispatched instruction is assigned a GPR rename register, an FPR rename register, and a CR field rename register, even if they do not specify a GPR, FPR, or CR operand. There is a set of GPR/FPR/CRF rename registers for each CQ entry.

  The three issue queues, BIQ, GIQ, and FIQ, can accept as many as one, two, and two instructions, respectively, in a cycle. Instruction dispatch requires the following:

  — Instructions dispatch only from IQ0 and IQ1.

  — As many as two instructions can be dispatched per clock cycle.

  — Space must be available in the CQ and the target issue queue for an instruction to decode and dispatch.

  — Dispatch is in order, if IQ0 cannot dispatch, IQ1 will not dispatch.

  In this chapter, dispatch is treated as an event at the end of the decode stage.
- The issue stage reads source operands from rename registers and register files and determines when instructions are latched into reservation stations.

  The general behavior of the issue queues is described as follows:

  — The GIQ accepts as many as two instructions from the dispatch unit per cycle. SFX0, SFX1, CFX, and all LSU instructions are dispatched to the GIQ, shown in Figure 10-1.

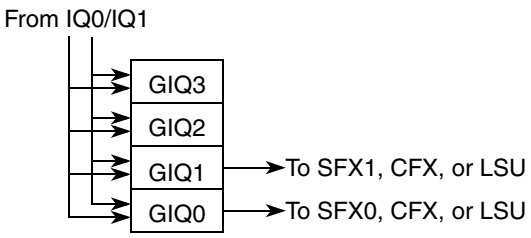


Figure 10-1. GPR Issue Queue (GIQ)

The GIQ can hold up to four instructions.

Instructions can be issued out-of-order from GIQ1–GIQ0. GIQ0 can issue to SFX0, CFX, and LSU. GIQ1 can issue to SFX1, CFX, and LSU.

SFX1 executes a subset of the instructions that can be executed in SFX0. The ability to identify and dispatch instructions to SFX1 increases the availability of SFX0 to execute more computationally intensive instructions.

An instruction in GIQ1 destined for SFX1 or the LSU need not wait for an CFX instruction in GIQ0 that is stalled behind a long-latency divide.

— The FIQ accepts as many as two instructions from the dispatch unit per cycle. FPU instructions are dispatched to the FIQ.

The FIQ can hold up to four instructions.

Instructions are issued in-order from the FIQ to the FPU and can issue at a rate of one per cycle.

— The BIQ accepts as many as one instruction from the dispatch unit per cycle. BU instructions are dispatched to the BIQ.

The BIQ can hold up to two instructions.

Instructions are issued in-order from the BIQ to the BU and can issue at a rate of one per cycle.

- The execute stage is comprised of individual nonblocking execution units implemented in parallel. Each execution unit has a reservation station that must be available for an instruction issue to occur. In most cases, instructions are issued both to the reservation station and to the execution unit simultaneously. However, under some circumstances, an instruction may issue only to a reservation station.

  In this stage, operands assigned to the execution stage are latched.

  The e500mc has the following execution units:

  — Branch unit (BU)—executes branches and CR logical operations

  — Floating-point unit (FPU)—executes FPR-based floating point computational instructions. Floating-point load and store instructions execute in the LSU.

  — Load/store unit (LSU)—executes loads from and stores to memory, as well as some MMU control, cache control, and cache locking instructions. This includes byte, halfword, word, and doubleword instructions.

  — Two simple units (SFX0 and SFX1)—execute move to/from SPR instructions, logical instructions, and all integer computational instructions except multiply and divide instructions.

  - SFX0 executes all integer simple unit instructions (that is, all that can be dispatched to simple units).
  - SFX1 executes most, but not all of the instructions that can be executed in SFX0.

  Most SFX instructions execute in 1 cycle. However some instructions can take more than 1 cycle.
- Complex unit (CFX) executes integer multiplication and division instructions.

The execution unit executes the instruction (perhaps over multiple cycles), writes results on its result bus, and notifies the CQ when the instruction finishes. The execution unit reports any exceptions to the completion stage. Instruction-generated exceptions are not taken until the excepting instruction is next to retire.

Most integer instructions have a 1-cycle latency, so results of these instructions are available 1 clock cycle after an instruction enters the execution unit. The LSU, FPU, and CFX are pipelined.

- The complete and write-back stages maintain the correct architectural machine state and commit results to the architecture-defined registers in the proper order. If completion logic detects an instruction containing an exception status or a mispredicted branch, all following instructions are cancelled, their execution results in rename registers are discarded, and the correct instruction stream is fetched.

  The complete stage ends when the instruction is retired. Two instructions can be retired per clock cycle. If no dependencies exist, as many as two instructions are retired in program order. The write-back stage occurs in the clock cycle after the instruction is retired.

10.3 General Timing Considerations

As many as four instructions can be fetched to the IQ during each clock cycle. Two instructions per clock cycle can be dispatched to the issue queues. Two instructions from the GIQ, one instruction from the FIQ, and one instruction from the BIQ can issue per clock cycle to the appropriate execution units. Two instructions can retire and two can write back per cycle.

The e500mc executes multiple instructions in parallel, using hardware to handle dependencies. When an instruction is issued, source data is provided to the appropriate reservation station from either the architected register (GPR, FPR, or CRF) or from a rename register.

Branch prediction is performed in parallel with the fetch stages using the branch prediction unit (BPU), which incorporates the branch target buffer (BTB). Predictions are resolved in the branch unit (BU). Incorrect predictions are handled as follows:

1. Fetch is redirected to the correct path, and mispredicted instructions are purged.
2. The mispredicted branch is marked as such in the CQ.
3. Eventually, the branch is retired and the CQ, issue queue, and execution units are flushed. If the correct-path instructions reach the IQ before the back half of the pipeline is flushed, they stall in the IQ until the flush occurs.

After an instruction executes, results are made available to subsequent instructions in the appropriate rename registers. The architecture-defined GPRs, FPRs, and CRs are updated in the write-back stage. Branch instructions that update LR or CTR write back in a similar fashion.

If a later instruction needs the result as a source operand, the result is simultaneously made available to the appropriate execution unit, which allows a data-dependent instruction to be decoded and dispatched without waiting to read the data from the architected register file. Results are then stored into the correct architected GPR, CR, or FPR during the write-back stage. Branch instructions that update either the LR or CTR write back their results in a similar fashion.

To resolve branch instructions and improve the accuracy of branch predictions, the e500mc implements a dynamic branch prediction mechanism using the 512-entry BTB, a four-way set associative cache of branch target effective addresses. A BTB entry is allocated whenever a branch resolves as taken—unallocated branches are always predicted as not taken. Each BTB entry holds a 2-bit saturating branch history counter whose value is incremented or decremented depending on whether the branch was taken. These bits can take four values: strongly taken, weakly taken, weakly not taken, and strongly not taken. This mechanism is described in Section 10.4.1.2, "BTB Branch Prediction and Resolution."

The e500mc ignores static branch prediction hints; *a* and *t* bits in the BO field in branch encodings are ignored.

Dynamic branch prediction is enabled by setting BUCSR[BPEN]. Clearing BUCSR[BPEN] disables dynamic branch prediction, in which case the e500mc predicts every branch as not taken.

Branch instructions are treated like any other instruction and are assigned CQ entries to ensure that the CTR and LR are updated sequentially.

The dispatch rate is affected by the serializing behavior of some instructions and the availability of issue queues and CQ entries. Instructions are dispatched in program order; an instruction in IQ1 cannot be dispatched ahead of one in IQ0.

10.3.1 Instruction Fetch Timing Considerations

Instruction fetch latency depends on the following factors:

- Whether the page translation for the effective address of an instruction fetch is in a TLB. This is described in Section 10.3.1.1, "L1 and L2 TLB Access Times."
- If a page translation is not in a TLB, an instruction TLB miss interrupt is taken. Section 10.3.1.2, "Interrupts Associated with Instruction Fetching," describes other conditions that cause an instruction fetch to take an interrupt.
- If an L1 instruction cache miss and an L2 cache miss occurs, a memory transaction is required in which fetch latency is affected by bus traffic and bus clock speed. These issues are discussed further in Section 10.3.1.3, "Cache-Related Latency."

10.3.1.1 L1 and L2 TLB Access Times

The L1 TLB arrays are checked for a translation hit in parallel with the on-chip L1 cache lookups and incur no penalty on an L1 TLB hit. If the L1 TLB arrays miss, the access proceeds to the L2 TLB arrays. For L1 instruction address translation misses, the L2 TLB latency is at least 5 clocks; for L1 data address translation misses, the L2 TLB latency is at least 5 clocks. These access times may be longer, depending on arbitration performed by the L2 arrays for simultaneous instruction L1 TLB misses, data L1 TLB

misses, the execution of TLB instructions, and TLB snoop operations (snooping of TLB invalidate operations from **tlbivax** instructions on CoreNet).

Note that when a TLB invalidate operation is detected, the L2 MMU arrays become inaccessible due to the snooping activity caused by the invalidate.

If the MMU is busy due to a higher priority operation, such as a **tlbivax** or **tlbilx**, instructions cannot be fetched until that operation completes.

If the page translation is in neither TLB, an instruction TLB error interrupt occurs, as described in Section 4.9.15, "Instruction TLB Error Interrupt—IVOR14/GIVOR14."

TLBs are described in detail in Chapter 6, "Memory Management Units (MMUs)."

10.3.1.2 Interrupts Associated with Instruction Fetching

An instruction fetch can generate the following interrupts:

- An instruction TLB error interrupt occurs when the effective address translation for a fetch is not found in the TLBs. This interrupt is described in detail in Section 4.9.15, "Instruction TLB Error Interrupt—IVOR14/GIVOR14."
- An instruction storage interrupt is caused when one of the following occurs during an attempt to fetch instructions:
  - An execute access control exception is caused when one of the following conditions exist:
    - In user mode, an instruction fetch attempts to access a memory location that is not user mode execute enabled (page access control bit UX = 0). This condition is detected solely on the basis of the MSR[PR] bit and occurs regardless of whether the processor is in guest state or not.
    - In supervisor mode, an instruction fetch attempts to access a memory location that is not supervisor mode execute enabled (page access control bit SX = 0). This condition is detected solely on the basis of the MSR[PR] bit and occurs regardless of whether the processor is in guest state or not.

  When an instruction storage interrupt occurs, the processor suppresses execution of the instruction causing the exception. For more information, see Section 4.9.5, "Instruction Storage Interrupt (ISI)—IVOR3/GIVOR3."

10.3.1.3 Cache-Related Latency

The following may happen when instructions are fetched from the instruction cache:

- If the fetch hits the cache or an ILFB (Instruction Line Fill Buffer), it takes 2 clock cycles after the request for as many as four instructions to enter the IQ. The cache is not blocked to internal accesses during a cache reload (hits under misses).

  The cache allows a hit under one miss and is only blocked by a cache line reload for the cycle during the cache write. For example, if a cache miss is discarded by a misprediction and a new fetch hits, the cache allows instructions to come back. As many as four instructions per cycle are fetched from the cache until the original miss comes back and a cache reload is performed, which blocks the cache for 1 cycle.

If the cache is busy due to a higher priority operation, such as an **icbi** or a cache line reload, instructions cannot be fetched until that operation completes.

- If an instruction fetch misses in the instruction cache, it is fetched from the L2 cache.
- If an instruction fetch misses in the instruction cache and the L2 cache, the e500mc initiates a bus transaction to the off-core memory system.

The architecture defines WIM (of WIMGE) bits that define caching characteristics for the corresponding page. Fetching instruction as caching-inhibited (I=1) produce the following actions:

- The ILFB may hit, and the instructions returned from the ILFB will be used, even if the ILFB entry was established by an earlier cacheable access.
- The instruction cache will perform an access and may hit, and if a hit occurs the instructions will be used.
- The L2 cache will not attempt to perform an access if the access is caching-inhibited.
- If the ILFB and instruction cache do not hit, the fetch is performed by performing bus transactions to memory and the fetch will return and use the entire fetch group that was requested. Fetching using caching-inhibited accesses will therefore not produce a bus transaction for each instruction, but instead one bus transaction for each fetch group.

Software should not alias caching and caching-inhibited real addresses without first invalidating the caches and performing an **isync** prior to fetching to those same addresses caching-inhibited.

10.3.2 Dispatch, Issue, and Completion Considerations

The core's ability to dispatch as many as two instructions per cycle depends on the mix of instructions and on the availability of issue queues and CQ entries. As many as two instructions can be dispatched in parallel, but an instruction in IQ1 cannot be dispatched ahead of an instruction in IQ0.

Instructions can issue out-of-order from GIQ0 and GIQ1. GIQ0 can issue to SFX0, CFX, and LSU. GIQ1 can issue to SFX1, CFX, and LSU. If an instruction stalls in GIQ0 (reservation station busy), an instruction in GIQ1 can issue if its reservation station is available.

Instructions can issue in-order from FIQ. FIQ can issue to the FPU at one per cycle.

Issue queues and reservation stations allow the e500mc to dispatch instructions even if execution units are busy. The issue logic reads operands from register files and rename registers and routes instructions to the proper execution unit. Execution begins when all operands are available, the instruction is in the reservation station, and any execution serialization requirements are met.

Instructions pass through a single-entry reservation station associated with each execution unit. If a data dependency keeps an instruction from starting execution, that instruction is held in a reservation station. Execution begins during the same clock cycle that the rename register is updated with the data the instruction is dependent on.

The CQ maintains program order after instructions are dispatched, guaranteeing in-order completion and a precise exception model. Instruction state and other information required for completion are kept in this 14-entry FIFO. All instructions complete in order; none can retire ahead of a previous instruction. In-order

completion ensures the correct architectural state when the e500mc must recover from a mispredicted branch or exception.

Instructions are retired much as they are dispatched: as many as two can be retired simultaneously, but never out of order.

NOTE

- Instructions must be nonspeculative to complete.
- As many as two rename registers can be updated per clock cycle. Because load and store with update instructions require two rename registers they are broken into two instructions at dispatch (**lwzu** is broken into **lwz** and **addi**). These two instructions are assigned two CQ entries and each is assigned CR and GPR renames at dispatch.
- Some instructions have retirement restrictions, such as retiring only out of CQ0. See Section 10.3.2.1, "Instruction Serialization."

Program-related exceptions are signaled when the instruction causing the exception reaches CQ0. Previous instructions are allowed to complete before the exception is taken, which ensures that any exceptions those instructions may cause are taken.

10.3.2.1 Instruction Serialization

Although the e500mc core can dispatch and complete two instructions per cycle, some serializing instructions limit dispatch and completion to one per cycle. There are seven basic types of instruction serialization:

Presync serialization — Presync-serialized instructions are held in the instruction queue until all prior instructions have completed. They are then decoded and execute. For example, instructions such as **mfspr** that read a non-renamed status register are marked as presync-serialized.

Postsync serialization — Postsync-serialized instructions, such as **mtspr**[XER], prevent other instructions from decoding until the serialized instruction completes. For example, instructions that modify processor state in a way that affects the handling of future instruction execution are marked with postsync-serialization. These instructions are identified in the latency tables in Section 10.5, "Instruction Latency Summary."

Move-from serialization — Move-from serialization is a weaker synchronization than presync serialization. A move-from serialized instruction can decode, but stalls in an execution unit's reservation station until all prior instructions have completed. If the instruction is currently in the reservation station and is the oldest instruction, it can begin execution in the next cycle. Note that subsequent instructions can decode and execute while a move-from serialized instruction is pending. Only a few instructions are move-from serialized, so that they do not examine architectural state until all older instructions that could affect the architectural state have completed.

Move-to serialization	A move-to serialized instruction cannot execute until the cycle after it is in CQ0, that is, the cycle after it becomes the oldest instruction. This serialization is weaker than move-from serialization in that the instruction need not spend an extra cycle in the reservation station. Move-to serializing instructions include **tlbre**, **tlbsx**, **tlbwe**, **mtmsr, wrtee**, **wrteei**, and all **mtspr** instructions.
Refetch serialization	Refetch-serialized instructions force refetching of subsequent instructions after completion. Refetch serialization is used when an instruction has changed or may change a particular context needed by subsequent instructions. Examples include **isync**, **sc**, **rfi**, **rfci**, **rfmci**, **rfdi**, **rfgi**, **wait**, **sc**, **ehpriv**, **dnh**, and any instruction that causes the summary-overflow XER(SO) bit to change state.
Store serialization	Applies to stores and some LSU instructions that access the data cache. Store-serialized instructions are dispatched and held in the LSU's finished store queue. They are not committed to memory until all prior instructions have completed. Although a store-serialized instruction waits in the finished store queue, other load/store instructions can be freely executed. Some store-serialized instructions are further restricted to complete only from CQ0. Only one store-serialized instruction can complete per cycle, although nonserialized instructions can complete in the same cycle as a store-serialized instruction. In general, all stores and cache operation instructions are store serialized.
Unit serialization	Unit serialization instructions proceed down the execution pipeline in a normal manner, but blocks the reservation station for the execution unit. This prevents other instructions from issuing to the reservation station while the unit serialized instruction executes. Normally such instructions will modify the architectural state of a renamed register and the serialization ensures that no other instruction will be accessing the renamed register when the unit serialized instruction executes.

10.3.3 Memory Synchronization Timing Considerations

This section describes the behavior of the **sync** and **mbar** instructions as they are implemented by the e500mc.

10.3.3.1 sync Instruction Timing Considerations

The **sync** instruction provides a memory barrier throughout the memory hierarchy, for example, to ensure that a control bit has been written to its destination control register in the system before the next instruction begins execution (such as to clear a pending interrupt). By its nature, it also provides an ordering boundary for pre- and post-**sync** storage transactions.

On the e500mc, **sync** waits for preceding data memory accesses to reach the point of coherency (that is, visible to the entire memory hierarchy), then it is broadcast on the CoreNet interface. A **sync** does not finish execution until all storage transactions caused by prior instructions complete entirely in its caches and externally on the bus (address and data complete on the bus, excluding instruction fetches). No subsequent instructions and associated storage transactions are initiated until such completion.

Execution of **sync** also generates a SYNC command on the CoreNet interface after which the **sync** instruction may be allowed to complete. Subsequent instructions can execute out of order, but they can complete only after **sync** completes.

It is the responsibility of the system to guarantee the intention of the SYNC command on the CoreNet interface—usually by ensuring that any transactions received before the SYNC command from the e500mc complete in its queues or at their destinations before completing the SYNC command on the CoreNet interface.

10.3.3.2 mbar Instruction Timing Considerations

The **mbar** instruction provides an ordering boundary for storage operations. Its architectural intent is to guarantee that storage operations resulting from previous instructions occur before any subsequent storage operations occur, thereby ensuring an order between pre- and post-**mbar** memory operations. It may be used, for example, to ensure that reads and writes to an I/O device or between I/O devices occur in program order or to ensure that memory updates occur before a semaphore is released.

The architecture allows an implementation to support several classes of storage ordering, selected by the MO field of the **mbar** instruction. The e500mc supports two classes for system flexibility.

The e500mc implements two variations of **mbar**, as follows:

- When MO = 0, **mbar** behaves as defined by the Power ISA™, which on the e500mc for all practical purposes produces the same memory barrier as **sync**.
- When MO = 1, **mbar** is a weaker, faster memory barrier; the e500mc executes it as a pipelined or flowing ordering barrier for potentially higher performance. This ordering barrier flows along with pre- and post-**mbar** memory transactions through the memory hierarchy (L1 cache, L2 cache, and CoreNet interface). On the CoreNet interface, this ordering barrier is issued as an EIEIO command. Note that **mbar** MO = 1 only orders a certain subset of memory transactions depending on the type of transaction and the WIMGE settings (see Section 5.5.5.3, “Memory Access Ordering“). **mbar** MO = 1 ensures that all data accesses (for the ordered subsets) caused by previous instructions complete before any caused by subsequent instructions. This order is seen by all mechanisms. However, unlike **sync** and **mbar** with MO = 0, subsequent instructions can complete without waiting for **mbar** to perform its CoreNet transaction. This provides a faster way to order data accesses.

10.4 Execution

The following sections describe instruction execution behavior within each of the respective execution units in the e500mc.

10.4.1 Branch Unit Execution

When branch or trap instructions change program flow, the IQ must be reloaded with the target instruction stream. Previously issued instructions continue executing while the new instruction stream makes its way into the IQ. Depending on whether target instructions are cached, opportunities may be missed to execute instructions.

The e500mc minimizes penalties associated with flow control operations by features such as the branch target buffer (BTB), dynamic branch prediction, speculative link and counter registers, and nonblocking caches.

10.4.1.1 Branch Instructions and Completion

Branch instructions are not folded on the e500mc; all branch instructions receive a CQ entry (and CRF and GPR renames) at dispatch and must write back in program order.

Branch instructions are dispatched to the BIQ and are assigned a CQ slot, as shown in Figure 10-2.

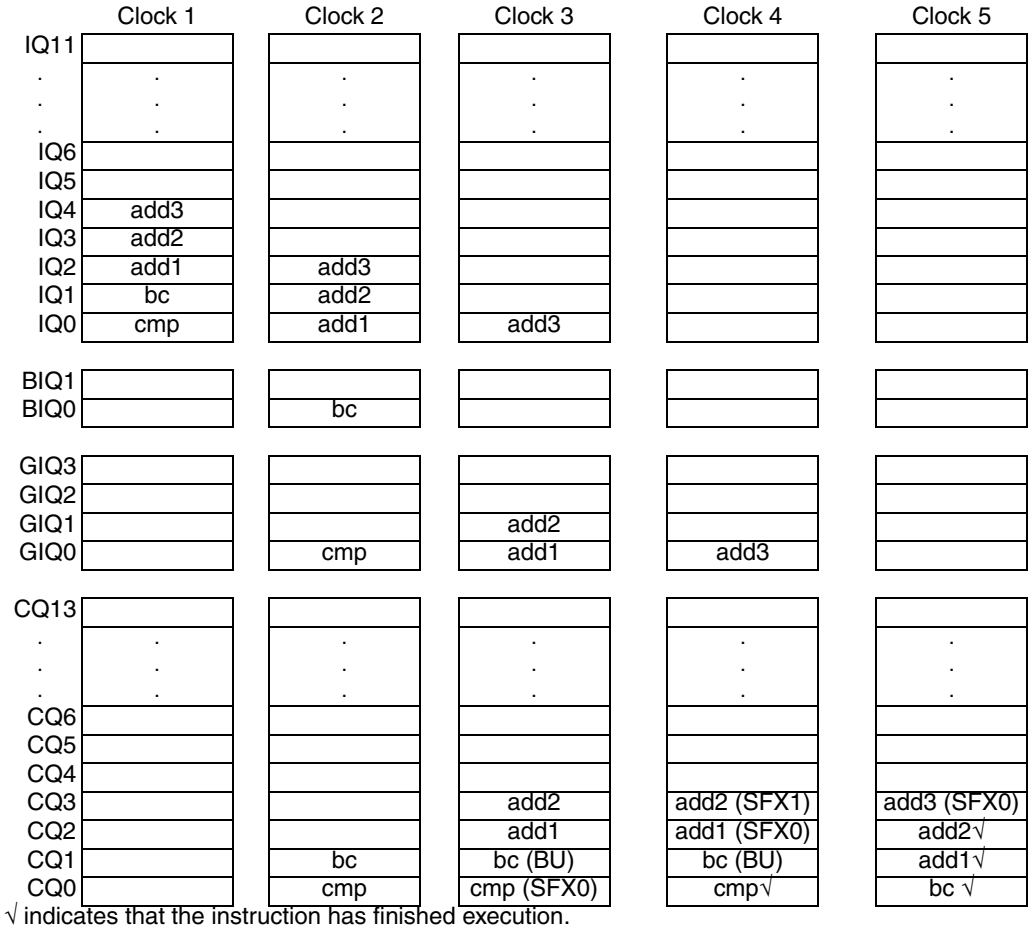


Figure 10-2. Branch Completion (LR/CTR Write-Back)

In this example, the **bc** depends on **cmp** and is predicted as not taken. At the end of clock cycle 1, **cmp** and **bc** are dispatched to the GIQ and BIQ, respectively, and are issued to SFX0 and the BU at the end of clock 2.

In clock cycle 3, the **cmp** executes in SFX0 but the **bc** cannot resolve and complete until the **cmp** results are available; add1 and add2 are dispatched to the GIQ.

In cycle 4, the **bc** resolves as correctly predicted; add1 and add2 are issued to the SUs and are marked as nonspeculative, and add3 is dispatched to the GIQ. The **cmp** is retired from the CQ at the end of cycle 4.

In cycle 5, **bc**, add1, and add2 finish execution, and **bc** and add1 retire.

10.4.1.2 BTB Branch Prediction and Resolution

The e500mc dynamic branch prediction mechanism monitors and records branch instruction behavior, from which the next occurrence of the branch instruction is predicted.

The e500mc does not support static branch prediction—the BO static prediction in branch instructions is ignored.

The valid bit in each BTB entry is zero (invalid) at reset. When a branch instruction first enters the instruction pipeline, it is not allocated in the BTB and so by default is predicted as not taken. If the branch is not taken, nothing is allocated in the BTB. If it is taken, the misprediction allocates a BTB entry for this branch with an initial prediction of strongly taken, as is shown in the example in Figure 10-3.

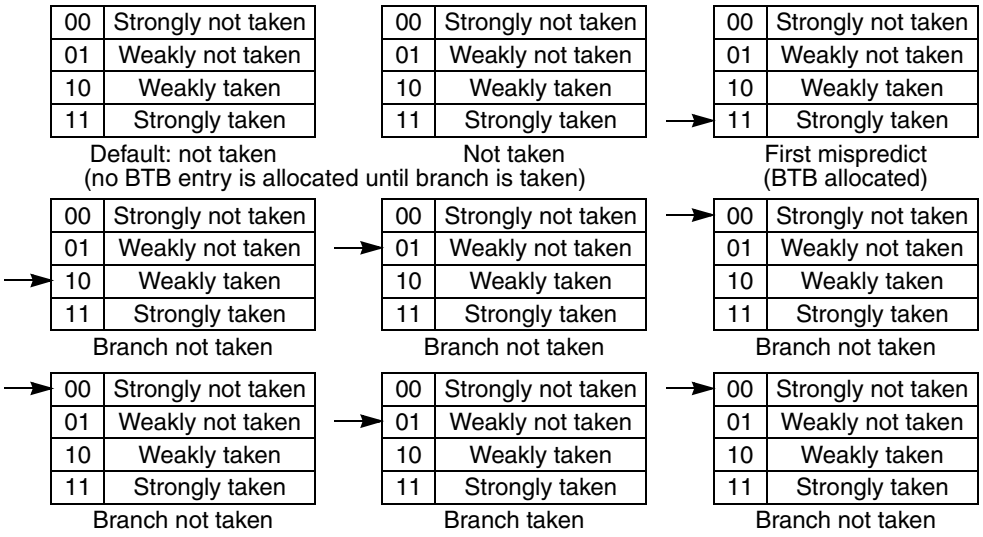


Figure 10-3. Updating Branch History

NOTE

Unconditional branches are allocated in the BTB the first time they are encountered. This example shows how the prediction is updated depending on whether a branch is taken.

The BPU detects whether a fetch group includes any branches that hit in the BTB, and if so, determines the fetching path based on the prediction and the target address.

If the prediction is wrong, subsequent instructions and their results are purged. Instructions ahead of the predicted branch proceed normally, instruction fetching resumes along the correct path, and the history bits are revised.

The number of speculative branches that have not yet been allocated (and are predicted as not taken) is limited only by the space available in the pipeline (the branch execute unit, the BIQ, and the IQ). The presence of speculative branches allocated in the BTB slightly reduces speculation depth.

Instructions after an unresolved branch can execute speculatively, but in-order completion ensures that mispredicted speculative instructions do not complete. When misprediction occurs, the e500mc easily redirects fetching and repairs its machine state because the architectural state is not updated. Any instructions dispatched after a mispredicted branch instruction are flushed from the CQ, and any results are flushed from the rename registers.

10.4.1.2.1 BTB Operations Controlled by BUCSR

This following BTB operations are controlled through BUCSR:

- BTB disabling. BUCSR[BPEN] is used to enable or disable the BTB. The BTB is enabled when the bit is set and disabled when it is cleared.When it is disabled, BTB contents are not used to predict the branch targets and the BTB is not updated as a result of executing branch.
- BTB invalidation. Flash invalidation of the BTB is accomplished by writing BUCSR[BBFI] with a 0 and then a 1 using **mtspr** instructions.

10.4.1.2.2 BTB Special Cases—Phantom Branches and Multiple Matches

The following describes special cases:

- Phantom branches. BTB entries hold effective addresses associated with a branch instruction. A process context switch might bring in another task whose MMU translations are such that it uses the same effective address for another nonbranch instruction for which the BTB has an entry for a previously encountered branch. This causes the fetch unit to redirect instruction fetch to the BTB's target address. Later, during execution of the instruction, the hardware realizes the error and evicts the BTB entry.
- Multiple matches. By ensuring that an entry is unique when it is allocated, the e500mc hardware prevents multiple matches for the same fetch address.

10.4.2 Complex and Simple Unit Execution

The e500mc has one complex unit (CFX) and two simple units (SFX0, SFX1). SFX0 and SFX1 execute all logical and integer computational instructions except multiplies and divides. SFX0 also executes move to and from special registers, performance monitor registers, and **tlbre**, **tlbwe**, and **tlbsx**. The CFX executes multiplies and divides.

10.4.2.1 CFX Divide Execution

Divide latency depends upon the operand data and ranges from 4 to 35 cycles, as shown in Table 10-1.

Table 10-1. The Effect of Operands on Divide Latency

Instruction	Condition	Latency
divw*x*, **divwu***x*	**r**A or **r**B is 0, or **r**A < **r**B	4
	rA is representable in 8 bits	11
	rA representable in 16 bits	19
	rA representable in 32 bits	35

10.4.2.2 CFX Multiply Execution

Table 10-2 shows the latency and repeat rate for multiply instructions.

Table 10-2. The Effect of Operands on Multiply Latency

Instruction	Condition	Latency
mullw*x*, **mulhw***x*, **mulli**	Multiply latency is not operand dependent.	4 cycles, repeat rate of 1

10.4.2.3 CFX Bypass Path

The CFX provides a bypass path for divides so the iterative portion of divide execution is performed outside of the CFX pipeline, allowing subsequent instructions (except other divides) to execute in the main CFX pipeline. In general the bypass path for the divide executes simultaneously with the execution of other instructions in the CFX (such as multiply instructions). However, both the normal path and the bypass path cannot produce a result on the same cycle. therefore, if both a multiply and a divide are scheduled to produce a result on the same cycle, a bubble is created in the CFX pipeline, effectively stalling the CFX pipeline (multiply instructions) from finishing execution in order to create a slot for the divide finish execution and write its result on the result bus. the result of the divide instruction is stalled until there is a slot available in the CFX pipeline.

A new divide instruction cannot start execution if another divide is executing.

10.4.3 Load/Store Execution

The LSU executes instructions that move data between the GPRs and the memory unit of the core (made up of the L1 cache, the L2 cache, and the bus interface unit buffers).

The execution of most load instructions is pipelined in the three LSU stages, during which the effective address is calculated, MMU translations are performed, the data cache array and tags are read, and cache way selection and data alignment are performed. Cacheable loads, when free of data dependencies, execute in a speculative manner with a maximum throughput of one instruction per cycle and 3-cycle latency (floating-point loads take 4 cycles of latency). Data returned from the cache is held in a rename register until the completion logic commits the value to the processor state.

Stores cannot be executed speculatively and must be held in the store queue until completion logic signals that the store instruction is to be committed, at which point the data cache array is updated.

If operands are misaligned, additional latency may be incurred either for an alignment exception or for additional cache or bus accesses. Table 10-4 gives load and store instruction execution latencies.

10.4.3.1 Effect of Operand Placement on Performance

The location and alignment of operands in memory may affect performance of memory accesses, in some cases significantly, as shown in Table 10-3.

Alignment of memory operands on natural boundaries guarantees the best performance. For the best performance across the widest range of implementations, the programmer should assume the performance

model described in the *EREF: A Programmer's Reference Manual for Freescale Power Architecture® Processors.*

The effect of alignment on memory operation performance is the same for big- and little-endian addressing modes, including load-multiple and store-multiple operations.

In Table 10-3, optimal means that one effective address (EA) calculation occurs during the memory operation. Good means that multiple EA calculations occur during the operation, which may cause additional cache or bus activities with multiple transfers. Poor means that an alignment interrupt is generated by the memory operation.

Table 10-3. Performance Effects of Operand Placement in Memory

Operand		Boundary Crossing[1]		
Size	**Byte Alignment**	**None**	**Cache Line**	**Protection Boundary**
8 byte	8 <4	optimal good	— good	— good
4 byte	4 <4	optimal good	— good	— good
2 byte	2 <2	optimal good	— good	— good
1 byte	1	optimal	—	—
lmw, **stmw**	4 <4	good poor	good poor	good poor

1 Optimal: One EA calculation occurs.
Good: Multiple EA calculations occur which may cause additional bus activities with multiple bus transfers.
Poor: Alignment Interrupt occurs

10.5 Instruction Latency Summary

Instruction latencies are shown in Table 10-4. The execution unit responsible for executing the instruction (where it is dispatched) is listed. Instructions that are dispatched to SFX0, SFX1 mean that those instructions can go to either execution unit. COMP means the instruction is not dispatched to a unit, and its execution is directly handled by the completion unit.

All latencies assume fairly normal conditions. In general these are also the best case conditions. Some instructions may incur additional stalls based on core and SoC conditions. For example, load and store instructions may miss in the L1 cache, or attempt to load Guarded Cache Inhibited memory, which may incur significant delay since the operation requires the core to retrieve the data from other parts of the system connected to the CoreNet interface. Incoming snoops received by the core can also make the cache or even the TLB unavailable for instruction use during any given cycle. Such interactions are not described here and are beyond the scope of this manual.

Information contained in Table 10-4 does not address all effects of the core pipeline, but is intended as a guide for instruction scheduling.

- The latency is execution latency from the point of when the instruction begins execution in an execution unit until the execution unit has produced the intended result (that is, when it finishes execution).
- Other results of the instruction, such as flags (like XER[OV] or the CR result of a "." instruction) may take l extra cycle after execution is finished to be available as inputs to other instructions.
- Other cycles taken for things such as instruction fetch, decode, dispatch, and completion are not represented in this table.
- The repeat rate specifies how many cycles it takes before another instruction dispatched to the unit can begin execution. For example, an instruction with a latency of 3 and a repeat rate of 1 means that even though it takes 3 cycles to produce the result, several of these instructions back to back can produce a result every cycle. This indicates how the particular execution unit is pipelined.
- The type of serialization performed on instructions is described in Section 10.3.2.1, "Instruction Serialization".

Table 10-4. e500mc Instruction Latencies

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
add	SFX0, SFX1	—	1	1	—
add.	SFX0, SFX1	—	1	1	—
addc	SFX0, SFX1	—	1	1	—
addc.	SFX0, SFX1	—	1	1	—
addco	SFX0, SFX1	—	1	1	—
addco.	SFX0, SFX1	—	1	1	—
adde	SFX0, SFX1	—	1	1	—
adde.	SFX0, SFX1	—	1	1	—
addeo	SFX0, SFX1	—	1	1	—
addeo.	SFX0, SFX1	—	1	1	—
addi	SFX0, SFX1	—	1	1	—
addic	SFX0, SFX1	—	1	1	—
addic.	SFX0, SFX1	—	1	1	—
addis	SFX0, SFX1	—	1	1	—
addme	SFX0, SFX1	—	1	1	—
addme.	SFX0, SFX1	—	1	1	—
addmeo	SFX0, SFX1	—	1	1	—
addmeo.	SFX0, SFX1	—	1	1	—
addo	SFX0, SFX1	—	1	1	—

Table 10-4. e500mc Instruction Latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
addo.	SFX0, SFX1	—	1	1	—
addze	SFX0, SFX1	—	1	1	—
addze.	SFX0, SFX1	—	1	1	—
addzeo	SFX0, SFX1	—	1	1	—
addzeo.	SFX0, SFX1	—	1	1	—
and	SFX0, SFX1	—	1	1	—
and.	SFX0, SFX1	—	1	1	—
andc	SFX0, SFX1	—	1	1	—
andc.	SFX0, SFX1	—	1	1	—
andi.	SFX0, SFX1	—	1	1	—
andis.	SFX0, SFX1	—	1	1	—
b	BU	—	1	1	—
ba	BU	—	1	1	—
bc	BU	—	1	1	—
bca	BU	—	1	1	—
bcctr	BU	—	1	1	—
bcctrl	BU	—	1	1	—
bcl	BU	—	1	1	—
bcla	BU	—	1	1	—
bclr	BU	—	1	1	—
bclrl	BU	—	1	1	—
bl	BU	—	1	1	—
bla	BU	—	1	1	—
cmp	SFX0, SFX1	—	1	1 or 2	EQ bit is 1 cycle to branch unit, other results are 2 cycles
cmpi	SFX0, SFX1	—	1	1 or 2	EQ bit is 1 cycle to branch unit, other results are 2 cycles
cmpl	SFX0, SFX1	—	1	1 or 2	EQ bit is 1 cycle, other results are 2 cycles
cmpli	SFX0, SFX1	—	1	1 or 2	EQ bit is 1 cycle, other results are 2 cycles
cntlzw	SFX0	—	1	1	—
cntlzw.	SFX0	—	1	1	—
crand	BU	—	1	1	—
crandc	BU	—	1	1	—

Table 10-4. e500mc Instruction Latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
creqv	BU	—	1	1	—
crnand	BU	—	1	1	—
crnor	BU	—	1	1	—
cror	BU	—	1	1	—
crorc	BU	—	1	1	—
crxor	BU	—	1	1	—
dcba	LSU	Store	1	3	—
dcbal	LSU	Store	1	3	—
dcbf	LSU	Store	1	3	—
dcbfep	LSU	Store	1	3	—
dcbi	LSU	Store	1	3	—
dcblc	LSU	Store	1	3	—
dcbst	LSU	Store	1	3	—
dcbstep	LSU	Store	1	3	—
dcbt	LSU	—	1	3	—
dcbtep	LSU	—	1	3	—
dcbtls	LSU	Store	1	3	—
dcbtst	LSU	—	1	3	—
dcbtstep	LSU	—	1	3	—
dcbtstls	LSU	Store	1	3	—
dcbz	LSU	Store	1	3	—
dcbzep	LSU	Store	1	3	—
dcbzl	LSU	Store	1	3	—
dcbzlep	LSU	Store	1	3	—
divw	CFX	—	4 to 35	4 to 35	See Section 10.4.2.1, “CFX Divide Execution”
divw.	CFX	—	4 to 35	4 to 35	See Section 10.4.2.1, “CFX Divide Execution”
divwo	CFX	—	4 to 35	4 to 35	See Section 10.4.2.1, “CFX Divide Execution”
divwo.	CFX	—	4 to 35	4 to 35	See Section 10.4.2.1, “CFX Divide Execution”
divwu	CFX	—	4 to 35	4 to 35	See Section 10.4.2.1, “CFX Divide Execution”
divwu.	CFX	—	4 to 35	4 to 35	See Section 10.4.2.1, “CFX Divide Execution”
divwuo	CFX	—	4 to 35	4 to 35	See Section 10.4.2.1, “CFX Divide Execution”
divwuo.	CFX	—	4 to 35	4 to 35	See Section 10.4.2.1, “CFX Divide Execution”

Table 10-4. e500mc Instruction Latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
dnh	COMP	Refetch	—	—	The **dnh** instruction executes during completion and is not dispatched to an execution unit.
dsn	LSU	Store	1	3	—
ehpriv	COMP	Refetch	—	—	The **ehpriv** instruction executes during completion and is not dispatched to an execution unit.
eqv	SFX0, SFX1	—	1	1	—
eqv.	SFX0, SFX1	—	1	1	—
extsb	SFX0, SFX1	—	1	1	—
extsb.	SFX0, SFX1	—	1	1	—
extsh	SFX0, SFX1	—	1	1	—
extsh.	SFX0, SFX1	—	1	1	—
fabs	FPU	—	2	8	—
fabs.	FPU	—	2	8	—
fadd	FPU	—	4	10	—
fadd.	FPU	—	4	10	—
fadds	FPU	—	2	8	—
fadds.	FPU	—	2	8	—
fcmpo	FPU	—	2	8	—
fcmpu	FPU	—	2	8	—
fctiw	FPU	—	2	8	—
fctiw.	FPU	—	2	8	—
fctiwz	FPU	—	2	8	—
fctiwz.	FPU	—	2	8	—
fdiv	FPU	—	68	68	
fdiv.	FPU	—	68	68	
fdivs	FPU	—	38	38	
fdivs.	FPU	—	38	38	
fmadd	FPU	—	4	10	—
fmadd.	FPU	—	4	10	—
fmadds	FPU	—	2	8	—
fmadds.	FPU	—	2	8	—
fmr	FPU	—	2	8	—

Table 10-4. e500mc Instruction Latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
fmr.	FPU	—	2	8	—
fmsub	FPU	—	4	10	—
fmsub.	FPU	—	4	10	—
fmsubs	FPU	—	2	8	—
fmsubs.	FPU	—	2	8	—
fmul	FPU	—	4	10	—
fmul.	FPU	—	4	10	—
fmuls	FPU	—	2	8	—
fmuls.	FPU	—	2	8	—
fnabs	FPU	—	2	8	—
fnabs.	FPU	—	2	8	—
fneg	FPU	—	2	8	—
fneg.	FPU	—	2	8	—
fnmadd	FPU	—	4	10	—
fnmadd.	FPU	—	4	10	—
fnmadds	FPU	—	2	8	—
fnmadds.	FPU	—	2	8	—
fnmsub	FPU	—	4	10	—
fnmsub.	FPU	—	4	10	—
fnmsubs	FPU	—	2	8	—
fnmsubs.	FPU	—	2	8	—
fres	FPU	—	38	38	—
fres.	FPU	—	38	38	—
frsp	FPU	—	2	8	—
frsp.	FPU	—	2	8	—
frsqrte	FPU	—	2	8	—
frsqrte.	FPU	—	2	8	—
fsel	FPU	—	2	8	—
fsel.	FPU	—	2	8	—
fsub	FPU	—	4	10	—
fsub.	FPU	—	4	10	—
fsubs	FPU	—	2	8	—

Table 10-4. e500mc Instruction Latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
fsubs.	FPU	—	2	8	—
icbi	LSU	Store	1	3	—
icbiep	LSU	Store	1	3	—
icblc	LSU	Store	1	3	—
icbt	LSU	—	1	3	Note **icbt** with CT=0, is treated as a NOP.
icbtls	LSU	Presync, postsync	1	3	—
isel	SFX0, SFX1	—	1	1	—
isync	COMP	Refetch	1	1	—
lbdx	LSU	—	—	—	Decorated loads are normally performed as Guarded and CI so latency/repeat rate is system driven.
lbepx	LSU	—	1	3	—
lbz	LSU	—	1	3	—
lbzu	LSU	—	1	3	—
lbzux	LSU	—	1	3	—
lbzx	LSU	—	1	3	—
lfd	LSU	—	1	4	—
lfddx	LSU	—	—	—	Decorated loads are normally performed as Guarded and CI so latency/repeat rate is system driven.
lfdepx	LSU	—	1	4	—
lfdu	LSU	—	1	4	—
lfdux	LSU	—	1	4	—
lfdx	LSU	—	1	4	—
lfs	LSU	—	1	4	—
lfsu	LSU	—	1	4	—
lfsux	LSU	—	1	4	—
lfsx	LSU	—	1	4	—
lha	LSU	—	1	3	—
lhau	LSU	—	1	3	—
lhaux	LSU	—	1	3	—
lhax	LSU	—	1	3	—
lhbrx	LSU	—	1	3	—

Table 10-4. e500mc Instruction Latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
lhdx	LSU	—	—	—	Decorated loads are normally performed as Guarded and CI so latency/repeat rate is system driven.
lhepx	LSU	—	1	3	—
lhz	LSU	—	1	3	—
lhzu	LSU	—	1	3	—
lhzux	LSU	—	1	3	—
lhzx	LSU	—	1	3	—
lmw	LSU	—	*r* + 3	*r* + 3	*r* indicates the number of register loaded. **lmw** will actually stall in decode while completion queue entries are allocated for it each cycle.
lwarx	LSU	Presync	3	3	—
lwbrx	LSU	—	1	3	—
lwdx	LSU	—	—	—	Decorated loads are normally performed as Guarded and CI so latency/repeat rate is system driven.
lwepx	LSU	—	1	3	—
lwz	LSU	—	1	3	—
lwzu	LSU	—	1	3	—
lwzux	LSU	—	1	3	—
lwzx	LSU	—	1	3	—
mbar	LSU	Store	1	3	In general, **mbar** will take several more cycles to perform the ordering
mcrf	BU	—	1	1	—
mcrfs	FPU	—	2	8	—
mcrxr	BU	Presync, postsync	1	1	—
mfcr	SFX0	Move-from	5	5	—
mffs	FPU	Move-from	8	8	—
mffs.	FPU	Move-from	8	8	—
mfmsr	SFX0	—	4	4	—
mfocrf	SFX0	Move-from	5	5	—
mfpmr	SFX0	—	4	4	—
mfspr (CTR)	SFX0, SFX1	—	1	1	**mfctr** stalls in decode until any other **mtctr** instruction finishes execution.
mfspr (DBSR)	SFX0	Presync, postsync	4	4	—

Table 10-4. e500mc Instruction Latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
mfspr (LR)	SFX0, SFX1	—	1	1	**mflr** stalls in decode until any other **mtlr** instruction finishes execution.
mfspr (other)	SFX0	—	4	4	—
mfspr (XER)	SFX0	Move-from	5	5	—
mftb	SFX0	—	4	4	—
msgclr	SFX0	Move-to	1	1	—
msgsnd	LSU	Store	1	3	—
mtcrf	SFX0	Presync, postsync, move-to	4	2	If only single field is moved, latency and repeat rate is same as **mtocrf** and there is no serialization.
mtfsb0	FPU	Move-to	8	8	—
mtfsb0.	FPU	Move-to	8	8	—
mtfsb1	FPU	Move-to	8	8	—
mtfsb1.	FPU	Move-to	8	8	—
mtfsf	FPU	Move-to	8	8	—
mtfsf.	FPU	Move-to	8	8	—
mtfsfi	FPU	Move-to	8	8	—
mtfsfi.	FPU	Move-to	8	8	—
mtmsr	SFX0, SFX1	Presync, postsync, move-to	4	2	—
mtocrf	SFX0, SFX1	—	1	1	—
mtpmr	SFX0	Move-to	1	1	—
mtspr (CTR)	SFX0, SFX1	Move-to	1	1	**mtctr** stalls in decode until any other **mtctr** instruction finishes execution.
mtspr (DBCR0, DBSR, or DBSRWR)	SFX0	Presync, postsync, move-to	4	2	—
mtspr (LR)	SFX0, SFX1	Move-to	1	1	**mtlr** stalls in decode until any other **mtlr** instruction finishes execution.
mtspr (NPIDR)	SFX0	Postsync, move-to	4	2	—
mtspr (other)	SFX0	Move-to	1	1	—
mtspr (PID)	SFX0	Presync, postsync, move-to	4	2	

Table 10-4. e500mc Instruction Latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
mtspr (XER)	SFX0	Postsync, move-to	4	2	—
mulhw	CFX	—	1	4	See Section 10.4.2.2, “CFX Multiply Execution
mulhw.	CFX	—	1	4	See Section 10.4.2.2, “CFX Multiply Execution
mulhwu	CFX	—	1	4	See Section 10.4.2.2, “CFX Multiply Execution
mulhwu.	CFX	—	1	4	See Section 10.4.2.2, “CFX Multiply Execution
mulli	CFX	—	1	4	See Section 10.4.2.2, “CFX Multiply Execution
mullw	CFX	—	1	4	See Section 10.4.2.2, “CFX Multiply Execution
mullw.	CFX	—	1	4	See Section 10.4.2.2, “CFX Multiply Execution
mullwo	CFX	—	1	4	See Section 10.4.2.2, “CFX Multiply Execution
mullwo.	CFX	—	1	4	See Section 10.4.2.2, “CFX Multiply Execution
nand	SFX0, SFX1	—	1	1	—
nand.	SFX0, SFX1	—	1	1	—
neg	SFX0, SFX1	—	1	1	—
neg.	SFX0, SFX1	—	1	1	—
nego	SFX0, SFX1	—	1	1	—
nego.	SFX0, SFX1	—	1	1	—
nor	SFX0, SFX1	—	1	1	—
nor.	SFX0, SFX1	—	1	1	—
or	SFX0, SFX1	—	1	1	—
or.	SFX0, SFX1	—	1	1	—
orc	SFX0, SFX1	—	1	1	—
orc.	SFX0, SFX1	—	1	1	—
ori	SFX0, SFX1	—	1	1	—
oris	SFX0, SFX1	—	1	1	—
rfci	COMP	Refetch	—	—	Return from interrupt instructions execute during completion and are not dispatched to an execution unit.
rfdi	COMP	Refetch	—	—	Return from interrupt instructions execute during completion and are not dispatched to an execution unit.
rfgi	COMP	Refetch	—	—	Return from interrupt instructions execute during completion and are not dispatched to an execution unit.

Table 10-4. e500mc Instruction Latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
rfi	COMP	Refetch	—	—	Return from interrupt instructions execute during completion and are not dispatched to an execution unit.
rfmci	COMP	Refetch	—	—	Return from interrupt instructions execute during completion and are not dispatched to an execution unit.
rlwimi	SFX0, SFX1	—	1	1	—
rlwimi.	SFX0, SFX1	—	1	1	—
rlwinm	SFX0, SFX1	—	1	1	—
rlwinm.	SFX0, SFX1	—	1	1	—
rlwnm	SFX0, SFX1	—	1	1	—
rlwnm.	SFX0, SFX1	—	1	1	—
sc	COMP	Refetch	—	—	The **sc** instruction executes during completion and is not dispatched to an execution unit.
slw	SFX0, SFX1	—	1	1	—
slw.	SFX0, SFX1	—	1	1	—
sraw	SFX0, SFX1	—	1	1	—
sraw.	SFX0, SFX1	—	1	1	—
srawi	SFX0, SFX1	—	1	1	—
srawi.	SFX0, SFX1	—	1	1	—
srw	SFX0, SFX1	—	1	1	—
srw.	SFX0, SFX1	—	1	1	—
stb	LSU	Store	1	3	—
stbdx	LSU	Store	1	3	—
stbepx	LSU	Store	1	3	—
stbu	LSU	Store	1	3	—
stbux	LSU	Store	1	3	—
stbx	LSU	Store	1	3	—
stfd	LSU	Store	1	3	—
stfddx	LSU	Store	1	3	—
stfdepx	LSU	Store	1	3	—
stfdu	LSU	Store	1	3	—
stfdux	LSU	Store	1	3	—
stfdx	LSU	Store	1	3	—

Table 10-4. e500mc Instruction Latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
stfiwx	LSU	Store	1	3	—
stfs	LSU	Store	1	3	—
stfsu	LSU	Store	1	3	—
stfsux	LSU	Store	1	3	—
stfsx	LSU	Store	1	3	—
sth	LSU	Store	1	3	—
sthbrx	LSU	Store	1	3	—
sthdx	LSU	Store	1	3	—
sthepx	LSU	Store	1	3	—
sthu	LSU	Store	1	3	—
sthux	LSU	Store	1	3	—
sthx	LSU	Store	1	3	—
stmw	LSU	Store	*r* + 1	*r* + 3	*r* indicates the number of register stored. **stmw** will actually stall in decode while completion queue entries are allocated for it each cycle.
stw	LSU	Store	1	3	—
stwbrx	LSU	Store	1	3	—
stwcx.	LSU	Presync, postsync, store	1	3	—
stwdx	LSU	Store	1	3	—
stwepx	LSU	Store	1	3	—
stwu	LSU	Store	1	3	—
stwux	LSU	Store	1	3	—
stwx	LSU	Store	1	3	—
subf	SFX0, SFX1	—	1	1	—
subf.	SFX0, SFX1	—	1	1	—
subfc	SFX0, SFX1	—	1	1	—
subfc.	SFX0, SFX1	—	1	1	—
subfco	SFX0, SFX1	—	1	1	—
subfco.	SFX0, SFX1	—	1	1	—
subfe	SFX0, SFX1	—	1	1	—
subfe.	SFX0, SFX1	—	1	1	—
subfeo	SFX0, SFX1	—	1	1	—

Table 10-4. e500mc Instruction Latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
subfeo.	SFX0, SFX1	—	1	1	—
subfic	SFX0, SFX1	—	1	1	—
subfme	SFX0, SFX1	—	1	1	—
subfme.	SFX0, SFX1	—	1	1	—
subfmeo	SFX0, SFX1	—	1	1	—
subfmeo.	SFX0, SFX1	—	1	1	—
subfo	SFX0, SFX1	—	1	1	—
subfo.	SFX0, SFX1	—	1	1	—
subfze	SFX0, SFX1	—	1	1	—
subfze.	SFX0, SFX1	—	1	1	—
subfzeo	SFX0, SFX1	—	1	1	—
subfzeo.	SFX0, SFX1	—	1	1	—
sync (msync)	LSU	Postsync, store	1	3	In general, **sync** will take several more cycles to perform the ordering
tlbilx	LSU	—	1 or 128	3 or 131	When T=0 or T=1, **tlbilx** requires 131 cycles latency and 128 cycles of repeat rate
tlbivax	LSU	—	1	3	—
tlbre	SFX0	Presync, postsync, move-to	4	2	—
tlbsx	SFX0	Presync, postsync, move-to	4	2	—
tlbsync	LSU	Store	1	3	—
tlbwe	SFX0	Presync, postsync, move-to	4	2	—
tw	SFX0	—	2	2	—
twi	SFX0	—	2	2	—
wait	COMP	Refetch	—	—	The **wait** instruction executes during completion and is not dispatched to an execution unit.
wrtee	SFX0	Move-to, postsync	4	2	—
wrteei	SFX0	Move-to, postsync	4	2	—
xor	SFX0, SFX1	—	1	1	—
xor.	SFX0, SFX1	—	1	1	—
xori	SFX0, SFX1	—	1	1	—
xoris	SFX0, SFX1	—	1	1	—

10.6 Instruction Scheduling Guidelines

This section provides an overview of instruction scheduling guidelines, followed by detailed examples showing how to optimize scheduling with respect to various pipeline stages. Performance can be improved by avoiding resource conflicts and scheduling instructions to take fullest advantage of the parallel execution units.

Instruction scheduling can be improved by observing the following guidelines:

- To reduce branch mispredictions, separate the instruction that sets CR bits from the branch instruction that evaluates them. Because there can be no more than 26 instructions in the processor (with the instruction that sets CR in CQ0 and the dependent branch instruction in IQ11), there is no advantage to having more than 24 instructions between them.
- When branching to a location specified by the CTR or LR, separate the **mtspr** instruction that initializes the CTR or LR from the dependent branch instruction. This ensures the register values are immediately available to the branch instruction.
- Schedule instructions so two can be dispatched at a time.
- Schedule instructions to minimize stalls due to busy execution units.
- Avoid scheduling high-latency instructions close together. Interspersing single-cycle latency instructions between longer-latency instructions minimizes the effect that instructions such as integer divide can have on throughput.
- Avoid using serializing instructions.
- Schedule instructions to avoid dispatch stalls. As many as 14 instructions can be assigned CR and GPR renames and can be assigned CQ entries; therefore, 14 instructions can be in the execute stages at any one time. (However, note the exception of load or store with update instructions, which are broken into two instructions at dispatch.)
- Avoid branches where possible; favor not-taken branches over taken branches.

Chapter 11
Core Software Initialization Requirements

This chapter describes the steps software should perform at boot time (that is, after power-on reset has occurred) to properly initialize the e500mc core.

11.1 Core State and Suggested Software Initialization After Reset

The state of each area of the core is presented with respect to how it is initially set after a reset has occurred, and what actions software should perform to properly initialize the core. Note that, in general, the boot loader will perform most of these actions, and operating systems or hypervisors will generally start execution with this state appropriately initialized.

There are other requirements for software to initialize areas outside of the core which are not addressed here. See the integrated device reference manual for more information.

11.2 MMU State

At reset the MMU has all valid bits (TLB[V]) set to 0 except for the initial boot page which is described in Section 6.6, "TLB States after Reset." In addition, all L1MMU entries are invalidated. No other information in the invalid TLB entries are initialized. If later software depends on certain values in TLB entries to be set to known values, software must do that by writing all the TLB entries individually and setting their fields to the known values.

Since the initial TLB entry for the boot page does not have the Guarded bit set, software may want to rewrite that TLB entry to set the Guarded bit if at the time it does not want speculative accesses to occur.

11.3 Register State

11.3.1 GPRs

After reset GPRs may contain random values that may differ from core to core or may differ from reset to reset. Practically, a GPR should not be used as a source input until it has been previously set to a value by software. However, to aid in debugging boot software, the GPRs should be set to known values at the start of reset. This can be accomplished by performing an **xor** instruction for each register using the same register as the **r**A, **r**S, and **r**B operands:

```
xor      r0,r0,r0      // set r0 to 0
xor      r1,r1,r1      // set r1 to 0
...                     // do for all 32 GPRs
```

11.3.2 FPRs

At reset FPRs may contain random values that may differ from core to core or may differ from reset to reset. Practically, an FPR should not be used as a source input until it has been previously set to a value by software. However, FPRs contain hidden tag bits that describe the type of information that the FPR holds, and using an FPR that has never been properly initialized may give unpredictable results. therefore the FPRs should be set to known values at the start of reset. This can be accomplished by loading the FPRs with a known value from memory. Note that this operation may not be able to be performed until later in the boot process when software has properly initialized memory, or even possibly at the start of the operating system or hypervisor. The following code sequence can be used assuming that r3 points to a doubleword aligned scratch memory location:

```
mfmsr    r5             // get current MSR
xor      r4,r4,r4       // set r4 to 0
ori      r4,r5,0x2000   // set MSR[FP]
mtmsr    r4
isync
xor      r4,r4,r4       // set to 0
stw      r4,0(r3)       // clear first word of memory location
stw      r4,4(r3)       // clear second word of memory location
lfd      fr0,0(r3)      // set fr0 to 0
fmr      fr1,fr0        // set fr1 to 0
fmr      fr2,fr0        // set fr2 to 0
...                     // set rest of FPRs using fmr from r0
mtmsr    r5             // restore MSR (turn off FP if desired)
isync
```

11.3.3 SPRs

At reset SPRs are generally set to 0, except for certain SPRs that contain either configuration values or that reflect special state out of reset. SPRs that have initial values other than 0 out of reset are shown in Table 11-1.

Table 11-1. SPRs with Non-Zero Reset Values

SPR	Description of Non-Zero Reset Values
CDCSR0	Set to configuration information denoting presence of Floating Point capability.
DBSR	DBSR[MRR] is set to reflect the most recent reset, which after a hard reset will be 0b10.
L1CFG0	Set to configuration information describing the L1 cache capabilities and organization.
L1CFG1	Set to configuration information describing the L1 cache capabilities and organization.
L2CFG0	Set to configuration information describing the L2 cache capabilities and organization.
MMUCFG	Set to configuration information describing the MMU capabilities and organization.
PIR	Set to a unique identifier of the core distinct from other cores in the system. This value is set from signal inputs from the integrated device. The initial value reflects the core's location in the device's topology and all cores in an integrated device contain unique values for that device.
PVR	Set to a value which can identify the version of the core from other Power Architecture® cores.

Table 11-1. SPRs with Non-Zero Reset Values (continued)

SPR	Description of Non-Zero Reset Values
SVR	Set to a unique identifier of the integrated device distinct from other SoC products and versions of the same SoC from Freescale Semiconductor. This value is set from signal inputs from the integrated device. All cores in the integrated device contain the same value.
TLB0CFG	Set to configuration information describing the TLB0 capabilities and organization.
TLB1CFG	Set to configuration information describing the TLB1 capabilities and organization.

Other SPRs will need to be set up by software, particularly those SPRs which enable and control various aspects about how the core operates. Table 11-2 lists SPRs for which software should initialize to appropriate values at boot time.

Table 11-2. SPRs to Configure the e500mc

SPR	What to Configure
BUCSR	Branch unit control and status register. See Section 2.11, "Branch Unit Control and Status Register (BUCSR)."
L1CSR0	L1 control and status register. See Section 2.14, "L1 Cache Registers."
L1CSR1	L1 control and status register. See Section 2.14, "L1 Cache Registers."
L1CSR2	L1 control and status register. See Section 2.14, "L1 Cache Registers."
L2CSR0	L2 control and status register. See Section 2.15, "L2 Cache Registers."
L2CSR1	L2 control and status register. See Section 2.15, "L2 Cache Registers."
HID0	Error management can be controlled with HID0. Software can set EMCP in order to receive asynchronous errors from the SoC. EN_L2MMU_MHD can also be set to have hardware detect multiple hits during translation which can result from MMU programming errors or soft errors in the TLB arrays. The core can be configured to strongly order all guarded cache inhibited loads and stores by setting CIGLSO which allows device drivers that perform memory mapped access to cache inhibited guarded memory to not require memory barriers. EN_MAS7_UPDATE should be set to 1 in order to use physical addresses larger than 32 bits. In general this will be the case for many SoCs in which e500mc operates. See Section 2.12, "Hardware Implementation-Dependent Register 0 (HID0)."

11.3.4 MSR and FPSCR

At reset, both the MSR and FPSCR are set to 0. The FPSCR does not require initialization and can be set at a later time before floating point is used depending on which modes software wishes to operate in.

11.4 Timer State

At reset, all the timers are set to 0 and do not require initialization. Timer controls are also set to 0 and when software wishes to begin using timers such as the Decrementer, FIT, or Watchdog timer, software will have to set appropriate values in the TCR.

Both the Time Base and the Alternate Time Base are set to 0 out of reset. The Alternate Time Base will begin counting immediately out of reset, however since Time Base ticks are externally signaled to the core, the Time Base will begin counting once the integrated device is programmed to enable Time Base ticks to the core. See the integrated device reference manual for more information on enabling Time Base ticks to the core.

11.5 L1 Cache State

At reset, the L1 cache (both instruction and data cache) are disabled. The contents of the L1 caches is random, therefore there can be random values for tag bits, data bits, and coherency bits. Valid and lock bits for the L1 caches are cleared at reset. While software does not have to initialize the L1 caches after reset, future processors cores may not provide this initialization upon reset. It is therefore recommended that software should initialize the L1 caches before the L1 cache is enabled. This can be accomplished by flash invalidating the L1 caches and the locks. This will clear the valid bits for all lines and clear the lock bits. The tag bits and data bits do not need to be initialized after flash invalidation, because all lines and tags will be invalid and will be set correctly when a new line is allocated.

To flash invalidate the L1 caches, software should execute the following code sequence prior to enabling the caches:

```
      // L1 data cache
      xor      r4,r4,r4      // set r4 to 0
      ori      r5,r4,0x0102  // set CFI and CFLC bits
      sync
      isync                  // synchronize setting of L1CSR0
      mtspr    L1CSR0,r5     // flash invalidate L1 data cache
      isync                   // synchronize setting of L1CSR0
dloop:
      mfspr    r4,L1CSR0      // get current value
      and.     r4,r4,r5       // test CFI and CFLC bits
      bne      dloop          // check again if not complete
      isync                   // discard prefetched instructions

      // L1 instruction cache
      xor      r4,r4,r4      // set r4 to 0
      ori      r5,r4,0x0102  // set ICFI and ICFLC bits
      sync
      isync                  // synchronize setting of L1CSR1
      mtspr    L1CSR1,r5     // flash invalidate L1 instruction cache
      isync                   // synchronize setting of L1CSR1
iloop:
      mfspr    r4,L1CSR1      // get current value
      and.     r4,r4,r5       // test ICFI and ICFLC bits
      bne      iloop          // check again if not complete
      isync                   // discard prefetched instructions
```

After the caches have been invalidated, they can be enabled by setting the L1CSR0[CE] and L1CSR1[ICE] bits respectively. Parity checking and/or write shadow mode can be enabled as well by setting the appropriate bits in L1CSR0 and L1CSR1. See Section 2.14, “L1 Cache Registers” for descriptions of L1CSR0 and L1CSR1.

11.6 L2 Cache State

At reset, the L2 cache is disabled. The contents of the L2 caches is random, therefore there can be random values for tag bits, data bits, valid bits, coherency bits, and lock bits. Software must properly initialize the L2 cache before the L2 cache is enabled. This can be accomplished by flash invalidating the L2 cache and also flash invalidating the L2 cache locks. This will clear the valid bits and lock bits for all lines. The lock bits must be cleared since the L2 cache supports persistent locks. If the lock bits are not cleared, then on average 50% of the cache will appear to be locked and those lines will not be available for allocation which can have serious performance consequences. The tag bits and data bits do not need to be initialized after flash and flash lock invalidation, because all lines and tags will be invalid and will be set correctly when a new line is allocated.

To flash invalidate the L2 cache, software should execute the following code sequence prior to enabling the L2 cache:

```
      // L2 data cache
      xor      r4,r4,r4      // set r4 to 0
      ori      r5,r4,0x0400  // set L2LFC bit
      oris     r5,r5,0x0020  // set L2FI
      sync
      isync                  // synchronize setting of L2CSR0
      mtspr    L2CSR0,r5     // flash invalidate L2 cache and locks
      isync                   // synchronize setting of L2CSR0
l2loop:
      mfspr    r4,L2CSR0     // get current value
      and.     r5,r5,r4      // compare to see if complete
      bne      l2loop
```

After the L2 cache has been invalidated, it can be enabled by setting the L2CSR0[L2E] bit. Error detection and correction can be enabled as well by setting the appropriate bits in the L2CSR0 register. See Section 2.15, “L2 Cache Registers” for descriptions of L2CSR0 and L2 error management registers.

11.7 Branch Target Buffer State

At reset, the branch target buffer (BTB) and other branch prediction mechanisms are disabled. To obtain full performance of the e500mc, branch prediction mechanisms should be enabled. Also at reset, the contents of the branch target buffer is random, therefore there can be random addresses and random valid bits for branch prediction. While this does not cause any specific problem since the BTB will self correct over time and mispredicted branches will be resolved correctly, software should invalidate the contents of the BTB after reset. This will assist in debugging boot software because fetch accesses will be more deterministic once branch prediction is enabled. The branch prediction mechanisms can be invalidated and enabled by the following code sequence:

```
      // Branch prediction
      xor      r4,r4,r4      // set r4 to 0
      ori      r5,r4,0x0201  // set BBFI and BPEN
      mtspr    BUCSR,r5      // flash invalidate and enable branch prediction
      isync                   // synchronize setting of BUCSR
```

Appendix A
Revision History

This appendix provides a list of the major differences between the *e500mc Core Reference Manual*, Revision 0 through Revision 3.

A.1 Changes From Revision 2 to Revision 3

Major changes to the *e500mc Core Reference Manual*, from Revision 2 to Revision 3, are as follows:

Section, Page	Changes
9.4.2/9-11	Referred users to both Freescale (CodeWarrior) and third-party vendor(s) for additional information regarding software development and debug tools.

A.2 Changes From Revision 1 to Revision 2

Major changes to the *e500mc Core Reference Manual*, from Revision 1 to Revision 2, are as follows:

Section, Page	Changes
2.9.9/2-22	In Figure 2-7, “Machine Check Syndrome Register (MCSR),” and Table 2-8, “Machine Check Syndrome Register (MCSR),” removed field STE.
2.15.2/2-37	Changed description of L2IO noting that when L2IO is set, data transactions are not processed in the L2 cache.
4.9.3.1.2/4-16	In Table 4-5, “Machine Check Exception Sources,” removed self-test error.
4.9.3.4/4-20	In Table 4-8, “Asynchronous Machine Check and NMI Exceptions,” remove self-test error.
9.4.2/9-11	Referred users to third-party tools vendor(s) for additional information regarding QorIQ debug capabilities.

A.3 Changes From Revision 0 to Revision 1

Major changes to the *e500mc Core Reference Manual*, from Revision 0 to Revision 1, are as follows:

Section, Page	Changes
2.15.2/2-37	Updated modes in which write shadow mode is allowed to operate such that L2 cache must be enabled and not in L2IONLY mode.
2.15.2/2-37	In Table 2-18, “L2CSR0 Field Descriptions,” updated the 00 and 10 bitfield setting descriptions and added the following text to the description for bits 50–51: “The Streaming PLRU modes perform a partial update of the PLRU bits when an L2 line is allocated, and a full update on L2 cache hits.”

5.4.2/5-8	After paragraph two, added the following text: “Only certain configurations of cache operation are supported when using write shadow mode. Invalid configurations are not guaranteed to preserve coherency for store operations performed by the processor.”
5.4.2/5-8	Added Table 5-1, “Valid Write Shadow Mode Configurations (when L1CSR2[DCWS] = 1).”
5.7/5-23	Rewrote section and added text to clarify the description of the cache flushing operation.

Appendix B
Simplified Mnemonics

This appendix describes simplified mnemonics, which are provided for easier coding of assembly language programs. Simplified mnemonics are defined for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions defined by the Power ISA™ and by implementations of and extensions to the Power ISA.

B.1 Overview

Simplified (or extended) mnemonics allow an assembly-language programmer to program using more intuitive mnemonics and symbols than the instructions and syntax defined by the instruction set architecture. For example, to code the conditional call “branch to an absolute target if CR4 specifies a greater than condition, setting the LR without simplified mnemonics, the programmer would write the branch conditional instruction, **bc 12,17,***target*. The simplified mnemonic, branch if greater than, **bgt cr4,***target*, incorporates the conditions. Not only is it easier to remember the symbols than the numbers when programming, it is also easier to interpret simplified mnemonics when reading existing code.

Although the Power ISA documents include a set of simplified mnemonics, these are not a formal part of the architecture, but rather a recommendation for assemblers that support the instruction set.

Many simplified mnemonics have been added to those originally included in the architecture documentation. Some assemblers created their own, and others have been added to support extensions to the instruction set. Simplified mnemonics have been added for new architecturally defined and new implementation-specific special-purpose registers (SPRs). These simplified mnemonics are described only in a very general way.

B.2 Subtract Simplified Mnemonics

This section describes simplified mnemonics for subtract instructions.

B.2.1 Subtract Immediate

There is no subtract immediate instruction, however, its effect is achieved by negating the immediate operand of an Add Immediate instruction, **addi**. Simplified mnemonics include this negation, making the intent of the computation more clear. These are listed in Table B-1.

Table B-1. Subtract Immediate Simplified Mnemonics

Simplified Mnemonic	Standard Mnemonic
subi r**D**,r**A**,value	**addi** r**D**,r**A**,–value
subis r**D**,r**A**,value	**addis** r**D**,r**A**,–value

Table B-1. Subtract Immediate Simplified Mnemonics (continued)

Simplified Mnemonic	Standard Mnemonic
subic **r**D,**r**A,value	**addic** **r**D,**r**A,–value
subic. **r**D,**r**A,value	**addic.** **r**D,**r**A,–value

B.2.2 Subtract

Subtract from instructions subtract the second operand (**r**A) from the third (**r**B). The simplified mnemonics in Table B-2 use the more common order in which the third operand is subtracted from the second.

Table B-2. Subtract Simplified Mnemonics

Simplified Mnemonic	Standard Mnemonic[1]
sub[**o**][.] **r**D,**r**A,**r**B	**subf**[**o**][.] **r**D,**r**B,**r**A
subc[**o**][.] **r**D,**r**A,**r**B	**subfc**[**o**][.] **r**D,**r**B,**r**A

[1] **r**D,**r**B,**r**A is not the standard order for the operands. The order of **r**B and **r**A is reversed to show the equivalent behavior of the simplified mnemonic.

B.3 Rotate and Shift Simplified Mnemonics

Rotate and shift instructions provide powerful, general ways to manipulate register contents, but can be difficult to understand. Simplified mnemonics are provided for the following operations:

Extract	Select a field of *n* bits starting at bit position *b* in the source register; left or right justify this field in the target register; clear all other bits of the target register.
Insert	Select a left- or right-justified field of *n* bits in the source register; insert this field starting at bit position *b* of the target register; leave other bits of the target register unchanged.
Rotate	Rotate the contents of a register right or left *n* bits without masking.
Shift	Shift the contents of a register right or left *n* bits, clearing vacated bits (logical shift).
Clear	Clear the leftmost or rightmost *n* bits of a register.
Clear left and shift left	Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a (known nonnegative) array index by the width of an element.

B.3.1 Operations on Words

The simplified mnemonics in Table B-3 can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

Table B-3. Word Rotate and Shift Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Extract and left justify word immediate	**extlwi** r**A**,r**S**,*n*,*b* (*n* > 0)	**rlwinm** r**A**,r**S**,*b*,**0**,*n* – 1
Extract and right justify word immediate	**extrwi** r**A**,r**S**,*n*,*b* (*n* > 0)	**rlwinm** r**A**,r**S**,*b* + *n*, 32 – *n*,**31**
Insert from left word immediate	**inslwi** r**A**,r**S**,*n*,*b* (*n* > 0)	**rlwimi** r**A**,r**S**,32 – *b*,*b*,(*b* + *n*) – 1
Insert from right word immediate	**insrwi** r**A**,r**S**,*n*,*b* (*n* > 0)	**rlwimi** r**A**,r**S**,32 – (*b* + *n*),*b*,(*b* + *n*) – 1
Rotate left word immediate	**rotlwi** r**A**,r**S**,*n*	**rlwinm** r**A**,r**S**,*n*,**0**,**31**
Rotate right word immediate	**rotrwi** r**A**,r**S**,*n*	**rlwinm** r**A**,r**S**,32 – *n*,**0**,**31**
Rotate word left	**rotlw** r**A**,r**S**,r**B**	**rlwnm** r**A**,r**S**,r**B**,**0**,**31**
Shift left word immediate	**slwi** r**A**,r**S**,*n* (*n* < 32)	**rlwinm** r**A**,r**S**,*n*,**0**,31 – *n*
Shift right word immediate	**srwi** r**A**,r**S**,*n* (*n* < 32)	**rlwinm** r**A**,r**S**,32 – *n*,*n*,**31**
Clear left word immediate	**clrlwi** r**A**,r**S**,*n* (*n* < 32)	**rlwinm** r**A**,r**S**,**0**,*n*,**31**
Clear right word immediate	**clrrwi** r**A**,r**S**,*n* (*n* < 32)	**rlwinm** r**A**,r**S**,**0**,**0**,31 – *n*
Clear left and shift left word immediate	**clrlslwi** r**A**,r**S**,*b*,*n* (*n* ≤ *b* ≤ 31)	**rlwinm** r**A**,r**S**,*n*,*b* – *n*,31 – *n*

Examples using word mnemonics follow:

1. Extract the sign bit (bit 0) of **r**S and place the result right-justified into **r**A.
   extrwi rA,**r**S,**1,0** equivalent to **rlwinm r**A,**r**S,**1,31,31**
2. Insert the bit extracted in (1) into the sign bit (bit 0) of **r**B.
   insrwi rB,**r**A,**1,0** equivalent to **rlwimi r**B,**r**A,**31,0,0**
3. Shift the contents of **r**A left 8 bits.
   slwi rA,**r**A,**8** equivalent to **rlwinm r**A,**r**A,**8,0,23**
4. Clear the high-order 16 bits of **r**S and place the result into **r**A.
   clrlwi rA,**r**S,**16** equivalent to **rlwinm r**A,**r**S,**0,16,31**

B.4 Branch Instruction Simplified Mnemonics

Branch conditional instructions can be coded with the operations, a condition to be tested, and a prediction, as part of the instruction mnemonic rather than as numeric operands (the BO and BI operands). Table B-4 shows the four general types of branch instructions. Simplified mnemonics are defined only for branch instructions that include BO and BI operands; there is no need to simplify unconditional branch mnemonics.

Table B-4. Branch Instructions

Instruction Name	Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI

The BO and BI operands correspond to two fields in the instruction opcode, as Figure B-1 shows for Branch Conditional (**bc**, **bca**, **bcl**, and **bcla**) instructions.

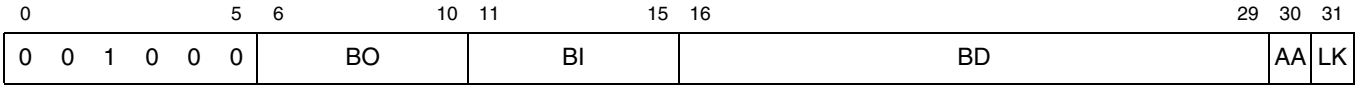


Figure B-1. Branch Conditional (bc) Instruction Format

The BO operand specifies branch operations that involve decrementing CTR. It is also used to determine whether testing a CR bit causes a branch to occur if the condition is true or false.

The BI operand identifies a CR bit to test (whether a comparison is less than or greater than, for example). The simplified mnemonics avoid the need to memorize the numerical values for BO and BI.

For example, **bc 16,0,***target* is a conditional branch that, as a BO value of 16 (0b1_0000) indicates, decrements CTR, then branches if the decremented CTR is not zero. The operation specified by BO is abbreviated as **d** (for decrement) and **nz** (for not zero), which replace the **c** in the original mnemonic; so the simplified mnemonic for **bc** becomes **bdnz**. The branch does not depend on a condition in the CR, so BI can be eliminated, reducing the expression to **bdnz** *target*.

In addition to CTR operations, the BO operand provides an optional prediction bit and a true or false indicator can be added. For example, if the previous instruction should branch only on an equal condition in CR0, the instruction becomes **bc 8,2,***target*. To incorporate a true condition, the BO value becomes 8 (as shown in Table B-6); the CR0 equal field is indicated by a BI value of 2 (as shown in Table B-7). Incorporating the branch-if-true condition adds a '**t**' to the simplified mnemonic, **bdnzt.** The equal condition, that is specified by a BI value of 2 (indicating the EQ bit in CR0) is replaced by the **eq** symbol. Using the simplified mnemonic and the **eq** operand, the expression becomes **bdnzt eq,***target*.

This example tests CR0[EQ]; however, to test the equal condition in CR5 (CR bit 22), the expression becomes **bc 8,22,***target*. The BI operand of 22 indicates CR[22] (CR5[2], or BI field 0b10110), as shown in Table B-7. This can be expressed as the simplified mnemonic. **bdnzt 4 * cr5 + eq,***target*.

The notation, **4 * cr5 + eq** may at first seem awkward, but it eliminates computing the value of the CR bit. It can be seen that (4 * 5) + 2 = 22. Note that although 32-bit registers in Power ISA processors are numbered 32–63, only values 0–31 are valid (or possible) for BI operands. The encoding of the field in the instruction uses numbering from 0 - 31 and the instruction converts this into the architecturally described bit number by adding 32.

B.4.1 Key Facts about Simplified Branch Mnemonics

The following key points are helpful in understanding how to use simplified branch mnemonics:

- All simplified branch mnemonics eliminate the BO operand, so if any operand is present in a branch simplified mnemonic, it is the BI operand (or a reduced form of it).
- If the CR is not involved in the branch, the BI operand can be deleted.
- If the CR is involved in the branch, the BI operand can be treated in the following ways:

— It can be specified as a numeric value, just as it is in the architecturally defined instruction, or it can be indicated with an easier to remember formula, **4 * cr***n* + [test bit symbol], where *n* indicates the CR field number.

— The condition of the test bit (eq, lt, gt, and so) can be incorporated into the mnemonic, leaving the need for an operand that defines only the CR field.

– If the test bit is in CR0, no operand is needed.

– If the test bit is in CR1–CR7, the BI operand can be replaced with a **cr**S operand (that is, **cr1**, **cr2**, **cr3**, and so forth).

B.4.2 Eliminating the BO Operand

The 5-bit BO field, shown in Figure B-2, encodes the following operations in conditional branch instructions:

- Decrement count register (CTR)
  — And test if result is equal to zero
  — And test if result is not equal to zero
- Test condition register (CR)
  — Test condition true
  — Test condition false
- Branch prediction (taken, fall through). If the prediction bit, *y*, is needed, it is signified by appending a plus or minus sign as described in Section B.4.3, "Incorporating the BO Branch Prediction."

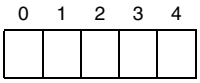


Figure B-2. BO Field (Bits 6–10 of the Instruction Encoding)

BO bits can be interpreted individually as described in Table B-5.

Table B-5. BO Bit Encodings

BO Bit	Description
0	If set, ignore the CR bit comparison.
1	If set, the CR bit comparison is against true, if not set the CR bit comparison is against false
2	If set, the CTR is not decremented.
3	If BO[2] is set, this bit determines whether the CTR comparison is for equal to zero or not equal to zero.
4	Used for static branch prediction. Use of the this bit is optional and independent from the interpretation of the rest of the BO operand. Because simplified branch mnemonics eliminate the BO operand, this bit (the *t* bit) and other branch prediction hint bits (the "*a*" bit) are programmed by adding a plus or minus sign to the simplified mnemonic, as described in Section B.4.3, "Incorporating the BO Branch Prediction."

Thus, a BO encoding of 10100 (decimal 20) means ignore the CR bit comparison and do not decrement the CTR—in other words, branch unconditionally. Encodings for the BO operand are shown in Table B-6.

A *z* bit indicates that the bit is ignored. However, these bits should be cleared, as they may be assigned a meaning in a future version of the architecture.

As shown in Table B-6, the '**c**' in the standard mnemonic is replaced with the operations otherwise specified in the BO field, (**d** for decrement, **z** for zero, **nz** for nonzero, **t** for true, and **f** for false).

NOTE

The test of when a the CTR reaches 0 varies between 32-bit mode and 64-bit mode. M = 32 in 32-bit mode (of a 64-bit implementation) and M = 0 in 64-bit mode.

Table B-6. BO Operand Encodings

BO Field	Value[1] (Decimal)	Description	Symbol
0000*z*[2]	0	Decrement the CTR, then branch if the decremented CTR ≠ 0; condition is FALSE.	**dnzf**
0001*z*	2	Decrement the CTR, then branch if the decremented CTR = 0; condition is FALSE.	**dzf**
001*at*[3]	4	Branch if the condition is FALSE.[4] Note that 'false' and 'four' both start with 'f'.	**f**
0100*z*	8	Decrement the CTR, then branch if the decremented CTR ≠ 0; condition is TRUE.	**dnzt**
0101*z*	10	Decrement the CTR, then branch if the decremented CTR = 0; condition is TRUE.	**dzt**
011*at*	12	Branch if the condition is TRUE. [2] Note that 'true' and 'twelve' both start with 't'.	**t**
1*a*00*t*[5]	16	Decrement the CTR, then branch if the decremented CTR ≠ 0.	**dnz**[6]
1*a*01*t*[5]	18	Decrement the CTR, then branch if the decremented CTR = 0.	**dz** [6]
1*z*1*zz* [5]	20	Branch always.	—

1 Assumes t = z = 0. Section B.4.3, "Incorporating the BO Branch Prediction," describes how to use simplified mnemonics to program the *y* bit for static prediction.

2 A *z* bit indicates a bit that is ignored. However, these bits should be cleared, as they may be assigned a meaning in a future version of the architecture.

3 The *a* and *t* bits are used for static branch prediction hints such that *at* = 0b00 specifies no hint, 0b10 specifies the branch is very likely not to be taken, and 0b11 specifies the branch is very likely to be taken.

4 Instructions for which B0 is 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field, as described in Section B.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

5 Simplified mnemonics for branch instructions that do not test CR bits (BO = 16, 18, and 20) should specify only a target. Otherwise a programming error may occur.

6 Notice that these instructions do not use the branch if condition true or false operations. For that reason, simplified mnemonics for these should not specify a BI operand.

B.4.3 Incorporating the BO Branch Prediction

As shown in Table B-6, the low-order bit (*t* bit) of the BO field along with the *a* bit provides a hint about whether the branch is likely to be taken (static branch prediction). Assemblers should clear these bits unless otherwise directed. This default action indicates the following:

- A branch conditional with a negative displacement field is predicted to be taken.

- A branch conditional with a nonnegative displacement field is predicted not to be taken (fall through).
- A branch conditional to an address in the LR or CTR is predicted not to be taken (fall through).

If the likely outcome (branch or fall through) of a given branch conditional instruction is known, a suffix can be added to the mnemonic that tells the assembler how to set the *at* bits. That is, '+' indicates that the branch is to be taken and '–' indicates that the branch is not to be taken. This suffix can be added to any branch conditional mnemonic, standard or simplified.

For relative and absolute branches (**bc**[**l**][**a**]), the setting of the *at* bits depends on whether the displacement field is negative or nonnegative. For negative displacement fields, coding the suffix '+' causes the bit to be cleared, and coding the suffix '–' causes the bit to be set. For nonnegative displacement fields, coding the suffix '+' causes the bit to be set, and coding the suffix '–' causes the bit to be cleared.

For branches to an address in the LR or CTR (**bclr**[**l**] or **bcctr**[**l**]), coding the suffix '+' causes the *at* bits to be set, and coding the suffix '–' causes the *at* bits to be set to 0b10.

Examples of branch prediction follow:

1. Branch if CR0 reflects less than condition, specifying that the branch should be predicted as taken.
   blt+ *target*
2. Same as (1), but target address is in the LR and the branch should be predicted as not taken.
   bltlr–

B.4.4 The BI Operand—CR Bit and Field Representations

With standard branch mnemonics, the BI operand is used when it is necessary to test a CR bit, as shown in the example in Section B.4, "Branch Instruction Simplified Mnemonics."

With simplified mnemonics, the BI operand is handled differently depending on whether the simplified mnemonic incorporates a CR condition to test, as follows:

- Some branch simplified mnemonics incorporate only the BO operand. These simplified mnemonics can use the architecturally defined BI operand to specify the CR bit, as follows:
  - The BI operand can be presented exactly as it is with standard mnemonics—as a decimal number, 0–31.
  - Symbols can be used to replace the decimal operand, as shown in the example in Section B.4, "Branch Instruction Simplified Mnemonics," where **bdnzt 4 * cr5 + eq,***target* could be used instead of **bdnzt 22,***target*. This is described in Section B.4.4.1.1, "Specifying a CR Bit."

NOTE

The simplified mnemonics in Section B.4.5, "Simplified Mnemonics that Incorporate the BO Operand," use one of these two methods to specify a CR bit.

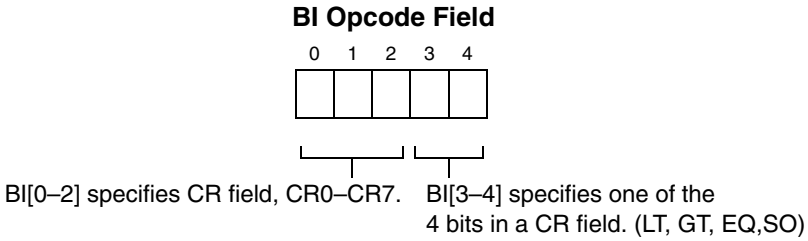
- Additional simplified mnemonics are specified that incorporate CR conditions that would otherwise be specified by the BI operand, so the BI operand is replaced by the **cr**S operand to specify the CR field, CR0–CR7. See Section B.4.4.1, "BI Operand Instruction Encoding."

These mnemonics are described in Section B.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

B.4.4.1 BI Operand Instruction Encoding

The entire 5-bit BI field, shown in Figure B-3, represents the bit number for the CR bit to be tested. For standard branch mnemonics and for branch simplified mnemonics that do not incorporate a CR condition, the BI operand provides all 5 bits.

For simplified branch mnemonics described in Section B.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)," the BI operand is replaced by a **cr**S operand. To understand this, it is useful to view the BI operand as comprised of two parts. As Figure B-3 shows, BI[0–2] indicates the CR field and BI[3–4] represents the condition to test.



Simplified mnemonics based on CR conditions but not CTR values—BO = 12 (branch if true) and BO = 4 branch if false)	Specified by a separate, reduced BI operand (**cr**S)	Incorporated into the simplified mnemonic.
Standard branch mnemonics and simplified mnemonics based on CTR values	The BI operand specifies the entire 5-bit field. If CR0 is used, the bit can be identified by LT, GT, EQ, or SO. If CR1–CR7 are used, the form 4 * **cr**S + LT|GT|EQ|SO can be used.	

Figure B-3. BI Field (Bits 11–14 of the Instruction Encoding)

Integer record-form instructions update CR0 and floating-point record-form instructions update CR1 as described in Table B-7.

B.4.4.1.1 Specifying a CR Bit

Note that the AIM version the PowerPC architecture numbers CR bits 0–31 and Book E numbers them 32–63. However, no adjustment is necessary to the code; in Book E devices, 32 is automatically added to the BI value, as shown in Table B-7 and Table B-8.

Table B-7. CR0 and CR1 Fields as Updated by Integer and Floating-Point Instructions

CR*n* Bit	CR Bits (Operand)	BI		Description
		0–2	3–4	
CR0[0]	32(0)	000	00	Negative (LT)—Set when the result is negative.
CR0[1]	33(1)	000	01	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	34(2)	000	10	Zero (EQ)—Set when the result is zero.
CR0[3]	35(3)	000	11	Summary overflow (SO). Copy of XER[SO] at the instruction's completion.

Some simplified mnemonics incorporate only the BO field (as described Section B.4.2, “Eliminating the BO Operand”). If one of these simplified mnemonics is used and the CR must be accessed, the BI operand can be specified either as a numeric value or by using the symbols in Table B-8.

Compare word instructions (described in Section B.5, “Compare Word Simplified Mnemonics”), floating-point compare instructions, move to CR instructions, and others can also modify CR fields, so CR0 and CR1 may hold values that do not adhere to the meanings described in Table B-7. CR logical instructions, described in Section B.7, “Condition Register Logical Simplified Mnemonics,” can update individual CR bits.

Table B-8. BI Operand Settings for CR Fields for Branch Comparisons

CR*n* Bit	Bit Expression	CR Bits		BI		Description
		BI Operand)	Power ISA Bit Number	0–2	3–4	
CR*n*[0]	**4 * cr0 + lt** (or **lt**) **4 * cr1 + lt** **4 * cr2 + lt** **4 * cr3+ lt** **4 * cr4 + lt** **4 * cr5 + lt** **4 * cr6 + lt** **4 * cr7 + lt**	0 4 8 12 16 20 24 28	32 36 40 44 48 52 56 60	000 001 010 011 100 101 110 111	00	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A < **fr**B.
CR*n*[1]	**4 * cr0 + gt** (or **gt**) **4 * cr1 + gt** **4 * cr2 + gt** **4 * cr3+ gt** **4 * cr4 + gt** **4 * cr5 + gt** **4 * cr6 + gt** **4 * cr7 + gt**	1 5 9 13 17 21 25 29	33 37 41 45 49 53 57 61	000 001 010 011 100 101 110 111	01	Greater than or floating-point greater than (GT, FG). For integer compare instructions: rA > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A > **fr**B.
CR*n*[2]	**4 * cr0 + eq** (or **eq**) **4 * cr1 + eq** **4 * cr2 + eq** **4 * cr3+ eq** **4 * cr4 + eq** **4 * cr5 + eq** **4 * cr6 + eq** **4 * cr7 + eq**	2 6 10 14 18 22 26 30	34 38 42 46 50 54 58 62	000 001 010 011 100 101 110 111	10	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
CR*n*[3]	**4 * cr0 + so/un** (or **so/un**) **4 * cr1 + so/un** **4* cr2 + so/un** **4* cr3 + so/un** **4* cr4 + so/un** **4* cr5 + so/un** **4* cr6 + so/un** **4* cr7 + so/un**	3 7 11 15 19 23 27 31	35 39 43 47 51 55 59 63	000 001 010 011 100 101 110 111	11	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at instruction completion. For floating-point compare instructions, one or both of frA and frB is a NaN.

To provide simplified mnemonics for every possible combination of BO and BI (that is, including bits that identified the CR field) would require $2^{10} = 1024$ mnemonics, most of that would be only marginally

useful. The abbreviated set in Section B.4.5, "Simplified Mnemonics that Incorporate the BO Operand," covers useful cases. Unusual cases can be coded using a standard branch conditional syntax.

B.4.4.1.2 The crS Operand

The **cr**S symbols are shown in Table B-9. Note that either the symbol or the operand value can be used in the syntax used with the simplified mnemonic.

Table B-9. CR Field Identification Symbols

Symbol	BI[0–2]	CR Bits
cr0 (default, can be eliminated from syntax)	000	32–35
cr1	001	36–39
cr2	010	40–43
cr3	011	44–47
cr4	100	48–51
cr5	101	52–55
cr6	110	56–59
cr7	111	60–63

To identify a CR bit, an expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol can be used, (for example, **cr0 * 4 + eq**).

B.4.5 Simplified Mnemonics that Incorporate the BO Operand

The mnemonics in Table B-10 allow common BO operand encodings to be specified as part of the mnemonic, along with the absolute address (AA) and set link register bits (LK). There are no simplified mnemonics for relative and absolute unconditional branches. For these, the basic mnemonics **b**, **ba**, **bl**, and **bla** are used.

Table B-10. Branch Simplified Mnemonics

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc	**bca**	**bclr**	**bcctr**	**bcl**	**bcla**	**bclrl**	**bcctrl**
Branch unconditionally [1]	—	—	**blr**	**bctr**	—	—	**blrl**	**bctrl**
Branch if condition true	**bt**	**bta**	**btlr**	**btctr**	**btl**	**btla**	**btlrl**	**btctrl**
Branch if condition false	**bf**	**bfa**	**bflr**	**bfctr**	**bfl**	**bfla**	**bflrl**	**bfctrl**
Decrement CTR, branch if CTR ≠ 0 [1]	**bdnz**	**bdnza**	**bdnzlr**	—	**bdnzl**	**bdnzla**	**bdnzlrl**	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bdnzt**	**bdnzta**	**bdnztlr**	—	**bdnztl**	**bdnztla**	**bdnztlrl**	—
Decrement CTR, branch if CTR ≠ 0 and condition false	**bdnzf**	**bdnzfa**	**bdnzflr**	—	**bdnzfl**	**bdnzfla**	**bdnzflrl**	—

Table B-10. Branch Simplified Mnemonics (continued)

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc	**bca**	**bclr**	**bcctr**	**bcl**	**bcla**	**bclrl**	**bcctrl**
Decrement CTR, branch if CTR = 0 [1]	**bdz**	**bdza**	**bdzlr**	—	**bdzl**	**bdzla**	**bdzlrl**	—
Decrement CTR, branch if CTR = 0 and condition true	**bdzt**	**bdzta**	**bdztlr**	—	**bdztl**	**bdztla**	**bdztlrl**	—
Decrement CTR, branch if CTR = 0 and condition false	**bdzf**	**bdzfa**	**bdzflr**	—	**bdzfl**	**bdzfla**	**bdzflrl**	—

[1] Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise a programming error may occur.

Table B-10 shows the syntax for basic simplified branch mnemonics

Table B-11. Branch Instructions

Instruction	Standard Mnemonic	Syntax	Simplified Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr	N/A, syntax does not include BO	
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr	**b***x*[1] (**b***x***a b***x***l b***x***la**)	BI[2]target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI	**b***x***lr** (**b***x***lrl**)	BI
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI	**b***x***ctr** (**b***x***ctrl**)	BI

[1] *x* stands for one of the symbols in Table B-6, where applicable.

[2] BI can be a numeric value or an expression as shown in Table B-9.

The simplified mnemonics in Table B-10 that test a condition require a corresponding CR bit as the first operand (as examples 2–5 below illustrate). The symbols in Table B-9 can be used in place of a numeric value.

B.4.5.1 Examples that Eliminate the BO Operand

The simplified mnemonics in Table B-10 are used in the following examples:

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR) (note that no CR bits are tested).
   bdnz *target* equivalent to **bc 16,0,***target*
   Because this instruction does not test a CR bit, the simplified mnemonic should specify only a target operand. Specifying a CR (for example, **bdnz** 0,*target* or **bdnz cr0,***target*) may be considered a programming error. Subsequent examples test conditions).
2. Same as (1) but branch only if CTR is nonzero and equal condition in CR0.
   bdnzt eq,*target* equivalent to **bc 8,2,***target*
   Other equivalents include **bdnzt 2,***target* or the unlikely **bdnzt 4*cr0+eq,***target*
3. Same as (2), but equal condition is in CR5.
   bdnzt 4 * cr5 + eq,*target* equivalent to **bc 8,22,***target*

bdnzt 22,*target* would also work

4. Branch if bit 59 of CR is false.
   bf 27,*target* equivalent to **bc 4,27,***target*
   bf 4*cr6+so,*target* would also work
5. Same as (4), but set the link register. This is a form of conditional call.
   bfl 27,*target* equivalent to **bcl 4,27,***target*

Table B-12 lists simplified mnemonics and syntax for **bc** and **bca** without LR updating.

Table B-12. Simplified Mnemonics for bc and bca without LR Update

Branch Semantics	bc	Simplified Mnemonic	bca	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true[1]	**bc 12,BI,**target	**bt BI,**target	**bca 12,BI,**target	**bta BI,**target
Branch if condition false [1]	**bc 4,BI,**target	**bf BI,**target	**bca 4,BI,**target	**bfa BI,**target
Decrement CTR, branch if CTR ≠ 0	**bc 16,0,**target	**bdnz** target[2]	**bca 16,0,**target	**bdnza** target [2]
Decrement CTR, branch if CTR ≠ 0 and condition true	**bc 8,BI,**target	**bdnzt BI,**target	**bca 8,BI,**target	**bdnzta BI,**target
Decrement CTR, branch if CTR ≠ 0 and condition false	**bc 0,BI,**target	**bdnzf BI,**target	**bca 0,BI,**target	**bdnzfa BI,**target
Decrement CTR, branch if CTR = 0	**bc 18,0,**target	**bdz** target [2]	**bca 18,0,**target	**bdza** target [2]
Decrement CTR, branch if CTR = 0 and condition true	**bc 10,BI,**target	**bdzt BI,**target	**bca 10,BI,**target	**bdzta BI,**target
Decrement CTR, branch if CTR = 0 and condition false	**bc 2,BI,**target	**bdzf BI,**target	**bca 2,BI,**target	**bdzfa BI,**target

[1] Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field, as described in Section B.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

[2] Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise a programming error may occur.

Table B-13 lists simplified mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table B-13. Simplified Mnemonics for bclr and bcctr without LR Update

Branch Semantics	bclr	Simplified Mnemonic	bcctr	Simplified Mnemonic
Branch unconditionally	**bclr 20,0**	**blr** [1]	**bcctr 20,0**	**bctr** [1]
Branch if condition true [2]	**bclr 12,BI**	**btlr BI**	**bcctr 12,BI**	**btctr BI**
Branch if condition false [2]	**bclr 4,BI**	**bflr BI**	**bcctr 4,BI**	**bfctr BI**
Decrement CTR, branch if CTR ≠ 0	**bclr 16,BI**	**bdnzlr BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bclr 8,BI**	**bdnztlr BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition false	**bclr 0,BI**	**bdnzflr BI**	—	—
Decrement CTR, branch if CTR = 0	**bclr 18,0**	**bdzlr** [1]	—	—

Table B-13. Simplified Mnemonics for bclr and bcctr without LR Update (continued)

Branch Semantics	bclr	Simplified Mnemonic	bcctr	Simplified Mnemonic
Decrement CTR, branch if CTR = 0 and condition true	**bclr 8,BI**	**bdnztlr BI**	—	—
Decrement CTR, branch if CTR = 0 and condition false	**bclr 2,BI**	**bdzflr BI**	—	—

1 Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.

2 Instructions for which B0 is 12 (branch if condition true) or 4 (branch if condition false) do not depend on a CTR value and can be alternately coded by incorporating the condition specified by the BI field. See Section B.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

Table B-14 provides simplified mnemonics and syntax for **bcl** and **bcla**.

Table B-14. Simplified Mnemonics for bcl and bcla with LR Update

Branch Semantics	bcl	Simplified Mnemonic	bcla	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true [1]	**bcl 12,BI,**target	**btl BI,**target	**bcla 12,BI,**target	**btla BI,**target
Branch if condition false [1]	**bcl 4,BI,**target	**bfl BI,**target	**bcla 4,BI,**target	**bfla BI,**target
Decrement CTR, branch if CTR ≠ 0	**bcl 16,0,**target	**bdnzl** target [2]	**bcla 16,0,**target	**bdnzla** target [2]
Decrement CTR, branch if CTR ≠ 0 and condition true	**bcl 8,0,**target	**bdnztl BI,**target	**bcla 8,BI,**target	**bdnztla BI,**target
Decrement CTR, branch if CTR ≠ 0 and condition false	**bcl 0,BI,**target	**bdnzfl BI,**target	**bcla 0,BI,**target	**bdnzfla BI,**target
Decrement CTR, branch if CTR = 0	**bcl 18,BI,**target	**bdzl** target [2]	**bcla 18,BI,**target	**bdzla** target [2]
Decrement CTR, branch if CTR = 0 and condition true	**bcl 10,BI,**target	**bdztl BI,**target	**bcla 10,BI,**target	**bdztla BI,**target
Decrement CTR, branch if CTR = 0 and condition false	**bcl 2,BI,**target	**bdzfl BI,**target	**bcla 2,BI,**target	**bdzfla BI,**target

1 Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field. See Section B.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

2 Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. A programming error may occur.

Table B-15 provides simplified mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table B-15. Simplified Mnemonics for bclrl and bcctrl with LR Update

Branch Semantics	bclrl	Simplified Mnemonic	bcctrl	Simplified Mnemonic
Branch unconditionally	**bclrl 20,0**	**blrl** [1]	**bcctrl 20,0**	**bctrl** [1]
Branch if condition true	**bclrl 12,BI**	**btlrl BI**	**bcctrl 12,BI**	**btctrl BI**
Branch if condition false	**bclrl 4,BI**	**bflrl BI**	**bcctrl 4,BI**	**bfctrl BI**
Decrement CTR, branch if CTR ≠ 0	**bclrl 16,0**	**bdnzlrl** [1]	—	—

Table B-15. Simplified Mnemonics for bclrl and bcctrl with LR Update (continued)

Branch Semantics	bclrl	Simplified Mnemonic	bcctrl	Simplified Mnemonic
Decrement CTR, branch if CTR ≠ 0 and condition true	**bclrl 8,BI**	**bdnztlrl BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition false	**bclrl 0,BI**	**bdnzflrl BI**	—	—
Decrement CTR, branch if CTR = 0	**bclrl 18,0**	**bdzlrl** [1]	—	—
Decrement CTR, branch if CTR = 0 and condition true	**bclrl 10, BI**	**bdztlrl BI**	—	—
Decrement CTR, branch if CTR = 0 and condition false	**bclrl 2,BI**	**bdzflrl BI**	—	—

[1] Simplified mnemonics for branch instructions that do not test a CR bit should not specify one. A programming error may occur.

B.4.6 Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)

The mnemonics in Table B-18 are variations of the branch-if-condition-true (BO = 12) and branch-if-condition-false (BO = 4) encodings. Because these instructions do not depend on the CTR, the true/false conditions specified by BO can be combined with the CR test bit specified by BI to create a different set of simplified mnemonics that eliminates the BO operand and the portion of the BI operand (BI[3–4]) that specifies one of the four possible test bits. However, the simplified mnemonic cannot specify in which of the eight CR fields the test bit falls, so the BI operand is replaced by a **cr**S operand.

The standard codes shown in Table B-16 are used for the most common combinations of branch conditions. Note that for ease of programming, these codes include synonyms; for example, less than or equal (**le**) and not greater than (**ng**) achieve the same result.

NOTE

A CR field symbol, **cr0**–**cr7**, is used as the first operand after the simplified mnemonic. If CR0 is used, no **cr**S is necessary.

Table B-16. Standard Coding for Branch Conditions

Code	Description	Equivalent	Bit Tested
lt	Less than	—	LT
le	Less than or equal (equivalent to **ng**)	**ng**	GT
eq	Equal	—	EQ
ge	Greater than or equal (equivalent to **nl**)	**nl**	LT
gt	Greater than	—	GT
nl	Not less than (equivalent to **ge**)	**ge**	LT
ne	Not equal	—	EQ
ng	Not greater than (equivalent to **le**)	**le**	GT
so	Summary overflow	—	SO
ns	Not summary overflow	—	SO

Table B-17 shows the syntax for simplified branch mnemonics that incorporate CR conditions. Here, **cr**S replaces a BI operand to specify only a CR field (because the specific CR bit within the field is now part of the simplified mnemonic. Note that the default is CR0; if no **cr**S is specified, CR0 is used.

Table B-17. Branch Instructions and Simplified Mnemonics that Incorporate CR Conditions

Instruction	Standard Mnemonic	Syntax	Simplified Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr	—	
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr	**b***x* [1](**b***x***a b***x***l b***x***la**)	**cr**S[2],target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI	**b***x***lr** (**b***x***lrl**)	**cr**S
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI	**b***x***ctr** (**b***x***ctrl**)	**cr**S

1 *x* stands for one of the symbols in Table B-16, where applicable.

2 BI can be a numeric value or an expression as shown in Table B-9.

Table B-18 shows the simplified branch mnemonics incorporating conditions.

Table B-18. Simplified Mnemonics with Comparison Conditions

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc	**bca**	**bclr**	**bcctr**	**bcl**	**bcla**	**bclrl**	**bcctrl**
Branch if less than	**blt**	**blta**	**bltlr**	**bltctr**	**bltl**	**bltla**	**bltlrl**	**bltctrl**
Branch if less than or equal	**ble**	**blea**	**blelr**	**blectr**	**blel**	**blela**	**blelrl**	**blectrl**
Branch if equal	**beq**	**beqa**	**beqlr**	**beqctr**	**beql**	**beqla**	**beqlrl**	**beqctrl**
Branch if greater than or equal	**bge**	**bgea**	**bgelr**	**bgectr**	**bgel**	**bgela**	**bgelrl**	**bgectrl**
Branch if greater than	**bgt**	**bgta**	**bgtlr**	**bgtctr**	**bgtl**	**bgtla**	**bgtlrl**	**bgtctrl**
Branch if not less than	**bnl**	**bnla**	**bnllr**	**bnlctr**	**bnll**	**bnlla**	**bnllrl**	**bnlctrl**
Branch if not equal	**bne**	**bnea**	**bnelr**	**bnectr**	**bnel**	**bnela**	**bnelrl**	**bnectrl**
Branch if not greater than	**bng**	**bnga**	**bnglr**	**bngctr**	**bngl**	**bngla**	**bnglrl**	**bngctrl**
Branch if summary overflow	**bso**	**bsoa**	**bsolr**	**bsoctr**	**bsol**	**bsola**	**bsolrl**	**bsoctrl**
Branch if not summary overflow	**bns**	**bnsa**	**bnslr**	**bnsctr**	**bnsl**	**bnsla**	**bnslrl**	**bnsctrl**
Branch if unordered	**bun**	**buna**	**bunlr**	**bunctr**	**bunl**	**bunla**	**bunlrl**	**bunctrl**
Branch if not unordered	**bnu**	**bnua**	**bnulr**	**bnuctr**	**bnul**	**bnula**	**bnulrl**	**bnuctrl**

Instructions using the mnemonics in Table B-18 indicate the condition bit, but not the CR field. If no field is specified, CR0 is used. The CR field symbols defined in Table B-9 (**cr0**–**cr7**) are used for this operand, as shown in examples 2–4 below.

B.4.6.1 Branch Simplified Mnemonics that Incorporate CR Conditions: Examples

The following examples use the simplified mnemonics shown in Table B-18:

1. Branch if CR0 reflects not-equal condition.
   bne *target* equivalent to **bc 4,2,***target*
2. Same as (1) but condition is in CR3.
   bne cr3,*target* equivalent to **bc 4,14,***target*
3. Branch to an absolute target if CR4 specifies greater than condition, setting the LR. This is a form of conditional call.
   bgtla cr4,*target* equivalent to **bcla 12,17,***target*
4. Same as (3), but target address is in the CTR.
   bgtctrl cr4 equivalent to **bcctrl 12,17**

B.4.6.2 Branch Simplified Mnemonics that Incorporate CR Conditions: Listings

Table B-19 shows simplified branch mnemonics and syntax for **bc** and **bca** without LR updating.

Table B-19. Simplified Mnemonics for bc and bca without Comparison Conditions or LR Update

Branch Semantics	bc	Simplified Mnemonic	bca	Simplified Mnemonic
Branch if less than	**bc 12,**BI[1],target	**blt cr**S target	**bca 12,**BI[1],target	**blta cr**S target
Branch if less than or equal	**bc 4,**BI[2],target	**ble cr**S target	**bca 4,**BI[2],target	**blea cr**S target
Branch if not greater than		**bng cr**S target		**bnga cr**S target
Branch if equal	**bc 12,**BI[3],target	**beq cr**S target	**bca 12,**BI[3],target	**beqa cr**S target
Branch if greater than or equal	**bc 4,**BI[1],target	**bge cr**S target	**bca 4,**BI[1],target	**bgea cr**S target
Branch if not less than		**bnl cr**S target		**bnla cr**S target
Branch if greater than	**bc 12,**BI[2],target	**bgt cr**S target	**bca 12,**BI[2],target	**bgta cr**S target
Branch if not equal	**bc 4,**BI[3],target	**bne cr**S target	**bca 4,**BI[3],target	**bnea cr**S target
Branch if summary overflow	**bc 12,**BI[4],target	**bso cr**S target	**bca 12,**BI[4],target	**bsoa cr**S target
Branch if unordered		**bun cr**S target		**buna cr**S target
Branch if not summary overflow	**bc 4,**BI[4],target	**bns cr**S target	**bca 4,**BI[4],target	**bnsa cr**S target
Branch if not unordered		**bnu cr**S target		**bnua cr**S target

[1] The value in the BI operand selects CR*n*[0], the LT bit.
[2] The value in the BI operand selects CR*n*[1], the GT bit.
[3] The value in the BI operand selects CR*n*[2], the EQ bit.
[4] The value in the BI operand selects CR*n*[3], the SO bit.

Table B-20 shows simplified branch mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table B-20. Simplified Mnemonics for bclr and bcctr without Comparison Conditions or LR Update

Branch Semantics	bclr	Simplified Mnemonic	bcctr	Simplified Mnemonic
Branch if less than	**bclr 12,**BI[1],target	**bltlr cr**S target	**bcctr 12,**BI[1],target	**bltctr cr**S target
Branch if less than or equal	**bclr 4,**BI[2],target	**blelr cr**S target	**bcctr 4,**BI[2],target	**blectr cr**S target
Branch if not greater than		**bnglr cr**S target		**bngctr cr**S target
Branch if equal	**bclr 12,**BI[3],target	**beqlr cr**S target	**bcctr 12,**BI[3],target	**beqctr cr**S target
Branch if greater than or equal	**bclr 4,**BI[1],target	**bgelr cr**S target	**bcctr 4,**BI[1],target	**bgectr cr**S target
Branch if not less than		**bnllr cr**S target		**bnlctr cr**S target
Branch if greater than	**bclr 12,**BI[2],target	**bgtlr cr**S target	**bcctr 12,**BI[2],target	**bgtctr cr**S target
Branch if not equal	**bclr 4,**BI[3],target	**bnelr cr**S target	**bcctr 4,**BI[3],target	**bnectr cr**S target
Branch if summary overflow	**bclr 12,**BI[4],target	**bsolr cr**S target	**bcctr 12,**BI[4],target	**bsoctr cr**S target
Branch if not summary overflow	**bclr 4,**BI[4],target	**bnslr cr**S target	**bcctr 4,**BI[4],target	**bnsctr cr**S target

1 The value in the BI operand selects CR*n*[0], the LT bit.
2 The value in the BI operand selects CR*n*[1], the GT bit.
3 The value in the BI operand selects CR*n*[2], the EQ bit.
4 The value in the BI operand selects CR*n*[3], the SO bit.

Table B-21 shows simplified branch mnemonics and syntax for **bcl** and **bcla**.

Table B-21. Simplified Mnemonics for bcl and bcla with Comparison Conditions and LR Update

Branch Semantics	bcl	Simplified Mnemonic	bcla	Simplified Mnemonic
Branch if less than	**bcl 12,**BI[1],target	**bltl cr**S target	**bcla 12,**BI[1],target	**bltla cr**S target
Branch if less than or equal	**bcl 4,**BI[2],target	**blel cr**S target	**bcla 4,**BI[2],target	**blela cr**S target
Branch if not greater than		**bngl cr**S target		**bngla cr**S target
Branch if equal	**bcl 12,**BI[3],target	**beql cr**S target	**bcla 12,**BI[3],target	**beqla cr**S target
Branch if greater than or equal	**bcl 4,**BI[1],target	**bgel cr**S target	**bcla 4,**BI[1],target	**bgela cr**S target
Branch if not less than		**bnll cr**S target		**bnlla cr**S target
Branch if greater than	**bcl 12,**BI[2],target	**bgtl cr**S target	**bcla 12,**BI[2],target	**bgtla cr**S target
Branch if not equal	**bcl 4,**BI[3],target	**bnel cr**S target	**bcla 4,**BI[3],target	**bnela cr**S target
Branch if summary overflow	**bcl 12,**BI[4],target	**bsol cr**S target	**bcla 12,**BI[4],target	**bsola cr**S target
Branch if not summary overflow	**bcl 4,**BI[4],target	**bnsl cr**S target	**bcla 4,**BI[4],target	**bnsla cr**S target

1 The value in the BI operand selects CR*n*[0], the LT bit.
2 The value in the BI operand selects CR*n*[1], the GT bit.
3 The value in the BI operand selects CR*n*[2], the EQ bit.
4 The value in the BI operand selects CR*n*[3], the SO bit.

Table B-22 shows the simplified branch mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table B-22. Simplified Mnemonics for bclrl and bcctrl with Comparison Conditions and LR Update

Branch Semantics	bclrl	Simplified Mnemonic	bcctrl	Simplified Mnemonic
Branch if less than	**bclrl 12,**BI[1],target	**bltlrl cr**S target	**bcctrl 12,**BI[1],target	**bltctrl cr**S target
Branch if less than or equal	**bclrl 4,**BI[2],target	**blelrl cr**S target	**bcctrl 4,**BI[2],target	**blectrl cr**S target
Branch if not greater than		**bnglrl cr**S target		**bngctrl cr**S target
Branch if equal	**bclrl 12,**BI[3],target	**beqlrl cr**S target	**bcctrl 12,**BI[3],target	**beqctrl cr**S target
Branch if greater than or equal	**bclrl 4,**BI[1],target	**bgelrl cr**S target	**bcctrl 4,**BI[1],target	**bgectrl cr**S target
Branch if not less than		**bnllrl cr**S target		**bnlctrl cr**S target
Branch if greater than	**bclrl 12,**BI[2],target	**bgtlrl cr**S target	**bcctrl 12,**BI[2],target	**bgtctrl cr**S target
Branch if not equal	**bclrl 4,**BI[3],target	**bnelrl cr**S target	**bcctrl 4,**BI[3],target	**bnectrl cr**S target
Branch if summary overflow	**bclrl 12,**B[4],target	**bsolrl cr**S target	**bcctrl 12,**BI[4],target	**bsoctrl cr**S target
Branch if not summary overflow	**bclrl 4,**BI[4],target	**bnslrl cr**S target	**bcctrl 4,**BI[4],target	**bnsctrl cr**S target

1 The value in the BI operand selects CR*n*[0], the LT bit.
2 The value in the BI operand selects CR*n*[1], the GT bit.
3 The value in the BI operand selects CR*n*[2], the EQ bit.
4 The value in the BI operand selects CR*n*[3], the SO bit.

B.5 Compare Word Simplified Mnemonics

In compare word instructions, the L operand indicates a word (L = 0) or a doubleword (L = 1). Simplified mnemonics in Table B-23 eliminate the L operand for word comparisons.

Table B-23. Word Compare Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Compare Word Immediate	**cmpwi cr**D,**r**A,SIMM	**cmpi cr**D,**0,r**A,SIMM
Compare Word	**cmpw cr**D,**r**A,**r**B	**cmp cr**D,**0,r**A,**r**B
Compare Logical Word Immediate	**cmplwi cr**D,**r**A,UIMM	**cmpli cr**D,**0,r**A,UIMM
Compare Logical Word	**cmplw cr**D,**r**A,**r**B	**cmpl cr**D,**0,r**A,**r**B

As with branch mnemonics, the **cr**D field of a compare instruction can be omitted if CR0 is used, as shown in examples 1 and 3 below. Otherwise, the target CR field must be specified as the first operand. The following examples use word compare mnemonics:

1. Compare **r**A with immediate value 100 as signed 32-bit integers and place result in CR0.
   cmpwi rA,**100** equivalent to **cmpi 0,0,r**A,**100**
2. Same as (1), but place results in CR4.
   cmpwi cr4,rA,**100** equivalent to **cmpi 4,0,r**A,**100**
3. Compare **r**A and **r**B as unsigned 32-bit integers and place result in CR0.
   cmplw rA,**r**B equivalent to **cmpl 0,0,r**A,**r**B

B.6 Compare Doubledoublewordword Simplified Mnemonics

In compare doubleword instructions, the L operand indicates a word (L = 0) or a doubleword (L = 1). Simplified mnemonics in Table B-23 eliminate the L operand for doubleword comparisons.

Table B-24. Doubleword Compare Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Compare Doubleword Immediate	**cmpdi** **cr**D,**r**A,SIMM	**cmpi** **cr**D,**1**,**r**A,SIMM
Compare Doubleword	**cmpd** **cr**D,**r**A,**r**B	**cmp** **cr**D,**1**,**r**A,**r**B
Compare Logical Doubleword Immediate	**cmpldi** **cr**D,**r**A,UIMM	**cmpli** **cr**D,**1**,**r**A,UIMM
Compare Logical Doubleword	**cmpld** **cr**D,**r**A,**r**B	**cmpl** **cr**D,**1**,**r**A,**r**B

As with branch mnemonics, the **cr**D field of a compare instruction can be omitted if CR0 is used, as shown in examples 1 and 3 below. Otherwise, the target CR field must be specified as the first operand. The following examples use word compare mnemonics:

1. Compare **r**A with immediate value 100 as signed 64-bit integers and place result in CR0.
   cmpdi rA,100 equivalent to **cmpi 0,1,rA,100**
2. Same as (1), but place results in CR4.
   cmpdi cr4,rA,100 equivalent to **cmpi 4,1,rA,100**
3. Compare **r**A and **r**B as unsigned 64-bit integers and place result in CR0.
   cmpld rA,rB equivalent to **cmpl 0,1,rA,rB**

B.7 Condition Register Logical Simplified Mnemonics

The CR logical instructions, shown in Table B-25, can be used to set, clear, copy, or invert a given CR bit. Simplified mnemonics allow these operations to be coded easily. Note that the symbols defined in Table B-8 can be used to identify the CR bit.

Table B-25. Condition Register Logical Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to
Condition register set	**crset** **b**x	**creqv** **b**x,**b**x,**b**x
Condition register clear	**crclr** **b**x	**crxor** **b**x,**b**x,**b**x
Condition register move	**crmove** **b**x,**b**y	**cror** **b**x,**b**y,**b**y
Condition register not	**crnot** **b**x,**b**y	**crnor** **b**x,**b**y,**b**y

Examples using the CR logical mnemonics follow:

1. Set CR[57].
   crset 25 equivalent to **creqv 25,25,25**
2. Clear CR0[SO].
   crclr so equivalent to **crxor 3,3,3**
3. Same as (2), but clear CR3[SO].
   crclr 4 * cr3 + so equivalent to **crxor 15,15,15**

4. Invert the CR0[EQ].**crnot eq,eq** equivalent to **crnor 2,2,2**
5. Same as (4), but CR4[EQ] is inverted and the result is placed into CR5[EQ].
   crnot 4 * cr5 + eq, 4 * cr4 + eq equivalent to **crnor 22,18,18**

B.8 Trap Instructions Simplified Mnemonics

The codes in Table B-26 are for the most common combinations of trap conditions.

Table B-26. Standard Codes for Trap Instructions

Code	Description	TO Encoding	<	>	=	<U[1]	>U [2]
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
—	Unconditional	31	1	1	1	1	1

1 The symbol ‘<U’ indicates an unsigned less-than evaluation is performed.
2 The symbol ‘>U’ indicates an unsigned greater-than evaluation is performed.

The mnemonics in Table B-27 are variations of trap instructions, with the most useful TO values represented in the mnemonic rather than specified as a numeric operand.

Table B-27. Trap Simplified Mnemonics

Trap Semantics	32-Bit Comparison	
	twi Immediate	tw Register
Trap unconditionally	—	**trap**
Trap if less than	**twlti**	**twlt**
Trap if less than or equal	**twlei**	**twle**
Trap if equal	**tweqi**	**tweq**
Trap if greater than or equal	**twgei**	**twge**

Table B-27. Trap Simplified Mnemonics (continued)

Trap Semantics	32-Bit Comparison	
	twi Immediate	tw Register
Trap if greater than	**twgti**	**twgt**
Trap if not less than	**twnli**	**twnl**
Trap if not equal	**twnei**	**twne**
Trap if not greater than	**twngi**	**twng**
Trap if logically less than	**twllti**	**twllt**
Trap if logically less than or equal	**twllei**	**twlle**
Trap if logically greater than or equal	**twlgei**	**twlge**
Trap if logically greater than	**twlgti**	**twlgt**
Trap if logically not less than	**twlnli**	**twlnl**
Trap if logically not greater than	**twlngi**	**twlng**

The following examples use the simplified trap mnemonics:

1. Trap if **r**A is not zero.
   twnei rA**,0** equivalent to **twi 24,r**A**,0**
2. Trap if **r**A is not equal to **r**B.
   twne rA**, r**B equivalent to **tw 24,r**A**,r**B
3. Trap if **r**A is logically greater than 0x7FF.
   twlgti rA**,** 0x7FF equivalent to **twi 1,r**A**,** 0x7FF
4. Trap unconditionally.
   trap equivalent to **tw 31,0,0**

Trap instructions evaluate a trap condition as follows: The contents of **r**A are compared with either the sign-extended SIMM field or the contents of **r**B, depending on the trap instruction.

The comparison results in five conditions that are ANDed with operand TO. If the result is not 0, the trap exception handler is invoked. See Table B-28 for these conditions.

Table B-28. TO Operand Bit Encoding

TO Bit	ANDed with Condition
0	Less than, using signed comparison
1	Greater than, using signed comparison
2	Equal
3	Less than, using unsigned comparison
4	Greater than, using unsigned comparison

B.9 Simplified Mnemonics for Accessing SPRs

The **mtspr** and **mfspr** instructions specify a special-purpose register (SPR) as a numeric operand. Simplified mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as a numeric operand. The pattern for **mtspr** and **mfspr** simplified mnemonics is straightforward: replace the **-spr** portion of the mnemonic with the abbreviation for the spr (for example XER, SRR0, or LR), eliminate the SPRN operand, leaving the source or destination GPR operand, **r**S or **r**D.

Following are examples using the SPR simplified mnemonics:

1. Copy the contents of **r**S to the XER.
   mtxer rS equivalent to **mtspr 1,r**S
2. Copy the contents of the LR to **r**D.
   mflr rD equivalent to **mfspr r**D**,8**
3. Copy the contents of **r**S to the CTR.
   mtctr rS equivalent to **mtspr 9,r**S

The architecture describes extended mnemonics for accessing CTR, LR, and XER only. However, some assemblers support other SPRs in the same fashion as shown in the following examples:

1. Copy the contents of **r**S to CSRR1.
   mtcsrr1 rS equivalent to **mtspr 59,r**S
2. Copy the contents of IVOR0 to **r**D.
   mfivor0 rD equivalent to **mfspr r**D**,400**
3. Copy the contents of **r**S to the SRR0.
   mtsrr0 rS equivalent to **mtspr 26,r**S

There is an additional simplified mnemonic convention for accessing SPRGs. These are shown in Table B-29 along with the equivalent simplified mnemonic using the formula described above.

Table B-29. Additional Simplified Mnemonics for Accessing SPRGs

SPR	Move to SPR		Move from SPR	
	Simplified Mnemonic	Equivalent to	Simplified Mnemonic	Equivalent to
SPRGs	**mtsprg** *n*, **r**S	**mtspr** 272 + *n*,**r**S	**mfsprg r**D, *n*	**mfspr r**D,272 + *n*
	mtsprg*n* **r**S		**mfsprg***n* **r**D	

B.10 Recommended Simplified Mnemonics

This section describes commonly-used operations (such as NOP, load immediate, load address, move register, and complement register).

B.10.1 NOP (nop)

Many instructions can be coded so that, effectively, no operation is performed. A mnemonic is provided for the preferred form of NOP. If an implementation performs any type of run-time optimization related to NOPs, the preferred form is the following:

nop equivalent to **ori 0,0,0**

B.10.2 Load Immediate (li)

The **addi** and **addis** instructions can be used to load an immediate value into a register. Additional mnemonics are provided to convey the idea that no addition is being performed but that data is being moved from the immediate operand of the instruction to a register.

1. Load a 16-bit signed immediate value into **r**D.
   li rD**,**value equivalent to **addi r**D**,0,**value
2. Load a 16-bit signed immediate value, shifted left by 16 bits, into **r**D.
   lis rD**,**value equivalent to **addis r**D**,0,**value

B.10.3 Load Address (la)

This mnemonic permits computing the value of a base-displacement operand, using the **addi** instruction that normally requires a separate register and immediate operands.

la rD**,**d(**r**A) equivalent to **addi r**D**,r**A**,**d

The **la** mnemonic is useful for obtaining the address of a variable specified by name, allowing the assembler to supply the base register number and compute the displacement. If the variable *v* is located at offset d*v* bytes from the address in **r***v*, and the assembler has been told to use **r***v* as a base for references to the data structure containing *v*, the following line causes the address of *v* to be loaded into **r**D:

la rD**,***v* equivalent to **addi r**D**,r***v***,**d*v*

B.10.4 Move Register (mr)

Several instructions can be coded to copy the contents of one register to another. A simplified mnemonic is provided that signifies that no computation is being performed, but merely that data is being moved from one register to another.

The following instruction copies the contents of **r**S into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

mr rA**,r**S equivalent to **or r**A**,r**S**,r**S

B.10.5 Complement Register (not)

Several instructions can be coded in a way that they complement the contents of one register and place the result into another register. A simplified mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of **r**S and places the result into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

not rA,**r**S	equivalent to	**nor r**A,**r**S,**r**S

B.10.6 Move to Condition Register (mtcr)

This mnemonic permits copying the contents of a GPR to the CR, using the same syntax as the **mfcr** instruction.

mtcr rS	equivalent to	**mtcrf** 0xFF,**r**S

B.10.7 Sync (sync)

The **sync** extended mnemonics provide simpler mnemonics for specifying certain **sync** operations:

Lightweight sync		
lwsync	equivalent to	**sync 1**
Heavyweight sync		
hwsync	equivalent to	**sync 0**
Book E / PowerPC compatibility		
sync	equivalent to	**sync 0**
msync	equivalent to	**sync 0**

B.10.8 Integer Select (isel)

The following mnemonics simplify the most common variants of the **isel** instruction that access CR0:

Integer Select Less Than		
isellt rD,**r**A,**r**B	equivalent to	**isel r**D,**r**A,**r**B,**0**
Integer Select Greater Than		
iselgt rD,**r**A,**r**B	equivalent to	**isel r**D,**r**A,**r**B,**1**
Integer Select Equal		
iseleq rD,**r**A,**r**B	equivalent to	**isel r**D,**r**A,**r**B,**2**

B.10.9 TLB Invalidate Local Indexed

The following simplified mnemonics are provided for **tlbilx** encodings:

tlbilxlpid	equivalent to	**tlbilx** 0,0
tlbilxpid	equivalent to	**tlbilx** 1,0,0
tlbilxva rA,**r**B	equivalent to	**tlbilx** 3,**r**A,**r**B
tlbilxva rB	equivalent to	**tlbilx 3,0,r**B