e6500 Core Reference Manual

Supports
e6500

E6500RM
Rev 0
06/2014

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: E6500RM
Rev 0, 06/2014

Contents

Paragraph Number	Title	Page Number

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Paragraph Number	Title	Page Number

Contents

Paragraph Number | **Title** | **Page Number**

Chapter 6
Memory Management Units (MMUs)

Contents

Contents

Paragraph Number	Title	Page Number

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Paragraph Number | Title | Page Number

Figures

Figures

Figure Number **Title** **Page Number**

Figures

Figures

Tables

Tables

Tables

Tables

Tables

Tables

Tables

Tables

About this book

This core reference manual includes the register model, instruction model, MMU, memory subsystem, and debug and performance monitor facilities of the e6500 core. The primary objective of this manual is to describe the functionality of the e6500 embedded microprocessor core for software and hardware developers. This manual is intended as a companion to the following documents:

- *EREF: A Programmer's Reference Manual for Freescale Power Architecture Processors* (hereafter called *EREF*),
- *AltiVec Technology Programming Environments Manual for Power ISA Processors*, and
- *Power ISA™ Version 2.06*.

These documents describe the architecture to which the e6500 core is implemented and referenced frequently. This manual focuses on features that are specific to the e6500 microprocessor.

Information in this manual is subject to change without notice, as described in the disclaimers on the title page of this manual. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation. Updates to this document and errata can be found at freescale.com.

Audience

It is assumed that the reader understands operating systems, microprocessor system design, and the basic principles of RISC processing. It is also assumed the reader has access to *EREF*, *AltiVec Technology Programming Environments Manual for Power ISA Processors*, and *Power ISA™ Version 2.06* (hereafter called Power ISA).

Organization

The following is a summary and a brief description of the major sections of this manual:

- Chapter 1, "e6500 Overview," provides a general description of e6500 functionality.
- Chapter 2, "Register Model," is useful for software engineers who need a general understanding of the e6500 register set and details of e6500-specific features.
- Chapter 3, "Instruction Model," provides a general overview of the addressing modes and a description of the instructions as defined by the architecture and indicates particular areas in which the e6500 provides implementation-specific details not described in the *EREF*. Instructions are organized by function.
- Chapter 4, "Interrupts and Exceptions," describes how the e6500 core implements the interrupt model.

- Chapter 5, "Core Caches and Memory Subsystem," describes the e6500 L1 cache, the shared L2 cache, and the memory subsystem (MSS).
- Chapter 6, "Memory Management Units (MMUs)," describes e6500 memory management, including the mechanisms and structures associated with address translation.
- Chapter 7, "Timer Facilities," describes timers provided by the e6500 core.
- Chapter 8, "Power Management," describes power management facilities provided by the e6500 core.
- Chapter 9, "Debug and Performance Monitor Facilities," describes the debug and performance monitor facilities implemented in the e6500 core.
- Chapter 10, "Execution Timing," describes how instructions are fetched, decoded, issued, executed, and completed and how instruction results are presented to the processor and memory system. This chapter also provides tables that indicate latency and repeat rate for each of the instructions supported by the e6500 core.
- Chapter 11, "Core and Cluster Software Initialization Requirements," describes the software initialization requirements after reset.
- Appendix A, "Simplified Mnemonics," describes extended mnemonics for assembly language programming.

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 on Power.org:

- *Power ISA™ Version 2.06*, January 30, 2009

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*—This book provides a higher-level view of the programming model as it is defined by Power ISA and Freescale implementation standards.
- *AltiVec Technology Programming Environments Manual for Power ISA Processors*—This book provides a higher-level view of the programming model for the vector processing provided by AltiVec technology as it is defined by 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 e6500 core.
- 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.
- Data sheets—Data sheets provide specific data regarding bus timing, signal behavior, and DC, AC, and thermal characteristics, as well as other design considerations.
- Product briefs—Each device has a product brief that provides an overview of its features. This document is roughly equivalent to the Overview chapter (Chapter 1) of an implementation's user manual.
- Application notes—These 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, visit freescale.com.

Conventions

This manual uses the following notational conventions:

cleared/set	When a bit takes the value zero, it is said to be cleared; when it takes the value 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, $\overline{\textit{qual BG}}$.
0x0	Prefix used to denote a hexadecimal number.
0b0	Prefix used to denote a 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.
vA, **v**B, vC, **v**S	Instruction syntax used to identify a source VR.
vD	Instruction syntax used to identify a destination VR.
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 a numeric variable.
¬	NOT logical operator.
&	AND logical operator.
|	OR logical operator.
[—]	Indicates reserved bits or fields in a register. Although these bits can be written to as ones or zeros, they are always read as zeros.

Terminology conventions

This table 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
e6500 Overview

This chapter provides a general overview of the e6500 microprocessor core. It includes the following:

- An overview of architecture features as implemented on the e6500 core
- Summaries of the core feature set and instruction pipeline and flow
- Overviews of the programming model, interrupts, and exception handling
- A description of the memory management architecture
- High-level details of the e6500 core memory and coherency model
- A brief description of the CoreNet interface
- A list of differences between different versions of the e500mc/e5500/e6500 cores from e500v2

The e6500 core provides features that the integrated device may not implement or may implement in a more specific way. Differences are summarized in the integrated device's documentation.

1.1 Overview

The e6500 core is a low-power, 64-bit, multi-threaded implementation of the resources for embedded processors defined by Power ISA. The core supports the simultaneous execution of two threads (processors). The core implements two sets of thirty two 64-bit general-purpose registers; however, it supports accesses to 40-bit physical addresses. The block diagram in Figure 1-1 shows how the e6500 core's functional units operate independently and in parallel. Note that this conceptual diagram does not attempt to show how these features are implemented physically.

The e6500 core is a multi-threaded superscalar processor that can decode two instructions and complete two instructions per thread per clock cycle. Instructions complete in order, but can execute out of order. Execution results are available to subsequent instructions in the same thread through the rename buffers, but those results are recorded into architected registers in program order, maintaining a precise exception model.

The processor core integrates the following execution units:

- Four simple instruction units (SFX0 and SFX1 per thread)
- One multiple-cycle instruction unit (MU)
- Two branch units (BU, one per thread)
- One floating-point unit (FPU)
- One AltiVec unit (VSFX, VCFX, VFPU, VPERM)
- Two load/store units (LSUs, one per thread).
  - — The LSUs support 64-bit integer and floating-point operands and 128-bit vector operands for AltiVec operations.

The ability to execute 12 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 e6500 core includes hardware managed, first-level instruction and data memory management units (MMUs) and a software managed, on-chip, second-level unified MMU with hardware assisted tablewalk.

The first-level MMUs for both instruction and data translation are each composed of two subarrays:

- An eight-entry fully associative array of translation look-aside buffer (TLB) entries for variable-sized pages; and
- A 64-entry, four-way set-associative array of TLB entries for fixed-size pages that provide virtual to physical memory address translation for variable-sized pages and demand-paged fixed pages, respectively.

NOTE

These TLB arrays are maintained entirely by the hardware with a true least-recently-used (LRU) algorithm and are a cache of the second-level MMU. If a second thread is active, the first-level instruction MMU is shared between threads, but each thread has a dedicated first-level data MMU.

The second-level MMU contains a 64-entry fully associative, unified (instruction and data) TLB that provides support for variable-sized pages. It also contains a 1024-entry, eight-way set-associative, unified TLB for 4 KB page size support. These second-level TLBs are maintained by both hardware and software. The software can enable a hardware tablewalk mechanism to automatically find a page table entry and load a 4 KB TLB entry into the TLB.

The e6500 core includes independent, 32 KB, eight-way set-associative, physically addressed L1 caches for instructions and data. It also includes a unified 2048 KB, 16-way set-associative, physically addressed backside L2 cache. The L1 caches are shared between active threads. The L2 cache is shared between cores in a cluster. Depending on the integrated device, the cluster may contain from one to four e6500 cores, all of which share the backside L2 cache and the interface to the rest of the memory subsystem.

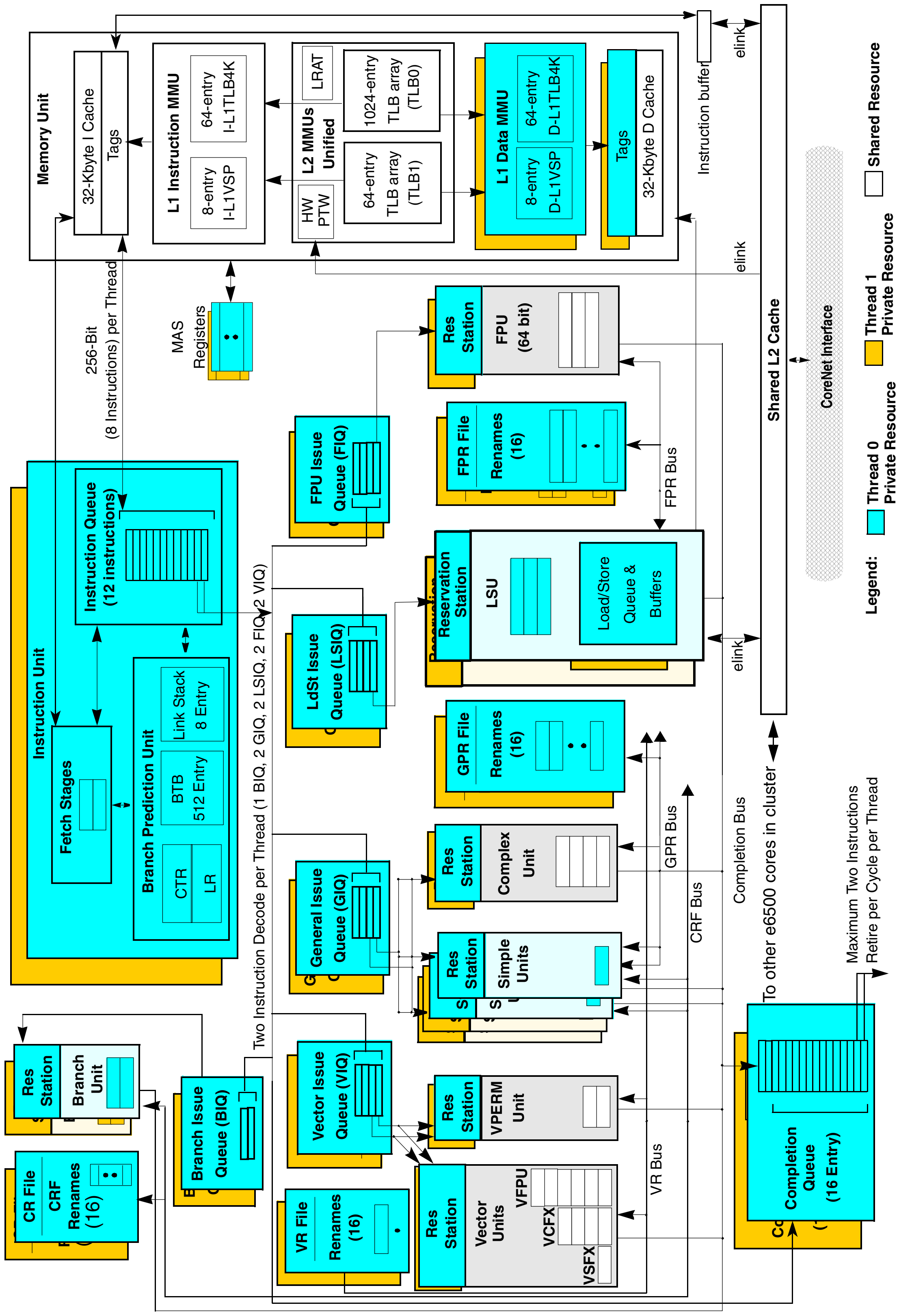


Figure 1-1. e6500 block diagram

Cache lines in the e6500 core 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 e6500 core 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, as shown in the following figure.

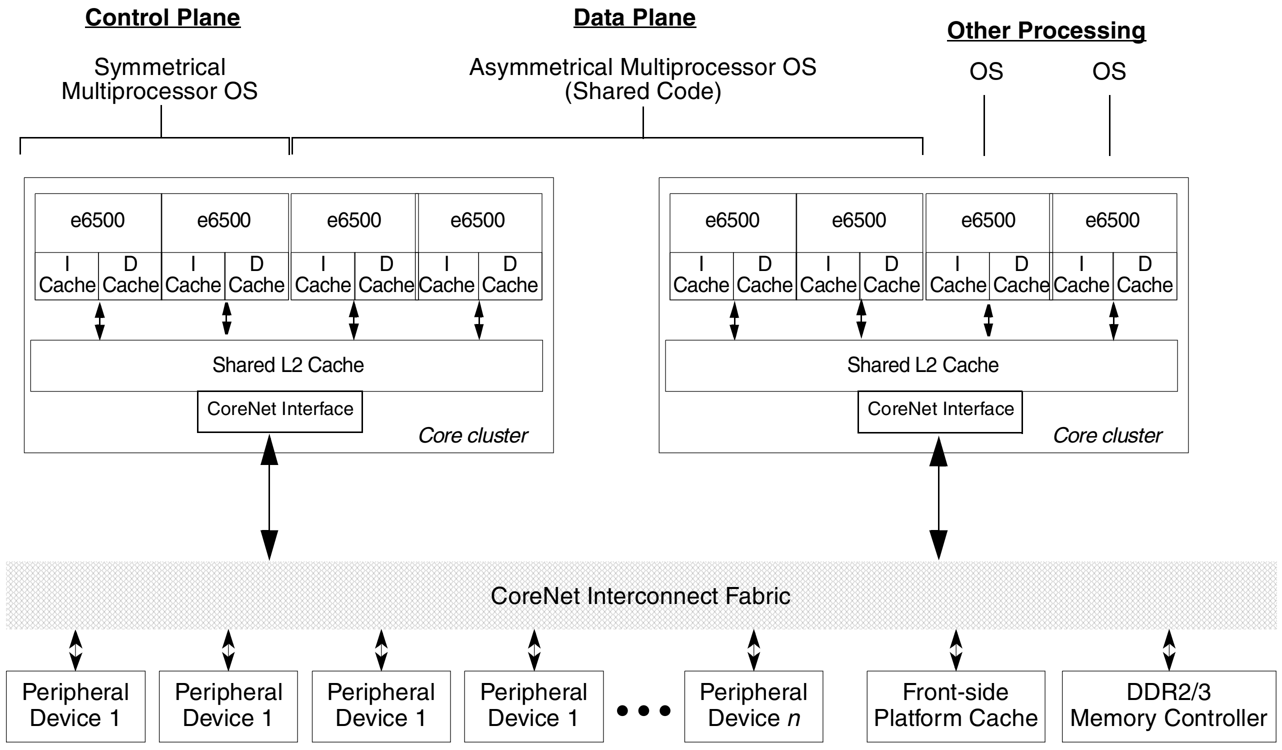


Figure 1-2. Example partitioning scenario of a multicore integrated device

The CoreNet interface provides the primary on-chip interface between the core cluster 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.

The architecture defines the resources required to allow orderly and secure interactions between thread processors, the cores, memory, peripheral devices, and virtual machines. These include 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, the e6500 core implements the following categories as defined in *EREF*:

- Base
- Embedded (E)
- Alternate Time Base (ATB)

- Decorated Storage (DS)
- Embedded.Enhanced Debug (E.ED)
- Embedded.External PID (E.PID)
- Embedded.Hypervisor (E.HV)
- Embedded.Hypervisor.LRAT (E.HV.LRAT)
- Embedded.Page Table (E.PT)
- Embedded.Little-Endian (E.LE)
- Embedded.Multi-Threading (E.EM)
- Embedded.Performance Monitor (E.PM)
- Embedded.Processor Control (E.PC)
- Embedded.Cache Locking (E.CL)
- External Proxy (EXP)
- Floating Point and Floating Point.Record (FP, FP.R)
- Vector (V)
- Wait (WT)
- 64-Bit (64)
- Data Cache Extended Operations (DEO)
- Enhanced Reservations (ER)
- Cache Stashing (CS)

The above categories define instructions, registers, and processor behavior associated with a given category. For a more complete and canonical definition of the e6500 core register and instruction set, see Chapter 2, "Register Model" and Chapter 3, "Instruction Model," respectively.

Some categories defined by Power ISA are included as a part of *EREF* and are not specified by categories in *EREF*. Such categories include: Cache Specification, Store Conditional Page Mobility, and Memory Coherence. In addition, *EREF* or the e6500 core may implement a subset of a category or provide extra implementation-dependent capabilities. Such distinctions are described in this manual.

1.2 Feature summary

Key features of the e6500 core are summarized as follows:

- 64-bit architecture implementation with 40-bit physical addressing
- Thirty two 64-bit General Purpose Registers (GPR) per thread
- Thirty two 64-bit Floating-Point Registers (FPR) per thread
- FPR-based floating-point binary compatible with e300, e600, e500mc, and e5500 cores
- Thirty two 128-bit Vector Registers (VR) per thread
- Enhanced branch prediction
  - 128 sets of four-way associative branch target and local history buffers per thread
  - Global history indexed 2048 entry pattern history table per thread
  - Eight-entry link stack per thread with stack underflow reversion to the BTB provided target
- Multicore architecture support

- Hardware support for the hypervisor programming model to provide partitioning and virtualization
  - 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
- Shared L2 cache and interface to the CoreNet interconnect fabric
  - Cores are provided in clusters that share an L2 cache and CoreNet interface. These clusters reduce the amount of coherency traffic on the CoreNet interface and provide faster coherent transactions among cores in a cluster.

- 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, it allows high performance atomic “fire and forget” operations on memory locations performed directly by the targeted device.
- L1 cache features
  - Separate 32 KB, eight-way set-associative level 1 (L1) instruction and data caches
  - 64 eight-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 data parity checking
    - Full recoverability from single bit errors in data or tags because modified data is written through to the inclusive L2 cache
  - Two-cycle L1 cache array access and three-cycle load-to-use latency
  - FIFO replacement algorithm
  - Cache coherency
    - Supports valid and invalid states per active thread. Stores are written through to the shared L2 cache, which implements a full MESI protocol.
    - Provides snooping for invalidations coming from the shared L2 cache.
    - Accepts cache stashes, which allow devices in the integrated device to push to the cache information that may be requested in the future by the core, significantly reducing latency.
  - 64-byte (16-word) cache line, coherency granule size
  - Persistent cache line 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. Locking is persistent in that locks are not cleared until software explicitly unlocks them. Cache locking functionality can be separately enabled for user mode or supervisor mode.
- L2 cache features
  - Shared inline L2 cache
  - Four independent banks
  - Address-mapped accesses to banks
  - 2048 KB capacity
  - 16-way set-associative
  - Four cores per cluster sharing the same cache banks
  - Way-partitionable based on cores in cluster
  - Inclusive for data stored in cluster cores' L1 caches
  - Streaming Pseudo-LRU (SPLRU) replacement algorithm
  - Enhanced error detection and correction
    - ECC single-bit correction and double-bit detection on data, tags, and status
  - Cache coherency
    - Shared, modified, and exclusive data intervention so cache contents can be shared without requiring a memory update
    - Accepts cache stashes, which allow devices in the integrated device to push information to the cache that may be requested in the future by a core in the cluster, significantly reducing latency.
    - Snoop filtering for cores in the cluster
  - 64-byte (16-word) cache line, coherency-granule size
  - Persistent cache line locking
    - Allows instructions and data to be locked into the cache on a cache block basis.
    - Locking is performed by a set of touch and lock set instructions. Locking is persistent in that locks are not cleared until software explicitly unlocks them. Cache locking 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

    - 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 supervisor privilege levels
    - System call instruction generates a system call or a hypervisor-level system call (hypercall) interrupt. Executing **sc** or **sc 0** generates a system call, and **sc 1** generates a hypercall interrupt.
    - Doorbell interrupts allow one processor to signal an interrupt to another core (doorbell, doorbell critical, guest doorbell, guest doorbell critical, or guest doorbell machine check).
    - Certain interrupts, including external interrupts, MMU interrupts, and performance monitor interrupts, can be configured to be delivered directly to the guest-supervisor state or to the hypervisor state (default).
    - Embedded hypervisor privilege interrupt captures guest supervisor attempts to access hypervisor resources.
    - TLBs can be programmed to always force a DSI to generate a virtualization fault to the hypervisor state.
  - External interrupt proxy
    - Provides automatic hardware acknowledgement of external interrupts signaled by the programmable interrupt controller (PIC) on the integrated device, which increases responsiveness to external interrupts from peripheral devices and reduces interrupt latency. See Section 4.9.6.1, “External proxy.”
    - The automatic hardware acknowledgement replaces the “read IACK” step.
  - Non-maskable interrupt for soft-reset type capability
  - One set of interrupt signal pins from the integrated device interrupt controller for each thread
- Memory management unit (MMU)
  - 64-bit effective address to 40-bit physical address translation
  - Virtual address fields in TLB entries
    - GS field indicates whether the access is hypervisor or guest address space (also indicates hypervisor or guest privilege).
    - AS field indicates one of two address spaces (from IS or DS in the machine state register).
    - 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.
  - External 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-sized (4 KB to 1 TB) and fixed-size (4 KB) pages
  - Data L1 MMU, per thread
    - Eight-entry, fully-associative TLB array for variable-sized pages
    - 64-entry, four-way set-associative TLB for 4 KB pages

- Instruction L1 MMU, shared between threads
  - Eight-entry, fully-associative TLB array for variable-sized pages
  - 64-entry, four-way set-associative TLB for 4 KB pages
- Unified L2 MMU, shared between threads
  - 64-entry, fully-associative TLB array (TLB1) for variable-sized pages
  - 1024-entry, eight-way set-associative, unified (for instruction and data accesses) TLB array (TLB0) that supports only 4 KB pages with single-bit error detection and auto correction through hardware invalidation
- Logical to real address translation (LRAT) structure to allow guest supervisor to securely write TLB entries without hypervisor intervention
  - Eight-entry, fully associative
  - Supports power-of-two variable page sizes
- Hardware assisted reload for TLB0 from a page table in memory
- 14-bit process ID (PID) supporting 16 K simultaneous contexts without TLB flushing
- Real memory support for as much as 1 TB (2^{40})
- 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 caches, decoded instruction types, 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**).
- Six performance monitor counters can be programmed from any defined event.

• Power management
- Low-power design
- Multiple power-saving modes
  - Static power reduction when core is not busy with fast return to normal operation
  - Hardware and software controlled entry and exit for low power states
  - Ability to wait in low power states until an interrupt or a store to a specified address occurs
- Dynamic power management
- Capability to power off AltiVec unit when not in use, further reducing static power

• Testability
- MUXD scan design
- Debug Notify Interrupt (**dni**) instruction provides a debug breakpoint interrupt when executed and the debugging is enabled, otherwise produces a no-op.

1.3 Instruction flow

The e6500 core is a pipelined, multi-threaded, superscalar processor with parallel execution units that allow instructions to execute out of order but record their results in order.

As a multi-threaded processor, the e6500 core appears as two processors to user and guest operating system software. The two processor threads share some resources, such as execution units and caches, but not others, such as the architected user-level state. As with pipelining, the execution of two software threads on one processor increases the overall system throughput.

As a superscalar processor, the e6500 core issues multiple independent instructions into separate execution units in a single cycle, allowing parallel execution. The e6500 core has ten execution unit types:

- Branch (BU)
- Load/store (LSU)
- Floating-point (FPU)
- Vector (VSFX, VCFX, VFPU, VPERM)
- Complex integer (CFX)
- Simple arithmetic (SFX0 and SFX1).

Each thread has dedicated branch, load/store, and simple arithmetic execution units, but all threads share the floating-point, vector, and complex integer execution units.

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 may finish executing before the higher latency divide instruction. Most instructions immediately make results available to subsequent instructions, but cannot update the architected GPR specified as its target operand out of program order.

Pipelining breaks instruction processing into discrete stages, so multiple instructions in an instruction sequence can occupy successive stages: as an instruction completes one stage, it passes to the next, leaving the previous stage available to a subsequent instruction. Although it may take multiple cycles for an instruction to pass through all of the pipeline stages, once a pipeline is full, instruction throughput is increased.

The common pipeline stages are as follows:

- Instruction fetch stage—includes the clock cycles necessary for an active thread to request an instruction and the time the memory system takes to respond to the request. Instructions retrieved are latched into the thread’s instruction queue (IQ) for subsequent consideration by the dispatcher.

  Instruction fetch timing depends on many variables, such as whether an instruction is in the 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.

- Decode/dispatch stage—fully decodes each instruction. Most instructions are dispatched to the issue queues; however, **isync**, **rfi**, **rfgi**, **rfci**, **rfdi**, **rfmci**, **sc**, **ehpriv**, **nop**, and some other instructions do not go to issue queues.
- The issue queues, BIQ, GIQ, VIQ, LSIQ, and FIQ can accept one (BIQ) or two instructions (all other issue queues) 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.
  - — Space must be available in the completion queue (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.
- 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 each thread processor in the e6500 has 16 rename registers, one for each completion queue entry, so instructions cannot stall because of a shortage of rename registers.

  The behavior of the issue queues follows from how dispatch places instructions on the issue queues. For example, the GIQ operates as follows:
  - — The GIQ accepts as many as two instructions from the dispatch unit per cycle. Instructions to be executed in SFX0, SFX1, and CFX are dispatched to the GIQ, shown in Figure 1-3.

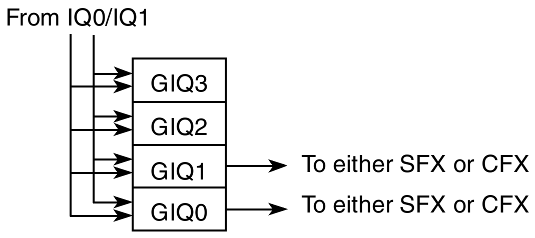


Figure 1-3. GPR Issue Queue (GIQ)

  - — Instructions can be issued out of order from the bottom two GIQ entries (GIQ1–GIQ0) to either SFX or CFX.

NOTE

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 an SFX need not wait for a CFX instruction in GIQ0 that is stalled behind a long-latency divide.

  Each thread has its own set of issue queues. Issue queues other than GIQ operate in a similar manner, each servicing specific execution units:
  - — LSIQ services the load/store unit (LSU), one LSIQ and one LSU per thread
  - — FIQ services the floating-point unit (FPU), one FIQ per thread and one shared FPU
  - — BIQ services the branch unit (BU), one BIQ and one BU per thread
  - — VIQ services the AltiVec execution units (VSFX, VCFX, VFPU, VPERM), one VIQ per thread and one set of shared execution units VSFX, VCFX, VFPU, VPERM

- Execution stage—accepts instructions from its issue queue when the appropriate reservation stations are not busy. In this stage, the operands assigned from the issue stage are latched.

  The execution units execute the instructions (perhaps over multiple cycles), write results on their result buses, and notify the CQ when the instructions finish. The execution units report any exceptions to the completion stage. Instruction-generated exceptions are not taken until the excepting instruction is next to retire.

  — The branch unit (BU) executes (resolves) all branch and CR logical instructions in the execution stage. If a branch is mispredicted, it takes at least five cycles for the next instruction to reach the execution stage.

  — The 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 the following figure, has these features:

    – Three-cycle load latency for most instructions except for AltiVec and floating-point load instructions, which take four cycles
    – Fully pipelined
    – Load-miss queue
    – Service for load hits when the load-miss queue is full
    – Up to eight load misses that can be pipelined in parallel while L1 cache hits continue to be serviced

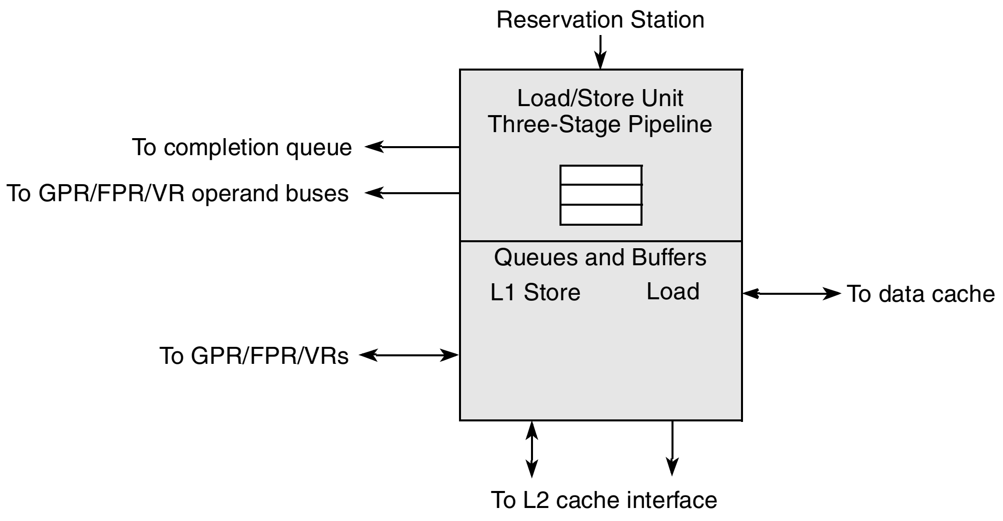


Figure 1-4. Three-stage load/store unit

- 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 in a thread 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 per thread can be retired per clock cycle. If no dependencies exist, as many as two instructions are retired in program order. Section 10.3.3, "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 e6500 core implements the registers and instructions as defined by the architecture (Power ISA and Freescale implementation standards) and are fully described in *EREF*. The following sections provide a high-level description and listing of the resources that are implemented on the e6500 core.

1.4.1 Register model overview

In general, registers on the e6500 core are implemented as defined by the architecture. Any e6500-specific differences from or extensions to the architecture are described in Chapter 2, "Register Model," of this manual.

The e6500 core implements the following types of registers:

- Registers that contain values specified by using operands that are part of the instruction syntax defined by *EREF*:
  - Thirty-two 64-bit general purpose registers (GPRs) per thread
    - **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, and **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) per thread
    - **fr**D indicates an FPR that is used as the destination or target of a floating-point computational or load instruction.
    - **fr**S indicates an FPR that is used as the source of a floating-point computational or store instruction.
    - **fr**A, **fr**B, and **fr**C indicate FPRs that are used to hold values that are operated upon for floating-point computational instructions.
  - Thirty-two 128-bit vector registers (VRs) per thread
    - **v**D indicates a VR that is used as the destination or target of a vector computational, logical, permute, or load instruction.
    - **v**S indicates a VR that is used as the source of a vector computational, logical, permute, or store instruction.

    - **v**A, **v**B, and **v**C indicate VRs that are used to hold values that are operated upon for vector computational, permute, or logical instructions.
- Registers that are updated automatically to record a condition that occurs as a by-product of a computation:
  - Condition (CR) register
    - Consists of eight 4-bit fields that record the results of certain operations that are typically used for testing and branching.
    - Can be accessed with special Move To and Move From instructions. See ***[ADD XREF HERE!]***
  - Integer Exception (XER) register
    - Records conditions, such as carries and overflows.
    - The XER is an SPR and can be accessed with Move To and Move From SPR instructions (**mtspr** and **mfspr**).
  - Floating-Point Status and Control (FPSCR) register
    - Records and controls exception conditions, such as overflows, controls the rounding mode.
    - Indicates the type of result for certain floating-point operations.
  - Vector Status and Control (VSCR) register
    - Records saturation exceptions.
    - Controls which mode vector floating-point operations are performed.
  - Machine State (MSR) register
    - MSR is a supervisor-level register; however, some fields can be written only by hypervisor-level software.
    - MSR is used to configure operational behavior, such as setting the privilege level and enabling asynchronous interrupts. When an interrupt is taken, certain MSR bits are stored into the appropriate save and restore register 1 (*x*SRR1) as determined by the interrupt type. The values in the *x*SRR1 are 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 and 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 **mtspr** and **mfspr** instructions and 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 2.2.2, “Special-purpose registers (SPRs).”
- Performance monitor registers (PMRs)
  - Configure and program the core-specific performance monitor.
  - PMRs are similar to SPRs in that they are accessed by Move To and Move From PMR instructions (**mtpmr** and **mfpmr**).

1.4.2 Instruction model overview

In general, instructions on the e6500 core are implemented as defined by the architecture. Any e6500-specific differences from or extensions to the architecture are described in Chapter 3, “Instruction Model,” of this manual.

Table 3-75 lists the instructions implemented in the e6500 core.

1.5 Summary of differences between previous e500 family cores

The following sections describe the changes between previous cores in the e500 family. These are high-level descriptions that are intended to explain the programming model changes.

1.5.1 Changes from e500v2 to e500mc

The e500mc core contains several differences from the e500v2 core. Significant programming model changes occur from:

- The removal of SPE (and the embedded floating-point functionality)
- The addition of FPR-based floating-point
- The addition of 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 uses any floating-point software must also be re-linked because the manner in which floating-point arguments are passed to functions is different. The floating-point model of the e500mc is compatible with the e300 and e600 cores and should provide a seamless transition when moving software from the e300 or the e600 to the e500mc.

A summary of the changes to the core is show in the following table. 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 to the e500mc.

Table 1-1. Summary of e500v2 and e500mc differences

Feature	e500v2	e500mc	Notes
Backside L2 cache	Not present	Present	An integrated backside L2 cache is present in e500mc.
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, rather than 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.

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

Feature	e500v2	e500mc	Notes
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.	SoC registers now almost completely control how power management functions are invoked. See Chapter 8, "Power Management."
External proxy	Not present	Present	External proxy is a mechanism that 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 4.9.6.1, "External proxy."
Additional interrupt level for Debug interrupts	Not present	Present	A separate interrupt level for Debug interrupts and the associated save/restore registers (DSRR0/DSRR1) are provided. 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 3.4.12.5, "Message Clear and Message Send instructions."
External PID load/store	Not present	Present	Instructions are provided for supervisor- and hypervisor-level software to perform load and store operations using a different address space context. See Section 3.4.12.3, "External PID load and store instructions."
Decorated storage	Not present	Present	Instructions are provided for performing load and store operations to devices that include metadata that is interpreted by the target address. Devices in some SoCs utilize this facility for performing atomic memory updates, such increments and decrements. See Section 3.4.3.2.10, "Decorated load and store instructions."
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 3.4.11.1, "User-level cache instructions," and Section 5.5.5, "Load/store operation ordering."
CoreNet	Uses CCB as an 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."

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

Feature	e500v2	e500mc	Notes
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 interrupts. The ways that errors are reported are more conducive in 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-through cache."
Cache block size	32 bytes	64 bytes	The e500mc core contains a larger cache block/line/coherency granule size.
Number of variable size TLB entries	16	64	The e500mc contains a larger number of variable-size TLB entries and a larger number of available page sizes. See Section 6.3.2, "L2 TLB arrays."

1.5.2 Changes from e500mc to e5500

The e5500 core contains several differences from the e500mc core. The programming model of the e5500 is compatible with the e500mc in the user, guest supervisor, and hypervisor modes with the exception that a few hypervisor-level resources that could be read in the e500mc core can no longer be read in the e5500 core (in particular, some registers that are architecturally write-only). All software written for the e500mc core should run unmodified on the e5500 core.

The e5500 core is a 64-bit implementation and adds the 64-bit mode, as well as several 64-bit instructions. It also supports 32-bit mode or running 32-bit software on a supervisor or hypervisor that is 64 bits.

A summary of the changes to the core is show in the following table. This table is intended to be a general summary and not an explicit list of differences.

Table 1-2. Summary of e500mc and e5500 differences

Feature	e500mc	e5500	Notes
64-bit execution	Not present (32-bit only)	Both 32-bit and 64-bit modes	The e5500 core provides 64-bit mode and several 64-bit instructions. Information about 64-bit features is discussed throughout this document.
L2 cache size	128 KB	512 KB	The e5500 core includes a larger L2 cache.
L2 cache latency (from L1 miss)	9	11	The e5500 core has two more cycles of latency.

Table 1-2. Summary of e500mc and e5500 differences

Feature	e500mc	e5500	Notes
Floating-point performance	Floating-point is not fully pipelined	Floating-point is fully pipelined	The e5500 core uses the same programming model for floating-point as the e500mc core, but provides a higher performance FPU that is fully pipelined. See Section 10.5, "Instruction latency summary."
mtspr/**mfspr** instruction execution unit	Executed in SFX0	Executed in CFX	**mtspr**/**mfspr**, as well as some other instructions that modify the architected state of registers, are executed in the CFX unit instead of the SFX0 unit.
Branch prediction	Uses BTB	Uses BTB, link stack, and STIC/STAC	The e5500 core improves branch prediction for function call and return. See Section 10.4.1.2, "Branch prediction and resolution."

1.5.3 Changes from e5500 to e6500

The programming model of the e6500 core is compatible with the e500mc and e5500 cores in the user, guest supervisor, and hypervisor modes. The only exceptions are:

- The addition of the AltiVec vector and thread management facilities
- The addition of some cache control
- The addition of cache locking operations

All software, with the exceptions of imprecise debug events and any cache control or cache locking software written for the e500mc and e5500 cores, should run unmodified on a e6500 processor.

A summary of the changes to the core is show in the following table. This table is intended to be a general summary and not an explicit list of differences.

Table 1-3. Summary of e5500 and e6500 differences

Feature	e5500	e6500	Notes
AltiVec vector registers and instructions	Not present	Present	The e6500 core adds AltiVec vector facility for improved performance using SIMD. The AltiVec facility is fully described in *EIS: Altivec Technology Implementation Standards for Power ISA Processors*.
Multi-threading	Not present	Present	The e6500 core adds multi-threaded execution capability for improved throughput.

Table 1-3. Summary of e5500 and e6500 differences

Feature	e5500	e6500	Notes
L2 cache	Private to core	Shared among cores in a core cluster	The e6500 core provides a shared backside L2 cache. This changes the L2 cache configuration on control methods. L2 cache and configuration is provided through memory mapped registers. The shared backside L2 is inline and provides snoop filtering for the cores in the complex. The L1 data cache now writes through to the L2 cache. Some MCSR bits for the integrated backside L2 cache are removed.
L2 cache size	512 KB	2048 KB	The e6500 core includes a larger L2 cache.
Branch prediction	Uses BTB, link stack, and STIC/STAC	Uses BTB, link stack, STIC/STAC, link stack underflow, global history, and pattern history	The e6500 core improves branch prediction, including function call and return prediction.
mfocrf optimization	Executes as **mfcr** and is serialized	Executes without serialization	Improves **mfocrf** instruction latency and repeat rate from five cycles to one.
LR and CTR optimization	Some LR/CTR accesses and updates are serialized and can stall	LR and CTR are fully renamed, and associated branches and **mtspr** and **mfspr** instructions execute without serialization	Improves performance of some subroutine linkage.
Byte and halfword load and reserve and store conditional instructions	Not present	Present	The **lbarx**, **lharx**, **stbcx.**, and **sthcx.** instructions are provided for doing byte and halfword load and reserve and store conditional operations.
Cache locking query	Cache locking operations use L1CSRx[CUL] to post a status about whether cache locking attempt was successful	Cache locks can be queried by using new lock query instructions, **dcblq.** and **icblq.**	Cache lock query allows software to enquire about locks and is more consistent with the way locks are established.
wait instruction	**wait** instruction only waits for an asynchronous interrupt	**wait** instruction can wait for interrupt or reservation to clear. Additional hint provided to enter lower power mode immediately.	The **wait** instruction can now wait for a store from another processor, and a programmer can specify whether to wait in a lower power mode.
miso instruction	Not present	Present	The **miso** instruction allows software to hint that all previous stores should be propagated to the coherency point to improve performance.

Table 1-3. Summary of e5500 and e6500 differences

Feature	e5500	e6500	Notes
Elemental memory barriers	Not present, lightest weight memory barrier is **sync 1** (**lwsync**)	Present	**sync** L,E provides more explicit memory barriers for improved performance using an additional operand to sync. Software can be coded using the new barriers such that e500mc and e5500 cores will execute using a correct memory barrier. Barriers provided are load with load, load with store, store with load, and store with store for cacheable non-write through required memory.
Real address size	36 bits	40 bits	Allows for larger physical address space.
Logical to real address translation (LRAT)	Required to be performed by hypervisor software	LRAT hardware provides the translation	On guest supervisor writes to the TLB, the LRAT structure provides the logical to real address translation, which reduces hypervisor overhead. The hypervisor maintains the LRAT table. A guest OS can now execute **tlbwe** without causing a hypervisor privilege interrupt.
TLB0 size and associativity	512 entries, four-way associative	1024 entries, eight-way associative	More TLB entries with greater associativity reduces conflict and capacity TLB misses.
TLB1 entry page sizes	4 KB to 4 GB (in powers of 4)	4 KB to 1 TB (in powers of 2)	Single page can map all of physical address space. More page sizes available with power-of-2 page sizes.
Hardware page table loading of TLB0 (4 KB pages)	Not present	Present	Hardware can resolve certain TLB misses by performing a hardware tablewalk and load TLB entries from a page table in memory.
Number of supported bits in PID register	8	14	Supports up to 16 K simultaneous PID values in use.
TLB0 parity	Not present	Present	Parity detection provided. Hardware flushes the TLB when an error is detected.
CCSRBAR setting available in SPR	Not present	Present	New SPR SCCSRBAR is added for software to read the current CCSRBAR setting. See Section 2.7.11, "Shifted CCSRBAR (SCCSRBAR) register".
Guest performance monitor interrupt	Not present	Present	Performance monitor interrupt can be taken directly in the guest OS.
Completion buffer and rename register entries	14	16	More completion and rename entries increases performance.
Performance monitor counters	4	6	More performance monitor counters allows for greater flexibility when analyzing performance issues. See Section 2.16, "Performance monitor registers (PMRs)."
Debug features: Imprecise debug events	Present	Not present	Imprecise debug events that occur when MSR[DE] = 0 are no longer supported.

Table 1-3. Summary of e5500 and e6500 differences

Feature	e5500	e6500	Notes
Debug feature: Instruction address compare (IAC) events	4	8	More IAC compares.
Debug feature: Assignment of debug events between internal and external debugger	All debug events are assigned to either internal or external debugger	Individual assignment of events to internal or external debugger	Individual events can be assigned between internal and external debuggers allowing both to operate simultaneously.
Debug feature: Watchdog timer	Runs when halted in external debug mode	Is suppressed from causing timeouts when halted in external debug mode	Makes it easier to prevent watchdog timeouts when the external debugger halts the processor.
Debug feature: **dni** instruction	Not present	Present	The **dni** instruction can be used as a breakpoint instruction and can trigger external events.
Debug feature: Nexus trace	Less trace bandwidth	Improved trace bandwidth plus additional Nexus features	Only transmits indirect branch history messages for **blr**-type instructions when link register has changed. Provides performance profiler counter message in trace stream, adds timestamp correlation message, captures PC snapshot in trace for events such as profile counter overflow, adds indication of clock frequency changes to Nexus trace stream, and extends watchpoint message event field to handle additional events.
Power management: Wake up on message acceptance	Not present	Present	Processor can wake up from power management state when a message (from **msgsnd**) is received and accepted.
Power management: Power management control register	Not present	Present	New power management control register (PWRMGTCR0) for controlling low power modes. See Section 2.7.7, “Power Management Control 0 (PWRMGTCR0) register.”
Power management: Static power reduction	Not present	Present	New power management states that reduce static power consumption while retaining their state.
Power management: Turn off AltiVec unit or reduce AltiVec static power when not in use	Not present	Present	See Section 2.7.7, “Power Management Control 0 (PWRMGTCR0) register.”
MCARUA as an alias for the upper 32-bits of MCAR	Not present	Present	Both MCARU and MCARUA provide the same alias. The MCARU alias may be phased out in future versions of the architecture. See Section 2.9.9, “Machine-check address registers (MCAR/MCARU/MCARUA).”

Chapter 2
Register Model

This chapter describes implementation-specific details of the register model as it is implemented on the e6500 core processors. It identifies all registers that are implemented on the e6500 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). *EREF* and *AltiVec Technology Programming Environments Manual for Power ISA Processors* describe these registers.

It is important to note that a device that integrates the e6500 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, see the e6500 core integration chapter in the reference manual for the device that incorporates the e6500 core. The register summary chapter in the device reference manual fully describes all registers and register fields as they are implemented on the device.

Only registers associated with the programming model of the core are described in this chapter.

The e6500 core is a dual-threaded machine and, as such, has some architected states that are duplicated and private to each thread and other architected states that are shared by the threads. Unless otherwise noted in this document, architected states are private to each thread, and each thread has its own independent copy of said state. In general, this document does not explicitly label states as private, but instead labels states that are shared. Note that writing to shared states may require special synchronization procedures that may involve disabling and enabling a thread.

2.1 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 2.3.1, "General-purpose registers (GPRs)."
- Floating-point registers (FPRs)—used as source and destination operands for floating-point computation operations. See Section 2.4.1, "Floating-point registers (FPRs)."
- Vector registers (VRs)—used as source and destination operands for vector computation operations. See Section 2.5.1, "Vector registers (VRs)."
- Special-purpose registers (SPRs)—accessed with the Move To Special-Purpose Register (**mtspr**) and Move From Special-Purpose Register (**mfspr**) instructions. Section 2.2.2, "Special-purpose registers (SPRs)," lists SPRs.
- System-level registers that are not SPRs. These are as follows:

- — Machine State (MSR) register—accessed with the Move To Machine State Register (**mtmsr**) and Move From Machine State Register (**mfmsr**) instructions. See Section 2.7.1, “Machine State (MSR) register.”
- — Condition (CR) register—bits are grouped into eight 4-bit fields, CR0-CR7, which are set as follows:
  - Specified CR fields are set by a Move To CR From GPR (**mtcrf**) instruction.
  - A specified CR field is set by a Move To CR from another CR field (**mcrf**) or from XER (**mcrxr**) instruction.
  - CR0 is set as the implicit result of an integer instruction.
  - CR1 is set as the implicit result of a floating-point instruction.
  - CR6 is set as the implicit result of an AltiVec compare instruction.
  - A specified CR field is set as the result of an integer or floating-point compare instruction.

  See Section 2.6.1, “Condition (CR) register.”

- Memory-mapped registers (MMRs)—accessed through load and store instructions. See Section 2.2.3, “Memory-mapped registers (MMRs).”
- Thread management registers (TMRs)—accessed by using dedicated move to and move from instructions (**mttmr** and **mftmr**). See Section 2.15, “Multi-threaded operation management registers.”
- Performance monitor registers (PMRs)—accessed by using dedicated move to and move from instructions (**mtpmr** and **mfpmr**). See Section 2.16, “Performance monitor registers (PMRs).”

2.2 e6500 register model

The following sections describe the e6500 core register model as defined by the architecture and the additional implementation-specific registers unique to the e6500 core.

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

- Registers used for integer operations, such as the general-purpose registers (GPRs) and the Integer Exception (XER) register. These registers are described in Section 2.3, “Registers for integer operations.”
- Registers used for floating-point operations, such as the floating-point registers (FPRs) and the Floating-Point Status and Control (FPSCR) register. These registers are described in Section 2.4, “Registers for floating-point operations.”
- Registers used for AltiVec operations, such as the vector registers (VRs) and the Vector Status and Control (VSCR) register. These registers are described in Section 2.5, “Registers for vector operations.”
- Condition (CR) register—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 (MSR) register—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 (MSR) register.”

- Special-purpose registers (SPRs)—accessed explicitly using **mtspr** and **mfspr** instructions. SPRs are listed in Table 2-2 in Section 2.2.2, “Special-purpose registers (SPRs).”
- Thread management registers (TMRs)—accessed explicitly using **mttmr** and **mftmr** instructions. TMRs are listed in Table 2-6 in Section 2.15, “Multi-threaded operation management registers.”
- Performance monitor registers (PMRs) — accessed with move to and move from PMR instructions (**mtpmr** and **mfpmr**).
- Memory-mapped registers (MMRs)—accessed through load and store instructions. Addresses for memory-mapped registers are translated through the MMU like normal loads and stores and are subject to further translation as defined by the SoC. MMRs associated with the processor are shared L2 control and status registers. See Section 2.2.3, “Memory-mapped registers (MMRs)” and Section 2.12, “L2 cache registers”.

SPRs, PMRs, and TMRs 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 2.7.4, “Branch Unit Control and Status (BUCSR) register”
- Section 2.7.5, “Hardware Implementation-Dependent 0 (HID0) register”
- Section 2.11, “L1 cache registers”
- Section 2.13, “MMU registers”
- Section 2.14, “Internal debug registers”
- Section 2.15, “Multi-threaded operation management registers”
- Section 2.16, “Performance monitor registers (PMRs)”

2.2.1 64-bit registers

The e6500 core is a 64-bit implementation of Power ISA. Some registers are defined by the architecture to be 64 bits. On 32-bit implementations, processors normally implement only the lower 32 bits of these registers. There are some exceptions, such as Time Base; however, those exceptions allow 32-bit access to the upper 32 bits of the register through a different register port.

To facilitate porting software from 32-bit mode to 64-bit mode, this table lists the registers that are 64-bit registers on the e6500 core.

Table 2-1. 64-bit registers

Register Abbreviation	Name	Notes
GPR	General-purpose registers (r0 - r31)	—
FPR	Floating-point registers (fr0 - fr31)	FPRs are always 64 bits, even in 32-bit implementations.
ATBL	Alternate Time Base Lower	Access to ATBL in 64-bit mode returns the entire 64-bit counter.
CSRR0	Critical Save/Restore 0	—
CTR	Count	—
DAC1	Data Address Compare 1	—
DAC2	Data Address Compare 2	—
DEAR	Data Exception Address	—
DSRR0	Debug Save/Restore 0	—
GDEAR	Guest Data Exception Address	—
GIVPR	Guest Interrupt Vector Prefix	—
GSPRG0	Guest SPR General 0	—
GSPRG1	Guest SPR General 1	—
GSPRG2	Guest SPR General 2	—
GSPRG3	Guest SPR General 3	—
GSRR0	Guest Save/Restore 0	—
IAC1	Instruction Address Compare 1	—
IAC2	Instruction Address Compare 2	—
IAC3	Instruction Address Compare 3	—
IAC4	Instruction Address Compare 4	—
IAC5	Instruction Address Compare 5	—
IAC6	Instruction Address Compare 6	—
IAC7	Instruction Address Compare 7	—
IAC8	Instruction Address Compare 8	—
IVPR	Interrupt Vector Prefix	—
LPER	Logical Page Exception	—
LR	Link	—
MAS0_MAS1	MMU Assist 0 and MMU Assist 1	—
MAS2	MMU Assist 2	—
MAS5_MAS6	MMU Assist 5 and MMU Assist 6[1]	—
MAS7_MAS3	MMU Assist 7 and MMU Assist 3	—
MAS8_MAS1	MMU Assist 8 and MMU Assist 1	—

Table 2-1. 64-bit registers (continued)

Register Abbreviation	Name	Notes
MCAR	Machine-Check Address	—
MCSRR0	Machine-Check Save/Restore 0	—
SPRG0	SPR General 0	—
SPRG1	SPR General 1	—
SPRG2	SPR General 2	—
SPRG3	SPR General 3	—
SPRG4	SPR General 4	—
SPRG5	SPR General 5	—
SPRG6	SPR General 6	—
SPRG7	SPR General 7	—
SPRG8	SPR General 8	—
SPRG9	SPR General 9	—
SRR0	Save/Restore 0	—
TBL(R)	Time Base Lower	Read access to TBL in 64-bit mode returns the entire 64-bit counter
XER	Integer Exception	Architecturally, the XER is a 64-bit register; however, the upper 32 bits of XER are reserved and always read as 0.

2.2.2 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. SPRs are accessed with the **mtspr** and **mfspr** instructions.

Access is given by the lowest level of privilege required to access the SPR. The access methods listed in Table 2-2 are defined as follows:

- User—denotes access is available for both **mtspr** and **mfspr,** regardless of privilege level.
- User RO—denotes access is available for only **mfspr,** regardless of privilege level.
- Guest supervisor—denotes access is available for both **mtspr** and **mfspr** when operating in supervisor mode (MSR[PR] = 0), regardless of the state of MSR[GS]. That is, it is available in hypervisor state, as well.
- Guest supervisor RO—denotes access is available for only **mfspr** when operating in supervisor mode (MSR[PR] = 0), regardless of the state of MSR[GS]. That is, it is available in hypervisor state, as well.
- Hypervisor—denotes access is available for both **mtspr** and **mfspr** when operating in hypervisor mode (MSR[GS,PR] = 00).

- Hypervisor RO—denotes access is available for only **mfspr** when operating in hypervisor mode (MSR[GS,PR] = 00).
- Hypervisor WO—denotes access is available for only **mtspr** when operating in hypervisor mode (MSR[GS,PR] = 00).
- Hypervisor R/Clear—denotes access is available for both **mtspr** and **mfspr** 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.
- Shared—denotes an SPR that is shared by all threads.

An **mtspr** or **mfspr** instruction that specifies an unsupported SPR number is considered an invalid instruction. The e6500 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 that are privileged, a **mtspr** or **mfspr** instruction while in user mode (MSR[PR] = 1) causes a privilege operation program exception.

Note that the behavior of the e6500 core 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.

Attempting to access a supported SPR that is hypervisor-privileged while in the guest-supervisor state causes an embedded hypervisor privilege exception. For example, attempting to read an SPR that has "Hypervisor RO" privilege while in the guest-supervisor state causes an embedded hypervisor privilege exception and subsequent interrupt. See Section 4.9.21, "Hypervisor privilege interrupt—IVOR41" for a complete list of actions that cause embedded hypervisor privilege exceptions.

The table summarizes SPR access methods.

Table 2-2. Special-purpose registers (by SPR abbreviation)

SPR Abbreviation	Name	Defined SPR Number	Access	Shared	Section/Page
ATBL	Alternate Time Base Lower	526	User RO	Yes	2.8.6/2-28
ATBU	Alternate Time Base Upper	527	User RO	Yes	2.8.6/2-28
BUCSR	Branch Unit Control and Status[1]	1013	Hypervisor	—	2.7.4/2-21
CDCSR0	Core Device Control and Status	696	Hypervisor	Yes	2.7.6/2-23
CIR	Chip Identification (alias to SVR)	283	Guest supervisor RO	Yes	2.7.10/2-25
CSRR0	Critical Save/Restore 0	58	Hypervisor	—	2.9.1/2-29
CSRR1	Critical Save/Restore 1	59	Hypervisor	—	2.9.1/2-29
CTR	Count	9	User	—	2.6.3/2-18
DAC1	Data Address Compare 1[1]	316	Hypervisor	—	2.14.11/2-100
DAC2	Data Address Compare 2[1]	317	Hypervisor	—	2.14.11/2-100
DBCR0	Debug Control 0 [1]	308	Hypervisor	—	2.14.4/2-85
DBCR1	Debug Control 1[1]	309	Hypervisor	—	2.14.5/2-88
DBCR2	Debug Control 2[1]	310	Hypervisor	—	2.14.6/2-91

Table 2-2. Special-purpose registers (by SPR abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR Number	Access	Shared	Section/Page
DBCR4	Debug Control 4[1]	563	Hypervisor	—	2.14.7/2-93
DBCR5	Debug Control 5[1]	564	Hypervisor	—	2.14.8/2-94
DBRR0	Debug Resource Request 0[1]	700	Hypervisor	—	2.14.2/2-82
DBSR	Debug Status [1]	304	Hypervisor R/Clear	—	2.14.9/2-96
DBSRWR	Debug Status Write[1]	306	Hypervisor	—	2.14.9/2-96
DDAM	Debug Data Acquisition Message	576	User	—	2.14.14/2-102
DEAR	Data Exception Address	61	Guest supervisor[2]	—	2.8.5/2-28
DEC	Decrementer	22	Hypervisor	—	2.8.4/2-28
DECAR	Decrementer Auto-Reload	54	Hypervisor WO	—	2.8.4/2-28
DEVENT	Debug Event	975	User	—	2.14.13/2-101
DSRR0	Debug Save/Restore 0	574	Hypervisor	—	2.9.1/2-29
DSRR1	Debug Save/Restore 1	575	Hypervisor	—	2.9.1/2-29
EDBRAC0	External Debug Resource Allocation Control 0[1]	638	Hypervisor RO	—	2.14.3/2-83
EPCR	Embedded Processor Control	307	Hypervisor	—	2.7.3/2-21
EPTCFG	Embedded Page Table Configuration	350	Hypervisor RO	Yes	2.13.7/2-66
EPLC	External PID Load Context[1]	947	Guest supervisor[3]	—	2.13.11.1/2-79
EPR	External Proxy	702	Guest supervisor RO[2]	—	2.9.6/2-32
EPSC	External PID Store Context[1]	948	Guest supervisor[3]	—	2.13.11.2/2-80
ESR	Exception Syndrome	62	Guest supervisor[2]	—	2.9.7/2-33
GDEAR	Guest Data Exception Address	381	Guest supervisor	—	2.8.5/2-28
GEPR	Guest External Proxy	380	Guest supervisor	—	2.9.6/2-32
GESR	Guest Exception Syndrome	383	Guest supervisor	—	2.9.7/2-33
GIVOR13	Guest Data TLB Error Interrupt Offset	444	Hypervisor[4]	—	2.9.4/2-31
GIVOR14	Guest Instruction TLB Error Interrupt Offset	445	Hypervisor[4]	—	2.9.4/2-31
GIVOR2	Guest Data Storage Interrupt Offset	440	Hypervisor[4]	—	2.9.4/2-31
GIVOR3	Guest Instruction Storage Interrupt Offset	441	Hypervisor[4]	—	2.9.4/2-31
GIVOR35	Guest Performance Monitor Interrupt Offset	464	Hypervisor[4]	—	2.9.4/2-31
GIVOR4	Guest External Input Interrupt Offset	442	Hypervisor[4]	—	2.9.4/2-31
GIVOR8	Guest System Call Interrupt Offset	443	Hypervisor[4]	—	2.9.4/2-31
GIVPR	Guest Interrupt Vector Prefix	447	Hypervisor[4]	—	2.9.4/2-31
GPIR	Guest Processor ID	382	Guest supervisor[4]	—	2.9.8/2-35
GSPRG0	Guest SPR General 0	368	Guest supervisor	—	2.10/2-39
GSPRG1	Guest SPR General 1	369	Guest supervisor	—	2.10/2-39

Table 2-2. Special-purpose registers (by SPR abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR Number	Access	Shared	Section/Page
GSPRG2	Guest SPR General 2	370	Guest supervisor	—	2.10/2-39
GSPRG3	Guest SPR General 3	371	Guest supervisor	—	2.10/2-39
GSRR0	Guest Save/Restore 0	378	Guest supervisor	—	2.9.1/2-29
GSRR1	Guest Save/Restore 1	379	Guest supervisor	—	2.9.1/2-29
HID0	Hardware Implementation Dependent 0[1]	1008	Hypervisor	Yes	2.7.5/2-22
IAC1	Instruction Address Compare 1[1]	312	Hypervisor	—	2.14.10/2-99
IAC2	Instruction Address Compare 2[1]	313	Hypervisor	—	2.14.10/2-99
IAC3	Instruction Address Compare 3[1]	314	Hypervisor	—	2.14.10/2-99
IAC4	Instruction Address Compare 4[1]	315	Hypervisor	—	2.14.10/2-99
IAC5	Instruction Address Compare 5[1]	565	Hypervisor	—	2.14.10/2-99
IAC6	Instruction Address Compare 6[1]	566	Hypervisor	—	2.14.10/2-99
IAC7	Instruction Address Compare 7[1]	567	Hypervisor	—	2.14.10/2-99
IAC8	Instruction Address Compare 8[1]	568	Hypervisor	—	2.14.10/2-99
IVOR0	Critical Input Interrupt Offset	400	Hypervisor	Yes	2.9.5/2-31
IVOR1	Machine Check Interrupt Offset	401	Hypervisor	Yes	2.9.5/2-31
IVOR10	Decrementer Interrupt Offset	410	Hypervisor	Yes	2.9.5/2-31
IVOR11	Fixed-Interval Timer Interrupt Offset	411	Hypervisor	Yes	2.9.5/2-31
IVOR12	Watchdog Timer Interrupt Offset	412	Hypervisor	Yes	2.9.5/2-31
IVOR13	Data TLB Error Interrupt Offset	413	Hypervisor	Yes	2.9.5/2-31
IVOR14	Instruction TLB Error Interrupt Offset	414	Hypervisor	Yes	2.9.5/2-31
IVOR15	Debug Interrupt Offset	415	Hypervisor	Yes	2.9.5/2-31
IVOR2	Data Storage Interrupt Offset	402	Hypervisor	Yes	2.9.5/2-31
IVOR3	Instruction Storage Interrupt Offset	403	Hypervisor	Yes	2.9.5/2-31
IVOR32	AltiVec Unavailable Interrupt Offset	528	Hypervisor	Yes	2.9.5/2-31
IVOR33	AltiVec Assist Interrupt Offset	529	Hypervisor	Yes	2.9.5/2-31
IVOR35	Performance Monitor Interrupt Offset	531	Hypervisor	Yes	2.9.5/2-31
IVOR36	Processor Doorbell Interrupt Offset	532	Hypervisor	Yes	2.9.5/2-31
IVOR37	Processor Doorbell Critical Interrupt Offset	533	Hypervisor	Yes	2.9.4/2-31
IVOR38	Guest Processor Doorbell Interrupt Offset	432	Hypervisor	Yes	2.9.5/2-31
IVOR39	Guest Processor Doorbell Critical and Machine-Check Interrupt Offset	433	Hypervisor	Yes	2.9.5/2-31
IVOR4	External Input Interrupt Offset	404	Hypervisor	Yes	2.9.5/2-31
IVOR40	Hypervisor System Call Interrupt Offset	434	Hypervisor	Yes	2.9.5/2-31
IVOR41	Hypervisor Privilege Interrupt Offset	435	Hypervisor	Yes	2.9.5/2-31

Table 2-2. Special-purpose registers (by SPR abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR Number	Access	Shared	Section/Page
IVOR42	LRAT Error Interrupt Offset	436	Hypervisor	Yes	2.9.5/2-31
IVOR5	Alignment Interrupt Offset	405	Hypervisor	Yes	2.11.5/2-42
IVOR6	Program Interrupt Offset	406	Hypervisor	Yes	2.9.5/2-31
IVOR7	Floating-Point Unavailable Interrupt Offset	407	Hypervisor	Yes	2.9.5/2-31
IVOR8	System Call Interrupt Offset	408	Hypervisor	Yes	2.9.5/2-31
IVOR9	APU Unavailable Interrupt Offset	409	Hypervisor	Yes	2.9.5/2-31
IVPR	Interrupt Vector Prefix	63	Hypervisor	Yes	2.9.4/2-31
L1CFG0	L1 Cache Configuration 0	515	User RO	Yes	2.11.4/2-41
L1CFG1	L1 Cache Configuration 1	516	User RO	Yes	2.11.5/2-42
L1CSR0	L1 Cache Control and Status 0[1]	1010	Hypervisor	Yes	2.11.1/2-40
L1CSR1	L1 Cache Control and Status 1[1]	1011	Hypervisor	Yes	2.11.2/2-40
L1CSR2	L1 Cache Control and Status 2[1]	606	Hypervisor	Yes	2.11.2/2-40
LPIDR	Logical PID[1]	338	Hypervisor	—	2.13.1/2-62
LR	Link	8	User	—	2.6.2/2-18
LRATCFG	Logical to Real Address Translation Configuration	342	Hypervisor RO	Yes	2.13.8/2-67
LRATPS	Logical to Real Address Translation Page Size	343	Hypervisor RO	Yes	2.13.9/2-69
LPER	Logical Page Exception	56	Hypervisor	—	2.9.3/2-30
LPERU	Logical Page Exception Upper	57	Hypervisor	—	2.9.3/2-30
MAS0	MMU Assist 0[1]	624	Guest supervisor	—	2.13.10.1/2-70
MAS0_MAS1	MMU Assist 0 and MMU Assist 1[1]	373	Guest supervisor	—	2.13.10.10/2-79
MAS1	MMU Assist 1[1]	625	Guest supervisor	—	2.13.10.2/2-71
MAS2	MMU Assist 2[1]	626	Guest supervisor	—	2.13.10.3/2-72
MAS3	MMU Assist 3[1]	627	Guest supervisor	—	2.13.10.4/2-73
MAS4	MMU Assist 4[1]	628	Guest supervisor	—	2.13.10.5/2-75
MAS5	MMU Assist 5[1]	339	Hypervisor	—	2.13.10.6/2-76
MAS5_MAS6	MMU Assist 5 and MMU Assist 6[1]	348	Hypervisor	—	2.13.10.10/2-79
MAS6	MMU Assist 6[1]	630	Guest supervisor	—	2.13.10.7/2-76
MAS7	MMU Assist 7[1]	944	Guest supervisor	—	2.13.10.8/2-77
MAS7_MAS3	MMU Assist 7 and MMU Assist 3[1]	372	Guest supervisor	—	2.13.10.10/2-79
MAS8	MMU Assist 8[1]	341	Hypervisor	—	2.13.10.9/2-78
MAS8_MAS1	MMU Assist 8 and MMU Assist 1[1]	349	Hypervisor	—	2.13.10.10/2-79

Table 2-2. Special-purpose registers (by SPR abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR Number	Access	Shared	Section/Page
MCAR	Machine-Check Address	573	Hypervisor RO	—	2.9.9/2-35
MCARU	Machine-Check Address Upper	569	Hypervisor RO	—	2.9.9/2-35
MCARUA	Machine-Check Address Upper Alias	637	Hypervisor RO	—	2.9.9/2-35
MCSR	Machine-Check Syndrome	572	Hypervisor	—	2.9.10/2-36
MCSRR0	Machine-Check Save/Restore 0	570	Hypervisor	—	2.9.1/2-29
MCSRR1	Machine-Check Save/Restore 1	571	Hypervisor	—	2.9.1/2-29
MMUCFG	MMU Configuration	1015	Hypervisor RO	Yes	2.13.4/2-63
MMUCSR0	MMU Control and Status 0[1]	1012	Hypervisor	Yes	2.13.3/2-62
MSRP	MSR Protect[1]	311	Hypervisor	—	2.7.2/2-20
NIA	Next Instruction Address	559	External debugger	—	9.2.5.2/99-5
NPIDR[5]	Nexus Processor ID	517	User	—	2.14.15/2-102
NSPC	Nexus SPR Access Configuration	984	Hypervisor	—	2.14.12/2-100
NSPD	Nexus SPR Access Data	983	Hypervisor	—	2.14.12/2-100
PID	Process ID[1]	48	Guest supervisor	—	2.13.2/2-62
PIR	Processor ID	286	Guest supervisor[2]	—	2.9.8/2-35
PPR32	Processor Priority	898	User	—	2.15.1.6/2-107
PVR	Processor Version	287	Guest supervisor RO	Yes	2.7.8/2-24
PWRMGTCR0	Power Management Control 0	1019	Hypervisor	Yes	2.7.7/2-24
SCCSRBAR	Shifted CCSRBAR from SoC	1022	Hypervisor RO	Yes	2.7.11/2-25
SPRG0	SPR General 0	272	Guest supervisor[2]	—	2.10/2-39
SPRG1	SPR General 1	273	Guest supervisor[2]	—	2.10/2-39
SPRG2	SPR General 2	274	Guest supervisor[2]	—	2.10/2-39
SPRG3	SPR General 3	259	User RO[2]	—	2.10/2-39
SPRG3	SPR General 3	275	Guest supervisor[2]	—	2.10/2-39
SPRG4	SPR General 4	260	User RO	—	2.10/2-39
SPRG4	SPR General 4	276	Guest supervisor	—	2.10/2-39
SPRG5	SPR General 5	261	User RO	—	2.10/2-39
SPRG5	SPR General 5	277	Guest supervisor	—	2.10/2-39
SPRG6	SPR General 6	262	User RO	—	2.10/2-39
SPRG6	SPR General 6	278	Guest supervisor	—	2.10/2-39
SPRG7	SPR General 7	263	User RO	—	2.10/2-39
SPRG7	SPR General 7	279	Guest supervisor	—	2.10/2-39
SPRG8	SPR General 8	604	Hypervisor	—	2.10/2-39

Table 2-2. Special-purpose registers (by SPR abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR Number	Access	Shared	Section/Page
SPRG9	SPR General 9	605	Guest supervisor	—	2.10/2-39
SRR0	Save/Restore 0	26	Guest supervisor[2]	—	2.9.1/2-29
SRR1	Save/Restore 1	27	Guest supervisor[2]	—	2.9.1/2-29
SVR	System Version	1023	Guest supervisor RO	Yes	2.7.9/2-25
TBL(R)	Time Base Lower (Read)	268	User RO	Yes	2.8.3/2-27
TBL(W)	Time Base Lower (Write)	284	Hypervisor WO	Yes	2.8.3/2-27
TBU(R)	Time Base Upper (Read)	269	User RO	Yes	2.8.3/2-27
TBU(W)	Time Base Upper (Write)	285	Hypervisor WO	Yes	2.8.3/2-27
TCR	Timer Control	340	Hypervisor	—	2.8.1/2-27
TENC	Thread Enable Clear	439	Hypervisor	Yes	2.15.1/2-103
TENS	Thread Enable Set	438	Hypervisor	Yes	2.15.1/2-103
TENSR	Thread Enable Status	437	Hypervisor RO	Yes	2.15.1/2-103
TIR	Thread Identification	446	Hypervisor RO	—	2.15.1/2-103
TLB0CFG	TLB 0 Configuration	688	Hypervisor RO	Yes	2.13.5/2-64
TLB0PS	TLB 0 Page Size	344	Hypervisor RO	Yes	2.13.6/2-66
TLB1CFG	TLB 1 Configuration	689	Hypervisor RO	Yes	2.13.5/2-64
TLB1PS	TLB 1 Page Size	345	Hypervisor RO	Yes	2.13.6/2-66
TSR	Timer Status	336	Hypervisor R/Clear	—	2.8.2/2-27
USPRG0 (VRSAVE)	User SPR General 0[6]	256	User	—	2.10/2-39
XER	Integer Exception	1	User	—	2.3.2/2-16

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

[2] When these registers are accessed in the guest-supervisor state, the accesses are mapped to their analogous guest SPRs (for example, DEAR is mapped to GDEAR). See Section 2.2.2.1, “Register mapping in the guest–supervisor state.”

[3] Certain fields in this register are only writeable when in the hypervisor state.

[4] This register is only writeable in the hypervisor state, but can be read in the guest-supervisor state.

[5] NPIDR contents are transferred to the Nexus port whenever it is written.

[6] USPRG0 is a separate physical register from SPRG0.

2.2.2.1 Register mapping in the guest–supervisor state

To improve emulation efficiency while providing 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, accesses to certain hypervisor-state registers are automatically redirected to the appropriate guest-state registers when in the guest–supervisor state. This 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 Table 2-3 are changed by the processor to the registers given in the table when MSR[PR] = 0 and MSR[GS] = 1. Accesses to these registers are not mapped when in the hypervisor state (MSR[PR] = 0 and MSR[GS] = 0) or when operating unprivileged (MSR[PR] = 1). The exception is that an unprivileged access to SPRG3 (SPR 259) is also mapped to GSPRG3.

Note that in the guest-supervisor state, execution of **rfi** is mapped to **rfgi**, and indirect accesses to SRR0 and SRR1 through execution of the **rfgi** instruction are handled through instruction mapping.

This table provides register mapping in the guest-supervisor state.

Table 2-3. Register mapping in the guest–supervisor state

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

2.2.3 Memory-mapped registers (MMRs)

MMRs are on-chip registers implemented on the processor core or within a cluster of processor cores. They are used to control and configure devices within the integrated device. MMRs are defined for the shared backside L2 cache.

This section describes the MMR address assignments that are supported by the e6500 core complex. The definitions of each MMR are found later in this chapter and are organized by function.

MMRs are accessed by performing loads and stores to the assigned address of the MMR. The MMR address is specified as an offset from a base address. The base address is a block of addresses defined by the SoC architecture and can be determined by consulting the integrated device reference manual as part of the CCSR Address Map. For MMRs defined in this document, both the name of the block of addresses and the offset within that block are used to identify the address.

MMRs do not have privilege associated with them and are accessible if mapped by a valid TLB entry. Therefore, system software should take care when mapping MMRs to address spaces.

MMRs located at address *A*, which are defined as 64 bits, are accessed with load or store doubleword instructions to address *A*. The two 32-bit portions may be accessed by two load or store word instructions that address the upper 32 bits at address *A* and the lower 32 bits at address *A*+4. MMRs that are defined as 32 bits are accessed with load or store word instructions at address *A*.

The following table summarizes the MMR blocks and each block's offset from CCSRBAR. The starting real address of the block is determined by reading the SCCSRBAR, shifting the contents left by 24 bits, and adding the block offset.

Table 2-4. Memory-mapped register blocks (by offset)

Block Offset from CCSRBAR	MMR Block Name
0xC2_0000	Shared L2 cluster 1
0xC6_0000	Shared L2 cluster 2
0xCA_0000	Shared L2 cluster 3
0xCE_0000	Shared L2 cluster 4
0xD2_0000	Shared L2 cluster 5
0xD6_0000	Shared L2 cluster 6
0xDA_0000	Shared L2 cluster 7
0xDE_0000	Shared L2 cluster 8
0xE2_0000	Shared L2 cluster 9
0xE6_0000	Shared L2 cluster 10
0xEA_0000	Shared L2 cluster 11
0xEE_0000	Shared L2 cluster 12
0xF2_0000	Shared L2 cluster 13
0xF6_0000	Shared L2 cluster 14
0xFA_0000	Shared L2 cluster 15
0xFE_0000	Shared L2 cluster 16

The following table summarizes MMRs for the shared L2, which is *Shared L2 cluster x*, where x identifies the shared L2 cluster from 1 to the number of clusters in the integrated device.

Table 2-5. Memory-mapped registers for block 'shared L2 cluster *x*' (by offset)

MMR Abbreviation	Defined MMR offset in block	Name	Length (in bits)	Section/ Page
L2CSR0	0x000	L2 Cache Control and Status 0	32	2.12.2/2-44
L2CSR1	0x004	L2 Cache Control and Status 1	32	2.12.3/2-47
L2CFG0	0x008	L2 Cache Configuration 0	32	2.12.1/2-43
L2PIR0	0x200	L2 Cache Partitioning ID 0	32	2.12.4.1/2-50

Table 2-5. Memory-mapped registers for block ‘shared L2 cluster *x*’ (by offset) (continued)

MMR Abbreviation	Defined MMR offset in block	Name	Length (in bits)	Section/ Page
L2PAR0	0x208	L2 Cache Partitioning Allocation 0	32	2.12.4.2/2-51
L2PWR0	0x20c	L2 Cache Partitioning Way 0	32	2.12.4.3/2-53
L2PIR1	0x210	L2 Cache Partitioning ID 1	32	2.12.4.1/2-50
L2PAR1	0x218	L2 Cache Partitioning Allocation 1	32	2.12.4.2/2-51
L2PWR1	0x21c	L2 Cache Partitioning Way 1	32	2.12.4.3/2-53
L2PIR2	0x220	L2 Cache Partitioning ID 2	32	2.12.4.1/2-50
L2PAR2	0x228	L2 Cache Partitioning Allocation 2	32	2.12.4.2/2-51
L2PWR2	0x22c	L2 Cache Partitioning Way 2	32	2.12.4.3/2-53
L2PIR3	0x230	L2 Cache Partitioning ID 3	32	2.12.4.1/2-50
L2PAR3	0x238	L2 Cache Partitioning Allocation 3	32	2.12.4.2/2-51
L2PWR3	0x23c	L2 Cache Partitioning Way 3	32	2.12.4.3/2-53
L2PIR4	0x240	L2 Cache Partitioning ID 4	32	2.12.4.1/2-50
L2PAR4	0x248	L2 Cache Partitioning Allocation 4	32	2.12.4.2/2-51
L2PWR4	0x24c	L2 Cache Partitioning Way 4	32	2.12.4.3/2-53
L2PIR5	0x250	L2 Cache Partitioning ID 5	32	2.12.4.1/2-50
L2PAR5	0x258	L2 Cache Partitioning Allocation 5	32	2.12.4.2/2-51
L2PWR5	0x25c	L2 Cache Partitioning Way 5	32	2.12.4.3/2-53
L2PIR6	0x260	L2 Cache Partitioning ID 6	32	2.12.4.1/2-50
L2PAR6	0x268	L2 Cache Partitioning Allocation 6	32	2.12.4.2/2-51
L2PWR6	0x26c	L2 Cache Partitioning Way 6	32	2.12.4.3/2-53
L2PIR7	0x270	L2 Cache Partitioning ID 7	32	2.12.4.1/2-50
L2PAR7	0x278	L2 Cache Partitioning Allocation 7	32	2.12.4.2/2-51
L2PWR7	0x27c	L2 Cache Partitioning Way 7	32	2.12.4.3/2-53
L2ERRINJHI	0xe00	L2 Cache Error Injection Mask High	32	2.12.5.10/2-61
L2ERRINJLO	0xe04	L2 Cache Error Injection Mask Low	32	2.12.5.10/2-61
L2ERRINJCTL	0xe08	L2 Cache Error Injection Control	32	2.12.5.9/2-60
L2CAPTDATAHI	0xe20	L2 Cache Error Capture Data High	32	2.12.5.6/2-59
L2CAPTDATALO	0xe24	L2 Cache Error Capture Data Low	32	2.12.5.6/2-59
L2CAPTECC	0xe28	L2 Cache Error Capture ECC Syndrome	32	2.12.5.7/2-59
L2ERRDET	0xe40	L2 Cache Error Detect	32	2.12.5.2/2-55
L2ERRDIS	0xe44	L2 Cache Error Disable	32	2.12.5.1/2-54
L2ERRINTEN	0xe48	L2 Cache Error Interrupt Enable	32	2.12.5.3/2-57
L2ERRATTR	0xe4c	L2 Cache Error Attribute	32	2.12.5.8/2-59

Table 2-5. Memory-mapped registers for block ‘shared L2 cluster *x*’ (by offset) (continued)

MMR Abbreviation	Defined MMR offset in block	Name	Length (in bits)	Section/ Page
L2ERREADDR	0xe50	L2 Cache Error Extended Address	32	2.12.5.5/2-59
L2ERRADDR	0xe54	L2 Cache Error Address	32	2.12.5.5/2-59
L2ERRCTL	0xe58	L2 Cache Error Control	32	2.12.5.4/2-58

2.2.3.1 Synchronization requirements for memory-mapped registers

MMRs associated with the e6500 core complex require synchronization to ensure that operations are performed when an MMR is written with a store instruction. The general synchronization requirement is to follow a store to an MMR with a load to the same MMR and verify that the operation is complete. Other synchronizations or actions that may be required are specific to each MMR and are documented in *EREF* and in the specific register descriptions in Chapter 2, “Register Model.”

2.2.4 Thread management registers (TMRs)

TMRs are on-chip registers implemented in the processor core that are used to control the use of threads in the e6500 core and other architected processor resources related to threads.

This section describes the TMRs that are implemented in the e6500. The definition of each individual TMR is contained in Section 2.15.2, “Thread management registers (TMRs)” organized by function. Note that the e6500 implementation of a TMR may be a subset of the architectural definition.

TMRs are accessed with the **mttmr** and **mftmr** instructions. Access is given by the lowest level of privilege required to access the TMR. Unlike SPRs, TMRs do not use bit 5 to denote privilege. The access methods listed in Table 2-6 are defined as follows:

- Hypervisor—denotes access is available for both **mttmr** and **mftmr** when operating in hypervisor mode (MSR[GS,PR] = 00).
- Hypervisor RO—denotes access is available for only **mftmr** when operating in hypervisor mode (MSR[GS,PR] = 00).
- Hypervisor WO—denotes access is available for only **mttmr** when operating in hypervisor mode (MSR[GS,PR] = 00).
- Hypervisor R/Clear—denotes access is available for both **mttmr** and **mftmr** when operating in hypervisor mode (MSR[GS,PR] = 00); however, an **mttmr** only clears bit positions in the SPR that correspond to the bits set in the source GPR.
- Shared—denotes the register is shared among all the threads of a multi-threaded processor; otherwise, each thread has a private copy of the register.

An **mttmr** or **mftmr** instruction that specifies an unsupported TMR number is considered an invalid instruction. In user mode, the processor takes an illegal operation program exception on all accesses to unsupported, unprivileged TMRs (or read accesses to TMRs that are write-only and write accesses to TMRs that are read-only). In supervisor or hypervisor mode, such accesses are boundedly undefined.

Table 2-6. Thread management registers (by TMR number)

Defined TMR Number	TMR Abbreviation	Name	Length (in bits)	Access (shared)	Shared	Section/ Page
16	TMCFG0	Thread Management Configuration 0	32	Hypervisor RO	Yes	2.15.2.1/2-108
192	TPRI0	Thread 0 Priority	32	Hypervisor	Yes	2.15.2.4/2-110
193	TPRI1	Thread 1 Priority	32	Hypervisor	Yes	2.15.2.4/2-110
288	IMSR0	Thread 0 Machine State	32	Hypervisor WO	Yes	2.15.2.2/2-109
289	IMSR1	Thread 1 Machine State	32	Hypervisor WO	Yes	2.15.2.2/2-109
320	INIA0	Thread 0 Next Instruction Address	64	Hypervisor WO	Yes	2.15.2.2/2-109
321	INIA1	Thread 1 Next Instruction Address	64	Hypervisor WO	Yes	2.15.2.2/2-109

2.3 Registers for integer operations

The following sections describe registers defined for integer computational instructions.

2.3.1 General-purpose registers (GPRs)

GPR0–GPR31 provide operand space for support integer operations. The instruction formats provide 5-bit fields for specifying GPRs to be used in the execution of the instruction. Each GPR is a 64-bit register and can be used to contain effective address and integer data.

GPRs are implemented as defined by Power ISA and as described in *EREF*.

The e6500 core has two independent sets of GPRs, one set for each thread.

2.3.2 Integer Exception (XER) register

XER bits are set based on the operation of an instruction considered as a whole, not based on intermediate results. For example, the Subtract from Carrying (**subfc**) instruction specifies the result as the sum of three values. This final sum is actually accomplished with an intermediate sum of two values, which is then added to the third to produce the final sum. The bits in XER are only set based on the entire instruction operation, not the intermediate value produced during the operation.

Note that XER is an SPR.

The e6500 core implements XER as defined in *EREF*.

The e6500 core has two independent XERs, one for each thread.

2.4 Registers for floating-point operations

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

2.4.1 Floating-point registers (FPRs)

FPR0–FPR31 provide operand space for supporting floating-point operations. The instruction formats provide 5-bit fields for specifying 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 e6500 core implements FPRs as defined by Power ISA and as described in *EREF*.

The e6500 core has two independent sets of FPRs, one set for each thread.

2.4.2 Floating-Point Status and Control (FPSCR) register

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.

If FPSCR[NI] is set for a specific thread in the e6500 core, 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.

The e6500 core implements FPSCR as defined by Power ISA and described in *EREF*.

The e6500 core has two independent FPSCRs, one for each thread.

2.5 Registers for vector operations

The following sections describe registers defined for vector computational instructions.

2.5.1 Vector registers (VRs)

VR0–VR31 provide operand space for supporting vector (AltiVec) operations. The instruction formats provide 5-bit fields for specifying the VRs to be used in the execution of the instruction. Each VR is a 128-bit register and can be used to contain a vector of integer or single-precision floating-point data.

The e6500 core implements VRs as defined by Power ISA and as described in *EREF*.

The e6500 has two independent sets of VRs, one set for each thread.

2.5.2 Vector Status and Control (VSCR) register

VSCR is a 32-bit register that is read and written in a manner similar to the FPSCR.

VSCR has two defined bits:

- AltiVec non-Java mode bit—VSCR[111]
- AltiVec saturation bit—VSCR[127]

The remaining bits are reserved.

Special Move To Vector Status and Control Register (**mfvscr**) and Move From Vector Status and Control Register (**mtvscr**) instructions are provided to move the contents of VSCR to and from a vector register. When moved to or from a vector register, the 32-bit VSCR is right-justified in the 128-bit vector register. When moved to a vector register, the upper 96 bits VR*n* [0–95] of the vector register are cleared.

VSCR is more completely defined in the *AltiVec Technology Programming Environments Manual for Power ISA Processors*.

The e6500 core has two independent VSCRs, one for each thread.

2.6 Registers for branch operations

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

2.6.1 Condition (CR) register

The e6500 core implements CR as defined in *EREF* for integer instructions.

The e6500 core has two independent CRs, one for each thread.

2.6.2 Link (LR) register

LR can be used to provide the branch target address for a Branch Conditional to LR instruction. It also holds the return address after branch and link instructions.

Note that LR is an SPR.

The e6500 core implements LR as defined in *EREF*.

The e6500 core has two independent LRs, one for each thread.

2.6.3 Count (CTR) register

CTR 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.

The e6500 core implements CTR as defined in *EREF*.

The e6500 core has two independent CTRs, one for each thread.

2.7 Processor control registers

This section describes registers associated with identifying and controlling thread and core features. In particular, it describes the following registers:

- Machine State (MSR)
- Machine State Register Protect (MSRP)

- Embedded Processor Control (EPRC)
- Branch Unit Control (BUCSR)
- Hardware Implementation-Dependent 0 (HID0)
- Core Device Control and Status (CDCSR0)
- Power Management Control 0 (PWRMGTCR0)
- Processor Version (PVR)
- System Version (SVR)
- Chip Identification (CIR)
- Shifted CCSRBAR (SCCSRBAR)

2.7.1 Machine State (MSR) register

MSR, shown in Figure 2-1, is used to define the processor state, which includes:

- Enabling and disabling of interrupts and debugging exceptions
- Address translation for instruction and data memory accesses
- Enabling and disabling some functionality
- Controlling whether the processor is in 32-bit or 64-bit mode
- Specifying whether the processor is in supervisor or user mode
- Specifying whether the processor is in hypervisor or guest state

The e6500 core has two independent MSRs, one for each thread. The MSR for each thread controls the machine state for that thread.

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

Certain MSR bits for a thread may be changed in the 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 (MSRP) register.” MSRP is writable only in the hypervisor state. When MSR is written in the guest state, bits protected by set MSRP bits are not written and remain unmodified. All other MSR bits are written with the updated values. An attempt to write the MSRP in the guest–supervisor state results in a hypervisor privilege exception.

Changing CM, 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 CM, GS, or IS can cause an implicit branch because these bits are used to compute the virtual address for instruction translation. Instructions may be fetched and executed from any context and 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 Real Page Number (RPN) fields. For this reason, it is unwise to use **mtmsr** to change these bits. Such changes should only be done using return-from-interrupt-type instructions, which provide the context synchronization atomically with instruction execution.

Guest supervisor

	32	33 34	35	36	37	38	39 … 45	46	47	48	49	50	51	52	53	54	55	56 57	58	59	60	61	62	63
R W	CM		GS	—	UCLE	SPV	—	CE	—	EE	PR	FP	ME	FE0	—	DE	FE1	—	IS	DS	—	PMM	RI	—
Reset	All zeros																							

Figure 2-1. Machine State (MSR) register

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

MSR[CM] controls whether a thread is in 32-bit mode or 64-bit mode. Power ISA defines two methods of a 64-bit implementation providing 32-bit mode. *EREF* provides 32-bit mode in a manner compatible with Power Architecture® processors that implement the server category. *EREF* calls this "hybrid 32-bit mode." In both 32-bit and 64-bit modes, instructions that set a 64-bit register affect all 64 bits. The computational mode controls:

- How the effective address is interpreted
- How CR bits and XER bits are set
- How LR is set by branch instructions in which LK = 1
- How CTR is tested by branch conditional instructions

In both modes, effective address computations use all 64 bits of the relevant registers and produce a 64-bit result. However, in 32-bit mode, the high-order 32 bits of the computed effective address are ignored for the purpose of addressing storage.

When executing in 32-bit mode, the upper 32 bits of the fetch address, effective addresses, DAC*x*, IAC*x*, IVPR, and GIVPR are ignored. When transitioning from 64-bit to 32-bit mode, the upper 32 bits of the fetch address are set to 0, regardless of whether the transition is the result of a return from interrupt instruction or a **mtmsr** instruction.

The e6500 core does not implement the WE bit found in some previous e500 cores. Power management operations on SoCs using the e6500 are handled through the PWRMGTCR0 register and an SoC programming model. See the reference manual for the integrated device for additional details.

2.7.2 Machine State Register Protect (MSRP) register

MSRP 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 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.

The e6500 core implements the MSRP as defined in *EREF*.

The e6500 core has two independent MSRPs, one for each thread.

2.7.3 Embedded Processor Control (EPCR) register

EPCR controls whether certain interrupts are directed to the hypervisor state or to the guest–supervisor state and whether the processor executes in 32-bit or 64-bit mode when an interrupt occurs. It also suppresses debug events when in the hypervisor state.

The e6500 core implements EPCR as defined in *EREF.*

The e6500 core has two independent EPCRs, one for each thread.

2.7.4 Branch Unit Control and Status (BUCSR) register

BUCSR, shown in Figure 2-2, is an e6500-specific register used for general control and status of the branch prediction mechanisms, which include the branch target buffer (BTB), the segment target index cache (STIC), and the segment target address cache (STAC). Writing to BUCSR requires synchronization, as described in Section 3.3.3, “Synchronization requirements.”

The e6500 core has two independent BUCSRs, one for each thread.

NOTE

EREF allows implementations to choose whether BUCSR is shared among threads or private to each thread. Software should take this into account when devising strategies for updating BUCSR.

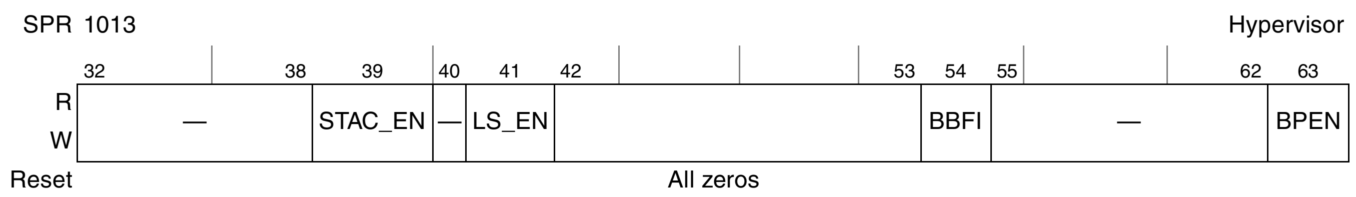


Figure 2-2. Branch Unit Control and Status (BUCSR) register

This table describes the BUCSR fields.

Table 2-7. BUCSR field descriptions

Bits	Name	Description
32–38	—	Reserved
39	STAC_EN	Segment Target Address Cache Enable. When enabled, the segment target address cache predicts the upper 32 bits of branches (that is, a segment is an aligned 4-GB section of effective address space). If disabled, prediction does not occur and branches that occur outside the current 4-GB effective address segment incur a performance penalty. Note that both the STAC and STIC are enabled and disabled by this bit. This bit has no effect if BPEN is not 1. 0 Segment target address cache is disabled. 1 Segment target address cache is enabled.
40	—	Reserved
41	LS_EN	Link Stack Enable. When enabled, the link stack is used to predict function call and return branch target addresses. If disabled, prediction does not occur and function call and return branches are predicted by the BTB. This bit has no effect if BPEN is not 1. 0 Function call and return branch prediction using the link stack is disabled. 1 Function call and return branch prediction using the link stack is enabled.
42–53	—	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 is disabled. 1 Branch prediction is enabled (enables BTB to predict branches).

2.7.5 Hardware Implementation-Dependent 0 (HID0) register

This section describes HID0, shown in Figure 2-3, as it is implemented on the e6500 core.

HID0 is used for configuration and control and is shared by both threads. Writing to HID0 requires synchronization, as described in Section 3.3.3, "Synchronization requirements."

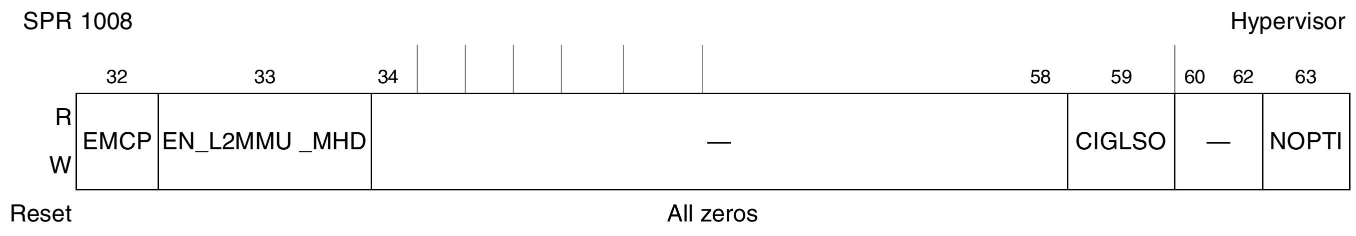


Figure 2-3. Hardware Implementation-Dependent 0 (HID0) register

This table describes the HID0 fields.

Table 2-8. 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 in the L2 MMU or the LRAT 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 or LRAT hit writes 1 to MCSR[L2MMU_MHIT]. If MSR[ME] = 1 or MSR[GS] = 1, a machine check exception and subsequent interrupt occurs.
34–58	—	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	No-Op 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**, **dcbtst**, 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 no-ops. When touch instructions are treated as no-ops 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 no-oped and does not occur. Note that data stream touch and data stream stop (**dss***/**dst***) instructions are always no-oped.

2.7.6 Core Device Control and Status (CDCSR0) register

CDCSR0, shown in Figure 2-4, is implemented as described in *EREF* and is shared by both threads. The e6500 core is aware of the following device programming models:

- Floating-point device—the device is aware, present and ready.
- AltiVec device—the device is aware and present. The device can be transitioned from the following states: Ready or Standby. For more information, see Section 8.6, “AltiVec power down and power up.”

For the e6500 core, writes to CDCSR0 device fields other than the AltiVec device are ignored.

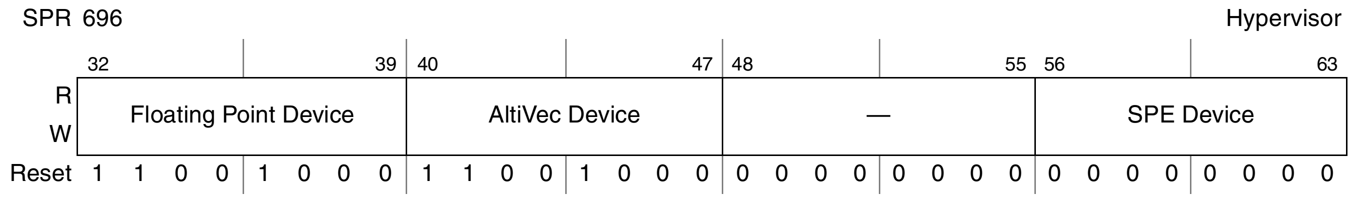


Figure 2-4. Core Device Control and Status 0 (CDCSR0) register

2.7.7 Power Management Control 0 (PWRMGTCR0) register

PWRMGTCR0, shown in Figure 2-5, is shared by both threads and provides fields for software control of specific power management features associated with core power management states. The fields in PWRMGTCR0 associated with AltiVec core device power management (AV_IDLE_PD_EN, AV_IDLE_CNT_P) and the fields associated with the *PW20* core activity state control (PW20_INV_ICACHE, PW20_WAIT, PW20_ENT_P) are implemented as described in *EREF*.

The fields associated with floating-point and SPE core device power management are not implemented.

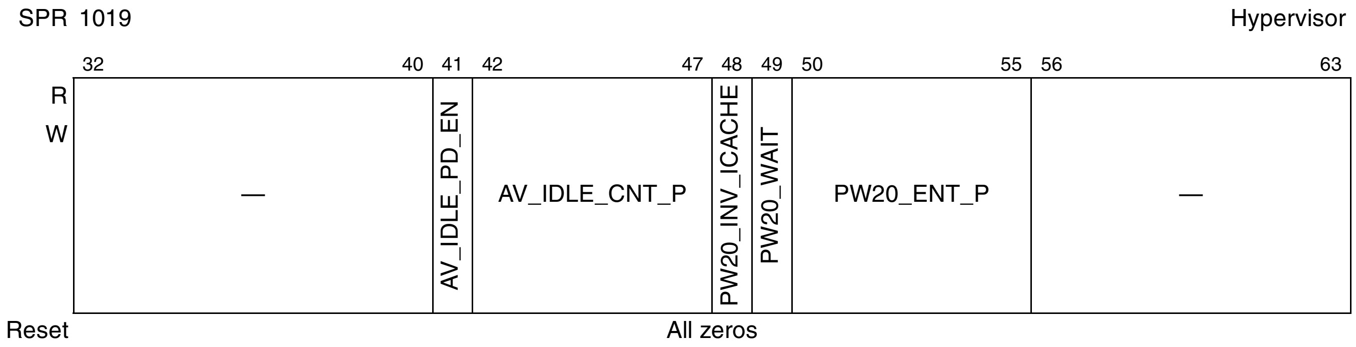


Figure 2-5. Power Management Control 0 (PWRMGTCR0) register

2.7.8 Processor Version (PVR) register

The PVR, shown in Figure 2-6, is shared by both threads (processors), and is implemented as defined by the architecture. The read-only value identifies the version of the core and revision level of the processor, distinguishing between processors with different attributes that may affect software.

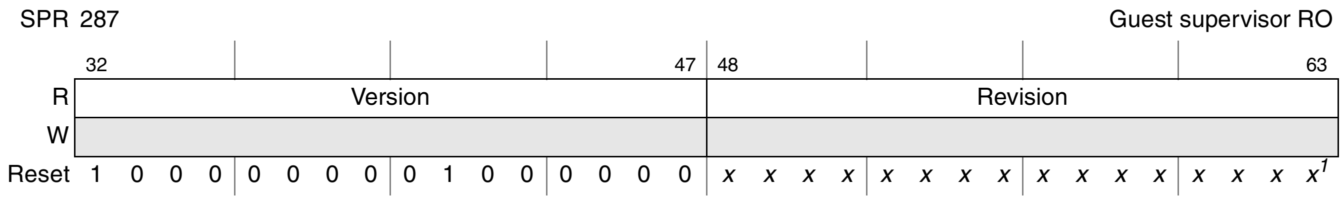


Figure 2-6. Processor Version (PVR) register

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

This table describes the PVR fields.

Table 2-9. 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.

2.7.9 System Version (SVR) register

SVR, shown in Figure 2-7, is shared by both threads and contains a read-only SoC-dependent value. For additional details, see the supporting documentation for the integrated device.

SVR is an alias to the Chip Identification (CIR) register.

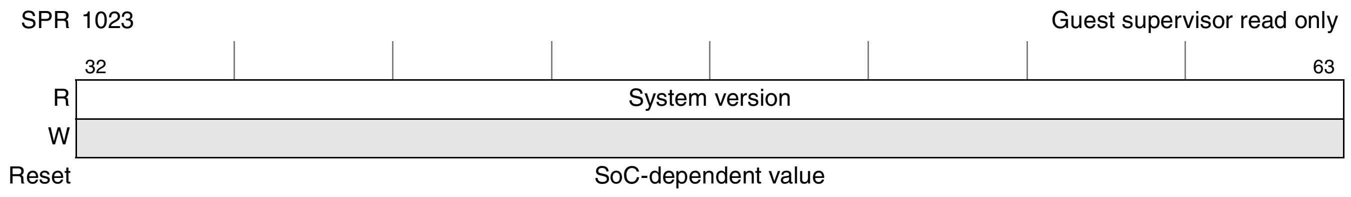


Figure 2-7. System Version (SVR) register

2.7.10 Chip Identification (CIR) register

CIR, shown in Figure 2-8, is shared by both threads and contains a read-only SoC-dependent value. For additional details, see the supported documentation for the integrated device.

CIR is an alias to SVR.

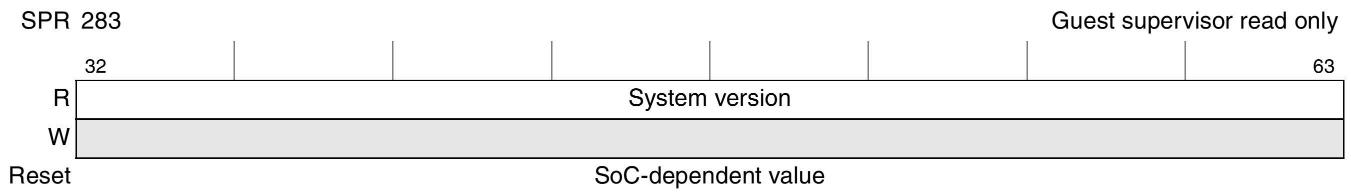


Figure 2-8. Chip Identification Register (CIR)

2.7.11 Shifted CCSRBAR (SCCSRBAR) register

SCCSRBAR, shown in Figure 2-9, is shared by both threads and contains a read-only SoC-dependent value that represents the CCSRBAR value currently in use by the SoC.

The e6500 core implements SCCSRBAR as defined in *EREF*.

This SPR register, when concatenated with 24 bits of 0, represents the value of the CCSRBAR SoC register.

For a description of how SCCSRBAR is interpetted, see the supporting documentation for the integrated device.

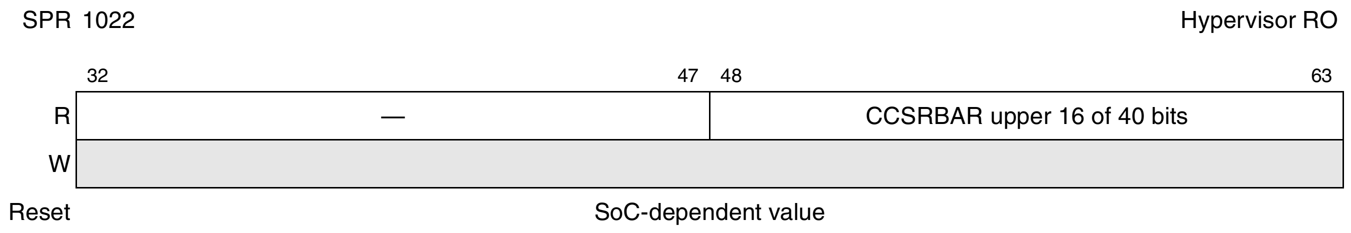


Figure 2-9. Shifted CCSRBAR (SCCSRBAR) register

2.8 Timer registers

The time base (TB), decrementer (DEC), fixed-interval timer (FIT), and watchdog timer provide timing functions for the system. The e6500 core provides the ability to select any of the TB bits to trigger watchdog and fixed-interval timer events, as shown in Figure 2-10.

The e6500 core has two independent sets of decrementer, fixed-interval timer, and watchdog timer, one set for each thread. However, the time base itself is shared by both threads.

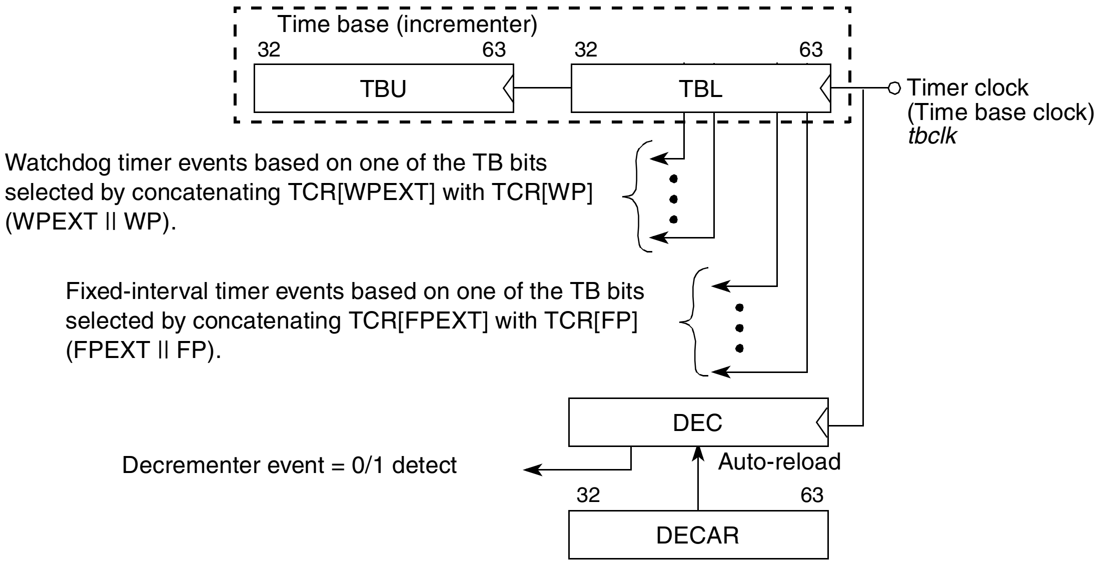

Time base (incrementer)
32
63
32
63
TBU
TBL
Timer clock
(Time base clock)
tbclk
Watchdog timer events based on one of the TB bits selected by concatenating TCR[WPEXT] with TCR[WP] (WPEXT || WP).
Fixed-interval timer events based on one of the TB bits selected by concatenating TCR[FPEXT] with TCR[FP] (FPEXT || FP).
DEC
Decrementer event = 0/1 detect
Auto-reload
32
63
DECAR

Figure 2-10. Relationship of timer facilities to the time base

Note the following characteristics of the e6500 time base implementation:

- The e6500 time base is clocked only by the SoC (TBCLK).
- The only enable/disable control over the time base is the TBEN core signal and when the time base is frozen due to a debug event (see Section 2.14.4, “Debug Control 0 (DBCR0) register”). The time base is controlled by the SoC through a memory-mapped register, allowing control of stopping and starting the time base on any core. See the reference manual for the integrated device, for additional details.
- The **mftb** instruction works as it did in the original PowerPC architecture.

The e6500 registers involved in timing are described as follows:

- The TB is a long-period counter shared by both threads, driven at an implementation-dependent frequency.
- A private DEC for each thread 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.
- A private watchdog timer per thread and a selected TB bit provide 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 (TCR) register

The e6500 core implements TCR as defined in *EREF*. The implementation of the integrated device determines the behavior of TCR[WRC]. For additional details, see the register summary chapter in the core section of the integrated device's reference manual.

The e6500 has two independent TCRs, one for each thread.

2.8.2 Timer Status (TSR) register

The e6500 core implements the TSR as defined in *EREF*. This 32-bit register contains the status of timer events and the most recent watchdog timer-initiated processor reset. All TSR bits function as write-1-to-clear.

The e6500 has two independent TSRs, one for each thread.

2.8.2.1 Watchdog Timer Reset Status (WRS) field

On the e6500 core, TSR[WRS] is nonwriteable (nonclearable) by software. 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 TSR is written by an **mtspr** instruction, WRS bits are not cleared, regardless of the mask bits supplied with GPR used for writing. Logically, the instruction **mtspr** TSR,**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 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.

2.8.2.2 Watchdog Interrupt Status (WIS) and Enable Next Watchdog (ENW) fields

On the e6500 core, when the core is in debug halt mode, the watchdog timer continues to run. However, the watchdog interrupt and watchdog reset are blocked from occurring by holding the TSR[WIS] and TSR[ENW] bits in reset (TSR state 00) while the core is in debug halt mode. When the core exits debug halt mode (to continue software execution), those bits are no longer held in reset, allowing subsequent time-outs to transition the state machine as normal.

2.8.3 Time base registers (TBU and TBL)

The e6500 core implements the time base registers as defined in *EREF*. 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 entire 64 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, setting the upper 32 bits of the destination GPR to 0. Writing the time base is done only through writing the upper 32 bits (SPR 285) and lower 32 bits (SPR 284) using 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. The time base will continue incrementing, if enabled, when the processor is in any core activity state during power management. The time base does not increment when clocks are stopped in the cluster.

The e6500 has one set of time base registers shared among both threads.

2.8.4 Decrementer (DEC) register

The e6500 core implements DEC per thread as defined in *EREF*. DEC is a 32-bit decrementing counter that decrements at the same rate that the time base increments. 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 decrements from 1 to 0. TCR can configure DEC to perform the following actions when it decrements from 1 to 0:

- Stop decrementing
- Auto-reload from DECAR (see Section 2.8.5, “Decrementer Auto-Reload (DECAR) register.”)
- 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).

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

The e6500 core has two independent DECs, one for each thread.

2.8.5 Decrementer Auto-Reload (DECAR) register

The e6500 core implements DECAR as defined in *EREF*. If the auto-reload function is enabled (TCR[ARE] = 1), the auto-reload value in DECAR is written to DEC when it decrements from 1 to 0.

The e6500 core has two independent DECARs, one for each thread.

2.8.6 Alternate time base registers (ATBL and ATBU)

The e6500 core implements the Alternate Time Base (ATB) counter register as defined in *EREF*. ATB is a 64-bit counter that increments at an implementation-dependent frequency. 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 Alternate Time Base Lower (ATBL) register places the entire 64 bits of the time base into the destination GPR. Reading the upper 32 bits of the Alternate Time Base Upper (ATBU) register places the upper 32 bits of the time base into the lower 32 bits of the destination GPR, writing 0 to the upper 32 bits of the destination GPR.

On the e6500 core, the frequency of ATB increment equals the core frequency. ATB is read-only accessible in user and supervisor mode. When the core is in power management states PW20, PH20, or PH30, ATB does not increment. In PH30, the value of ATB will reset to 0 when the core is reset to exit PH30.

The e6500 core has one set of alternate time base counter registers shared among both threads.

2.9 Interrupt registers

This section describes the following register bits and their fields:

2.9.1 Save/restore registers (*x*SRR0/*x*SRR1)

Each thread in the e6500 core implements the following sets of save/ restore registers, which support the different types of interrupts implemented on the e6500 core:

- 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 2.2.2.1, "Register mapping in the guest–supervisor state."

These registers are implemented as defined by the architecture and described in *EREF*.

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; however, for instruction-caused exceptions, it is typically the address of the instruction that causes 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.

On the e6500 core, *x*SRR0 registers are 64-bit registers.

The e6500 core has two independent sets of save/restore registers (*x*SRR0/*x*SRR1), one for each thread.

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

*x*SRR1 is provided to save the 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** instruction 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 (DEAR/GDEAR) registers

Each thread of the e6500 core implements DEAR/GDEAR as defined in *EREF*. DEAR is loaded with the effective address (EA) 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 to DEAR.

GDEAR is supervisor privileged (MSR[PR] = 0) and is read/write. Accesses to DEAR in guest–supervisor state (MSR[GS,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. Therefore, DEAR should be set instead of GDEAR.

On the e6500 core, DEAR/GDEAR are 64-bit registers.

The e6500 core has two independent sets of DEAR/GDEARs, one for each thread.

2.9.3 Logical Page Exception (LPER/LPERU) register

LPER, shown in Figure 2-11, gives information from the page table entry (PTE) that was used to translate a virtual address during a page table translation, which subsequently results in an LRAT error interrupt. The information in LPER is used by software to determine why the LRAT translation failed and to determine how page table management (or LRAT replacement) should proceed.

LPER is a 64-bit, hypervisor-privileged register. LPERU is an alias for the upper 32 bits of LPER.

The e6500 core only implements the low-order 28 bits of the architected 40-bit ALPN field.

The e6500 core has two independent LPERs/LPERUs, one for each thread.

SPR 56 (LPER); 57 (LPERU) Hypervisor

	0 – 23	24 – 51	52 – 56	57 – 59	60 – 63
W	—	ALPN	WIMGE	—	LPS
Reset	All zeros				

Figure 2-11. Logical Page Exception (LPER) register

This table describes the LPER fields.

Table 2-10. LPER Field Descriptions

Bits	Name	Description
0–23	—	Reserved
24–51	ALPN	Abbreviated Logical Page Number. The abbreviated real page number field from the PTE (PTE[ARPN]) that caused the LRAT error interrupt.

Table 2-10. LPER Field Descriptions (continued)

Bits	Name	Description
52–56	WIMGE	WIMGE bits. The WIMGE field from the PTE (PTE[WIMGE]) that caused the LRAT error interrupt.
57–59	—	Reserved
60–63	LPS	Logical Page Size. The logical page size from the PTE (PTE[PS]) that caused the LRAT error interrupt. To convert LPS to a full page size, 0b0 is prepended to LPS to form the page size.

2.9.4 (Guest) Interrupt Vector Prefix (IVPR/GIVPR) registers

The e6500 core implements IVPR and guest IVPR (GIVPR) as defined in *EREF*. These registers are used with Interrupt Vector Offset (IVORs/GIVORs) registers, respectively, to determine the vector address. (G)IVPR[0:47] provides the high-order 48 bits of the address of the exception processing routines. The 16-bit vector offsets (IVORs) are concatenated to the right of IVPR/GIVPR 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 64 bit registers on the e6500 core.

The e6500 core has two independent GIVPRs, one for each thread. The IVPR is shared by both threads.

2.9.5 (Guest) Interrupt Vector Offset (IVORs/GIVORs) registers

The e6500 core implements the IVORs and guest IVORs (GIVORs) as defined in *EREF*. IVORs/GIVORs use only (G)IVOR*n*[48–59], as shown in Figure 2-12, to hold the quad-word index from the base address provided by the IVPR for each interrupt type.

The e6500 core has two independent sets of GIVORs, one for each thread. The IVORs are shared by both threads.

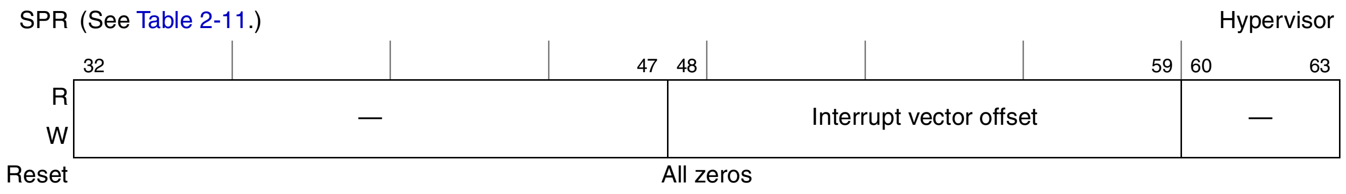


Figure 2-12. (Guest) Interrupt Vector Offset (IVORs/GIVORs) registers

This table lists the (G)IVORs implemented on the e6500 core.

Table 2-11. IVOR assignments

IVOR Number	Interrupt Type
IVOR0	Critical input
IVOR1	Machine check

Table 2-11. IVOR assignments (continued)

IVOR Number	Interrupt Type
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
IVOR14	Instruction TLB error
IVOR15	Debug
IVOR32	AltiVec unavailable
IVOR33	AltiVec assist
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
IVOR42	LRAT error
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
GIVOR35	Guest performance monitor

2.9.6 (Guest) External Proxy (EPR/GEPR) registers

EPR and GEPR are implemented as defined in *EREF*. These registers are 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 4.9.6.1, “External proxy.”

When executing in the guest-supervisor state, any read accesses to the EPR are mapped to GEPR upon executing **mfspr**. See Section 2.2.2.1, “Register mapping in the guest–supervisor state,” for more details.

EPR is not writable; however, GEPR is writable.

The e6500 core has two independent sets of EPR/GEPRs, one for each thread.

2.9.7 (Guest) Exception Syndrome (ESR/GESR) registers

ESR and GESR are implemented as defined by the architecture and described in *EREF*, with the following exception:

- The e6500 core does not implement AP, PUO, VLEMI, MIF, TLBI, or XTE.

Figure 2-13 shows ESR and GESR as they are implemented on the e6500 core. 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 ESR.

When executing in the guest-supervisor state, any accesses to ESR are mapped to GESR upon executing **mtspr** or **mfspr**. See Section 2.2.2.1, “Register mapping in the guest–supervisor state,” for more details.

ESR and GESR provide 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 ESR/GESR bits are cleared. Other interrupt types do not affect ESR/GESR contents. The (G)ESR does not need to be cleared by software. Table 2-12 shows ESR/GESR bit definitions. For machine-check exceptions, the e6500 core uses MCSR, described in Section 2.9.10, “Machine Check Syndrome (MCSR) register.”

The e6500 core has two independent sets of ESR/GESRs, one for each thread.

SPR 62 (ESR); 383 (GESR) — Guest supervisor

	32–35	36	37	38	39	40	41	42	43	44–45	46	47	48–52	53	54	55	56	57	58–63
R/W	—	PIL	PPR	PTR	FP	ST	—	DLK	ILK	—	BO	—	—	DATA	—	PT	SPV	EPID	—
Reset	All zeros																		

Figure 2-13. (Guest) Exception Syndrome (ESR/GESR) registers

This table describes ESR/GESR fields and associated interrupts.

NOTE

ESR/GESR information is incomplete, so system software may need to identify the type of instruction that causes an interrupt, examine the TLB entry, and examine ESR/GESR 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 SRR1/ GSRR1 and the TLB entry page protection bits, to determine whether a protection violation also occurred.

Table 2-12. 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 (DLK0). Set when a DSI occurs because **dcbtls**, **dcbtstls**, or **dcblc** is executed in user mode while MSR[UCLE] = 0.	DSI
43	ILK	Instruction cache locking (DLK1). Set when a DSI occurs because **icbtls** or **icblc** is executed in user mode while MSR[UCLE] = 0.	DSI
44	—	Not supported on the e6500 core. Defined by the architecture as auxiliary processor operation (AP).	—
45	—	Not supported on the e6500 core. Unimplemented operation exception. On the e6500 core, unimplemented instructions are handled as illegal instructions.	Program
46	BO	Byte-ordering exception	DSI, ISI
47	—	Not supported on the e6500 core. Imprecise exception. On the e6500 core, imprecise exceptions are never reported, even when a delayed floating-point-enabled exception occurs.	Program
48–52	—	Reserved	—
53	DATA	Data access. Indicates on an LRAT error exception from a page table translation that the access was a data access and not an instruction fetch.	LRAT error
54	—	Reserved	—
55	PT	Page table translation. Indicates that the exception occurred during a page table translation and no TLB entry was created from the page table translation.	Data storage, Instruction storage, LRAT error

Table 2-12. ESR/GESR field descriptions (continued)

Bits	Name	Syndrome	Interrupt Types
56	SPV	AltiVec Instruction. Indicates that the exception was caused by an AltiVec instruction.	Data storage, Data TLB, AltiVec unavailable, AltiVec assist
57	EPID	External PID instructions. 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, or Alignment error
58–63	—	Reserved	—

2.9.8 (Guest) Processor ID (PIR/GPIR) registers

The e6500 core implements PIR/GPIR as defined in *EREF*. The processor sets the initial value of PIR at reset driven from signal pins from the SoC, after which it is writable by hypervisor software. The initial value of the PIR is a processor-unique value within the coherence domain and is described in *EREF*. 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. **mtspr** accesses are not mapped, and guest supervisor attempts to change PIR or GPIR cause an embedded hypervisor privilege interrupt. See Section 2.2.2.1, “Register mapping in the guest–supervisor state,” for more details.

The e6500 has two independent sets of PIR/GPIRs, one for each thread.

2.9.9 Machine-check address registers (MCAR/MCARU/MCARUA)

When a thread takes a machine-check interrupt, MCAR may indicate the address of the data associated with the machine check exception. MCAR is a 64-bit address and may contain a logical address, real (physical) address, or an effective address. MCARUA and MCARU are 32-bit aliases to the upper 32 bits of MCAR. 32-bit software should use MCARUA to address the upper 32 bits. MCARU is provided for compatibility with older processors. 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.

The e6500 core has two independent sets of MCAR/MCARU/MCARUAs, one for each thread.

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

Table 2-13. 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 is unmodified if currently valid (hold value if already valid).
0	1	0	MCAR[24–63]	Updated with a logical (in the case of a LRAT multi-way hit) or real (physical) address.
0	1	1	MCAR[0–63]	Updated with the EA associated with the error. If the detected error is a multi-way hit in the L2MMU (MCSR[L2MMU_MHIT]), the lower 12 bits of the EA are cleared, providing an EPN for the translation.

2.9.10 Machine Check Syndrome (MCSR) register

In addition to the MCSR fields defined in *EREF*, the e6500 core implements a number of other implementation-specific fields, as shown in Table 2-14. When a thread in the core takes a machine-check interrupt, it updates its MCSR to differentiate between machine check conditions. MCSR indicates the type of error detected. Software can use this information to determine whether the error is recoverable and what steps may be necessary to correct the error.

The e6500 has two independent MCSRs, one for each thread.

MCSR is shown in the following figure.

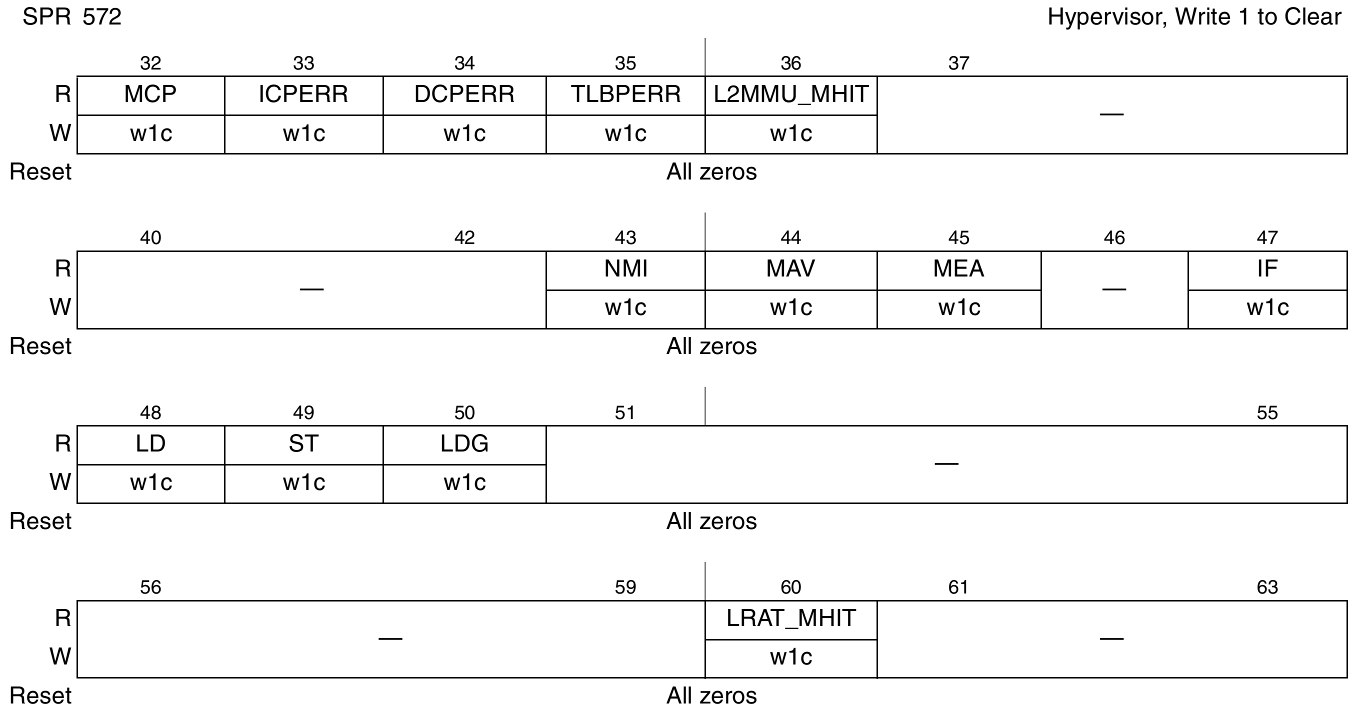


Figure 2-14. Machine Check Syndrome (MCSR) register

This table describes the MCSR fields.

Table 2-14. MCSR field descriptions

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	ICPERR	Instruction cache tag or data array parity error	Async	L1CSR1[ICECE] and L1CSR1[ICE]
34	DCPERR	Data cache data or tag parity error due to a load	Async	L1CSR0[CECE] and L1CSR0[DCE]
35	TLBPERR	TLB0 array parity error	Async	—
36	L2MMU_MHIT	L2 MMU simultaneous hit. Multi-way hits in the LRAT are also reported using this bit.	Async	HID0[EN_L2MMU_MHD]
37–42	—	Reserved	—	—
43	NMI	Nonmaskable interrupt	NMI	None
44	MAV	MCAR address valid. The address contained in the MCAR is updated by the processor and corresponds to the first detected error condition that contains 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 writing 1 to MSR[ME].	Status	—
45	MEA	MCAR effective address. Meaningful only if MAV=1. 0 The MCAR contains a logical or 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

Table 2-14. MCSR field descriptions (continued)

Bit	Name	Description	Exception Type[1]	Additional Gating Condition[2]
51–59	—	Reserved	—	—
60	LRAT_MHIT	LRAT translation during a **tlbwe** instruction hit in more than one entry	Async	HID0[EN_L2MMU_MHD]
61–63	—	Reserved	—	—

[1] "Exception Type" indicates which of the following exception types causes 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 of this particular error condition.

The settings of MCSR[LD] and MCSR[ST] that identify the type of instruction are implementation dependent. For the e6500 core, 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 depending on whether LD or ST is set for an error report.

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

> **dcbt**, **dcbtst**, **icbt**, **lbz**, **lbzu**, **lbzx**, **lbzux**, **ld**, **ldarx**, **ldbrx**, **lddx**, **ldepx**, **ldu**, **ldux**, **ldx**, **lha**, **lhau**, **lhax**, **lhaux**, **lhz**, **lhzu**, **lhzx**, **lhzux**, **lhbrx**, **lmw**, **lwa**, **lwarx**, **lwaux**, **lwax**, **lwz**, **lwzu**, **lwzx**, **lwzux**, **lwbrx**, **lbepx**, **lhepx**, **lwepx**, **dcbtep**, **dcbtstep**, **lbdx**, **lhdx**, **lwdx**, **lfddx**, **lfd**, **lfdu**, **lfdux**, **lfdx**, **lfdepx**, **lfs**, **lfsu**, **lfsux**, **lfsx, lvebx, lvehx, lvepx, lvepxl, lvewx, lvexbx, lvexhx, lvexwx, lvtlx, lvtlxl, lvtrx, lvtrxl, lvswx, lvswxl, lvx, lvxl**

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**, **std**, **stdbrx**, **stdcx.**, **stddx**, **stdepx**, **stdu**, **stdux**, **stdx**, **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, stvebx, stvehx, stvepx, stvepxl, stvewx, stvexbx, stvexhx, stvexwx, stvflx, stvflxl, stvfrx, stvfrxl, stvswx, stswxl, stvx, stvxl**

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

The e6500 core implements the software-use SPRs (SPRG0–SPRG7, SPRG8, SPRG9, GSPRG0-GSPRG3, USPRG0) as defined in *EREF*.

The e6500 core has two independent sets of software-use SPRs, one for each thread.

Their functionality is defined by the user and they are accessed as shown in the following table.

Table 2-15. 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
SPRG2	SPR General 2	274	Guest supervisor
SPRG3	SPR General 3	259	User RO[1]
SPRG3	SPR General 3	275	Guest supervisor
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	SPR General 8	604	Hypervisor
SPRG9	SPR General 9	605	Guest supervisor
USPRG0 (VRSAVE)	User SPR General 0[2]	256	User

[1] When these registers are accessed in the guest-supervisor state, the accesses are mapped to their analogous guest SPRs (for example, SPRG0 is mapped to GSPRG0). See Section 2.2.2.1, "Register mapping in the guest–supervisor state."

[2] USPRG0 is a separate physical register from SPRG0.

Operating system software should always use SPRG0, SPRG1, SPRG2, SPRG3 when accessing GSPRG0, GSPRG1, GSPRG2, and GSPRG3 because, in the 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 the e6500 core, these registers are 64 bits. USPRG0 (VRSAVE) is a 32-bit register regardless of whether the processor is a 32-bit or 64-bit implementation.

2.11 L1 cache registers

The L1 cache registers provide control, configuration, and status information for the L1 cache implementation. These registers are shared by the e6500 core threads.

2.11.1 L1 Cache Control and Status 0 (L1CSR0) register

L1CSR0 is used for general control and status of the L1 data cache. The e6500 core implements L1CSR0 fields as defined in *EREF*, except for the following:

- Cache way partitioning bits—L1CSR0[32–42]
- Data cache lock overflow allocate (CLOA) bit—L1CSR0[56]
- Cache snoop lock clear (CSLC) bit—L1CSR0[52]. Cache locking is persistent.
- Cache unable to lock (CUL) bit—L1CSR0[53]. Cache lock status can be queried with the **dcblq.** instruction.
- Cache operation aborted (CABT) bit—L1CSR0[61]. Cache operations are never aborted on e6500.

For L1CSR0[CEA], the e6500 core 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 L1CSR0[CECE] = 1. Any other value written to this field is ignored.

The e6500 core only supports L1CSR0[CEDT] = 0b00 for parity detection on data arrays and tags and supports L1CSR0[CEIT] = 0b00 for setting the cache error injection type to inject single bit data error. Any other values written to CEDT and CEIT are ignored.

Note that on the e6500 core, when writing 1 to L1CSR0[CEI], it is required that L1CSR0[CECE] also be set with the same **mtspr** instruction. If L1CSR0[CECE] is not set, the processor will clear L1CSR0[CEI].

After the L1 data cache has been enabled, if L1CSR0[CE] = 0 (that is, the L1 data cache is disabled), any stashing to the L1 data cache must first be disabled by writing 0 to L1CSR2[DCSTASHID] and performing the appropriate synchronization.

Writing to L1CSR0 requires isolated shared synchronization, as described in Section 3.3.3, "Synchronization requirements."

2.11.2 L1 Cache Control and Status 1 (L1CSR1) register

L1CSR1 is used for general control and status of the L1 instruction cache. The e6500 core implements the L1CSR1 fields as they are defined in *EREF*, except for the following:

- Instruction cache lock overflow allocate (ICLOA) bit—L1CSR1[56]
- Instruction cache unable to lock (ICUL) bit—L1CSR1[53]. Cache lock status can be queried with the **icblq.** instruction.
- Instruction cache snoop lock clear bit, ICSLC, (L1CSR1[52]). Cache locking is persistent.

- Cache operation aborted (ICABT) bit—L1CSR1[61]. Cache operations are never aborted on e6500.

For L1CSR1[ICEA], the e6500 core 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] = 1. Any other value written to this field is ignored.

Only implementation-specific error detection type (ICEDT = 0b00), parity detection on data and tags, is supported for the e6500 core. Only single-bit error injection type (ICEIT = 0b00) is supported on the e6500 core. Any other values written to ICEDT and ICEIT are ignored.

Note that on the e6500 core, when writing 1 to L1CSR1[ICEI], it is required that L1CSR0[ICECE] also be set with the same **mtspr** instruction. If L1CSR1[ICECE] is not set, the processor will clear L1CSR1[ICEI].

The e6500 core has one L1CSR1 shared by both threads.

Writing to L1CSR1 requires isolated shared synchronization, as described in Section 3.3.3, “Synchronization requirements.”

2.11.3 L1 Cache Control and Status 2 (L1CSR2) register

L1CSR2 provides additional control and status for the primary L1 data cache of the processor. The e6500 core implements L1CSR2 as defined in *EREF*, with the following exceptions:

- Data cache write shadow, DCWS, (L1CSR2[33]) is not implemented. Writing to the L1 data cache is always written through to the shared backside L2 cache.
- Although the architecture defines DCSTASHID as L1CSR2[54–63], the e6500 core implements only 8 bits (L1CSR2[56–63]) and supports only stash ID values of 8 to 255.

The e6500 has one L1CSR2 shared by both threads.

Writing to L1CSR2 requires synchronization, as described in Section 3.3.3, “Synchronization requirements.”

2.11.4 L1 Cache Configuration 0 (L1CFG0) register

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

SPR 515 User RO

	32–33	34	35	36	37–38	39–40	41–42	43	44	45–52	53–63
R	CARCH	CWPA	CFAHA	CFISWA	—	CBSIZE	CREPL	CLA	CPA	CNWAY	CSIZE
W											
Reset	0 0	0	0	0	0 0	0 1	1 1	1	1	0 0 0 0 0 1 1 1	0 0 0 0 0 1 0 0 0 0 0

Figure 2-15. L1 Cache Configuration 0 (L1CFG0) register fields implemented on the e6500

EREF describes the L1GCFG0 fields as they are defined in the architecture. The following table describes how they are implemented on the e6500 core.

Table 2-16. 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. 3 indicates FIFO 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 eight ways.
53–63	CSIZE	Cache size. 32 indicates 32 KB.

The e6500 core has one L1CFG0 shared by both threads.

2.11.5 L1 Cache Configuration 1 (L1CFG1) register

L1CFG1, shown in Figure 2-16, provides configuration information for the L1 instruction cache.

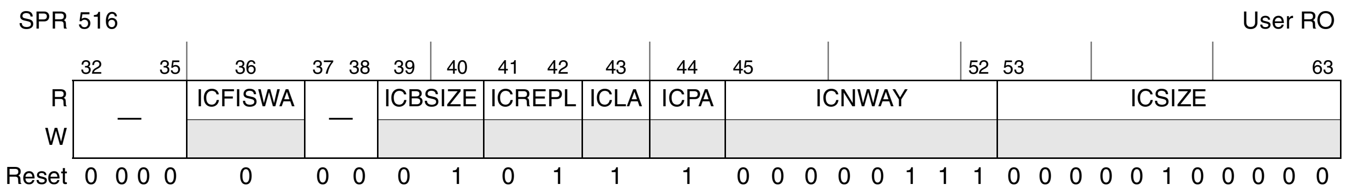


Figure 2-16. L1 Cache Configuration 1 (L1CFG1) register

This table describes the L1CFG1 fields.

Table 2-17. 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.

Table 2-17. L1CFG1 field descriptions (continued)

Bits	Name	Description
45–52	ICNWAY	Instruction cache number of ways. 7 indicates eight ways.
53–63	ICSIZE	Instruction cache size. 32 indicates 32 KB.

The e6500 core has one L1CFG1 shared by both threads.

2.12 L2 cache registers

L2 cache status, control, and error handling is accomplished through MMRs. Shared L2 configuration and control uses the same general formats as the integrated backside L2 cache provided in previous Freescale cores, although those controls were performed through SPRs.

2.12.1 L2 Configuration 0 (L2CFG0) register

L2CFG0 is provided for software to determine the organization and capabilities of the secondary cache. The e6500 core implements L2CFG0 as defined by the architecture and described in *EREF*.

L2CFG0, shown in Figure 2-17, provides configuration information for the L2 cache.

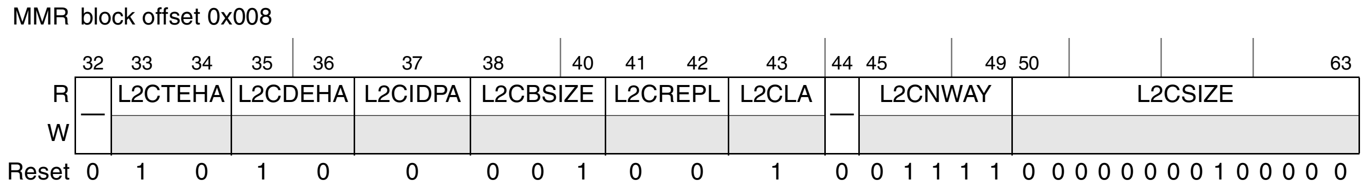


Figure 2-17. L2 Cache Configuration 0 (L2CFG0) register

This table provides the L2CFG0 field descriptions.

Table 2-18. L2CFG0 field descriptions

Bits	Name	Description
32	—	Reserved
33–34	L2CTEHA	L2 cache tags error handling available. 0b10 indicates single-bit ECC correction, double-bit ECC detection is available.
35–36	L2CDEHA	L2 cache data error handling available. 0b10 indicates single-bit ECC correction, double-bit ECC detection is available.
37	L2CIDPA	Cache instruction and data partitioning available. 0 indicates not available.
38–40	L2CBSIZE	Cache line size. 1 indicates 64 bytes.
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. 0 indicates streaming pseudo-LRU.
43	L2CLA	Cache line locking available. 1 indicates available.
44	—	Reserved

Table 2-18. L2CFG0 field descriptions (continued)

Bits	Name	Description
45–49	L2CNWAY	Number of cache ways minus one. 15 indicates 16 ways.
50–63	L2CSIZE	Cache size as a multiple of 64 KB. 32 indicates 2048 KB cache.

2.12.2 L2 Cache Control and Status 0 (L2CSR0) register

L2CSR0, shown in Figure 2-18, provides general control and status for the L2 cache of the processor. The e6500 core implements L2CSR0 as defined by the architecture and described in *EREF,* with the following exceptions:

- It does not implement the following fields: L2WP, L2CM, L2IO, L2DO, L2FCID. Note that these fields are notated in parentheses in Table 2-19.

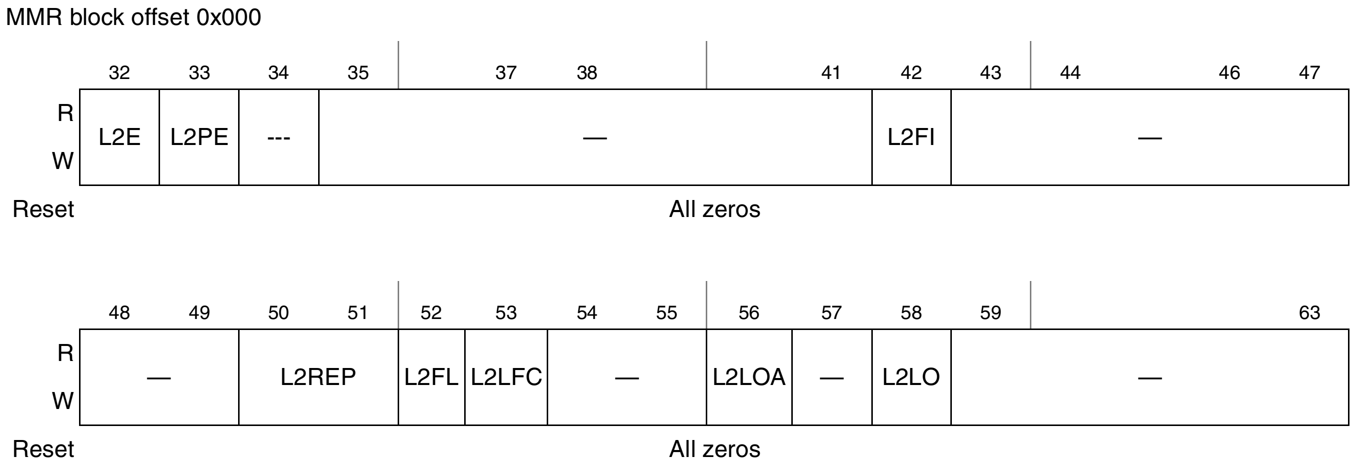


Figure 2-18. L2 Cache Control and Status 0 (L2CSR0) register

This table describes the L2CSR0 fields.

Table 2-19. L2CSR0 field descriptions

Bits	Name	Description
32	L2E	L2 cache enable. Implemented as defined in *EREF*. The e6500 core requires software to continue to read this bit after setting it to ensure the desired value has been set before continuing. **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.
33	L2PE	L2 cache parity/ECC error checking enable. Implemented as defined in *EREF*. **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 11.7, "L2 cache state," for more details on L2 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. **Note:** The value of L2PE must not be changed while the L2 cache is enabled.
34	—	Reserved
35–37	(L2WP)	L2 instruction/data way partitioning. This field is not implemented in the e6500 core and always reads as 0.
38–39	(L2CM)	L2 cache coherency mode. This field is not implemented in the e6500 core and always reads as 0.
40–41	—	Reserved
42	L2FI	L2 cache flash invalidate. Implemented as defined in *EREF*. 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:** Writing a 1 during any sequential operation causes undefined results. Writing a 0 during an invalidation operation is ignored. **Note:** If L2FI and L2LFC are set with the same register write operation, then the flash invalidate and the lock flash clear functions are performed simultaneously.
43	(L2IO)	L2 cache instruction only. This field is not implemented in the e6500 core and always reads as 0. Similar functionality can be accomplished using L2 cache partitioning, which is described in Section 2.12.4, "L2 cache partitioning registers."
44–46	—	Reserved
47	(L2DO)	L2 cache data only. This field is not implemented in the e6500 core and always reads as 0. Similar functionality can be accomplished using L2 cache partitioning, which is described in Section 2.12.4, "L2 cache partitioning registers."
48–49	—	Reserved

Table 2-19. L2CSR0 field descriptions (continued)

Bits	Name	Description
50-51	L2REP	L2 line replacement algorithm. 00 Streaming Pseudo Least Recently Used (SPLRU) 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 updated to an intermediate state between least recently used and most recently used on most L2 cache allocations and to the most recently used state on the remainder of L2 cache allocations. 01 First-in-first-out (FIFO). 10 Streaming Pseudo Least Recently Used (SPLRU). 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 updated to an intermediate state between least recently used and most recently used on all L2 cache allocations. 11 Pseudo Least Recently Used (PLRU). 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*. On the e6500 core, 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. **Note:** To flush the L2 cache and ensure that no valid entries exist after the flush, the following sequence should be used: Clear all the bits of L2PAR0 - L2PAR3 to prevent further allocations. Read L2PAR0 - L2PAR3 to ensure that the changes are in effect. Write 1 to L2CSR0[L2FL]. Continue to read L2CSR0[L2FL] until it reads 0.
53	L2LFC	L2 cache lock flash clear. On boot, the processor should set this bit to clear any lock state bits that may be randomly set out of reset, prior to enabling the L2 cache.
54–55	(L2FCID)	Not implemented on the e6500 core. L2LFC lock clearing always behaves as if L2FCID = 0b11 and all locks are cleared.
56	L2LOA	L2 cache lock overflow allocate. Implemented as defined in *EREF*. Note that cache line locking in the e6500 L2 is persistent.
57	—	Reserved
58	L2LO	L2 cache lock overflow. Implemented as defined in *EREF*.
59–63	—	Reserved

2.12.3 L2 Cache Control and Status 1 (L2CSR1) register

L2CSR1, shown in Figure 2-19, provides general control and status for the L2 cache of the processor. The e6500 core implements L2CSR1 as defined by the architecture and described in *EREF,* with the following exceptions:

- It implements only the 8 least significant bits of the L2STASHID (L2CSR1[L2STASHID]).
- It does not support stash ID values less than eight.

In addition, it implements the implementation-specific fields DYNAMICHARVARD, L2BHM, and L2STASHRSRV.

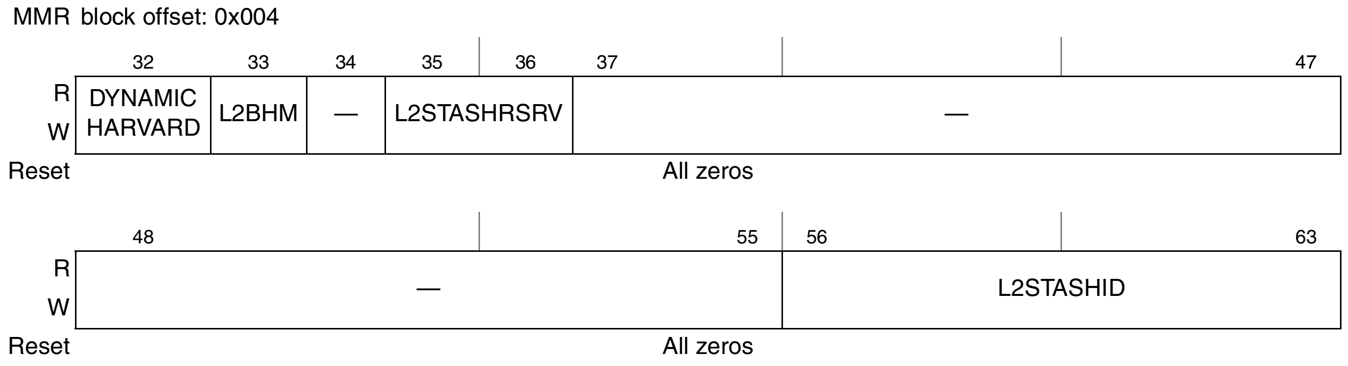


Figure 2-19. L2 Cache Control and Status 1 (L2CSR1) register

This table describes the L2CSR1 fields.

Table 2-20. L2CSR1 e6500-specific field descriptions

Bits	Name	Description
32	DYNAMICHARVARD	Dynamic Harvard Mode 0 Enabled. Cacheable instruction fetches requested by the processor that miss are requested from CoreNet as non coherent (Memory coherence required = 0). When the line is allocated it is marked to allow a hit from instruction fetches but not data accesses. 1 Disabled. Cacheable instruction fetches requested by the processor that miss are requested from CoreNet as coherent (Memory coherence required = 1). When the line is allocated it is marked to allow a hit from instruction fetches and data accesses.
33	L2BHM	Bank Hash Mode 0 Use decode hash (bits 56:57 of real address). 1 Use XOR hash (bits 42:57 of real address).
34	—	Reserved
35–36	L2STASHRSRV	L2 Stashing Reserved Resources. The number of resources per bank in which to allocate only stashes. 00 Allocate opportunistically with general resources (default) 01 One resource 10 Two resources 11 Three resources

Table 2-20. L2CSR1 e6500-specific field descriptions (continued)

Bits	Name	Description
37–55	—	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 e6500 supports only stash ID values of 8 and larger (that is values between 8 and 255); values from 1 to 7 are illegal.

2.12.4 L2 cache partitioning registers

L2PIR*n*, L2PAR*n*, and L2PWR*n* are sets of registers that are used to define how individual transactions performed by the L2 cache are allocated. The number of registers *n* may vary between implementations, but for any given value *n* supported by an implementation, the same number of registers exist for L2PIR, L2PAR, and L2PWR. The number of registers *n* implemented represents the number of different allocation policies that can be applied at any given time.

Each transaction sent to the L2 cache by a processor is tagged with an identifier. The identifier is used to distinguish which allocation policies should be used when the L2 cache processes transactions. On the e6500 core, the identifier is set to the value of the Processor ID register PIR[59:63] at reset. The identifiers for L2 cache partitioning do not change if software changes the value in PIR. This allows for a unique identifier for each core and thread in the cluster. Software can change the identifier by changing the PIR register.

Let *ID* be the identifier for a transaction presented to the L2 cache and *n* be the number of different allocation policies implemented (that is, the number *n* of registers implemented). *id* + 32 corresponds to a column of bits in the L2PIR*n* registers and is used to determine which allocation policies are to be applied as follows:

```
bit_num ← ID + 32
ways ← 0

for reg_num = 0 to n - 1
   if L2PIR[reg_num]$_{bit\_num}$ = 1 | stash then
       policy ← L2PAR[reg_num]
       if instruction fetch & policy$_{IRDALLOC}$ then
           ways ← ways | L2PWR[reg_num]
       else if data read & policy$_{DRDALLOC}$ then
           ways ← ways | L2PWR[reg_num]
       else if data store & policy$_{DSTALLOC}$ then
           ways ← ways | L2PWR[reg_num]
       else if stash & policy$_{STALLOC}$ then
           ways ← ways | L2PWR[reg_num]
endfor
if ways ≠ 0 then
   allocate line in ways
else
   line is not allocated
```

L2PIR*n* maps a possible set of 32 identifiers to specific allocation policies. L2PAR*n* and L2PWR*n* are used to process the allocation. L2PAR*n* determines what allocation policy is used. L2PWR*n* determines in which ways the allocation may occur.

L2PIR*n*, L2PAR*n*, and L2PWR*n* are initialized by hardware at reset to allow all types of accesses from any identifier to allocate in any way (that is, any request that could perform an allocation does allocate and can be allocated in any way).

Note that stash transactions that are targeted to the L2 or any of the processor L1 caches in the cluster do not provide identifiers for the purpose of determining allocation policy and way selection. Instead, stashes behave as if the identifier for the transaction has bits set in all of L2PIR*n*. Stashes that are targeted to the L1 cache of a processor, but that cannot allocate in the L2 cache because of the setting of L2PAR*n*[STALLOC] (or other reasons), are invalidated in the processor's L1 cache.

L2 cache partitioning only affects when a line in the cache may be allocated or not and in which ways it may be allocated. Transactions to the L2 cache that do not require allocation (for example, a load operation to an address that is present in the L2 cache) are unaffected by the settings of L2PIR*n*, L2PAR*n*, and L2PWR*n*.

For the e6500 core, the ID for partitioning transactions to the L2 cache is dependent on the integrated device; however, if the integrated device initializes PIR as defined by EREF (for each L2 cache), then the ID for partitioning in a four core cluster is as listed in the following table.

Table 2-21. L2 cache partitioning identifiers

ID	Core	Thread
0b00000	0	0
0b00001	0	1
0b01000	1	0
0b01001	1	1
0b10000	2	0
0b10001	2	1
0b11000	3	0
0b11001	3	1

Integrated devices with less than four cores per L2 cache cluster do not use the IDs for cores that are not present on the cluster. In all cases, the core number is encoded in the first 2 bits and the thread number in the core is encoded in the lower 3 bits.

Partitioning the L2 cache can prevent one processor from victimizing lines established by other processors. This may be important to protect lines established by a processor that may be running a real-time application that needs a more predictable performance characteristic and can be programmed to limit how many lines can be allocated by other processors by choosing the ways in the cache that each processor can allocate into. To accomplish this, each processor can be assigned the L2 cache ways in which it will allocate.

To partition the L2 cache ways, first decide which L2 cache ways and what types of accesses are desired to allocate in those L2 cache ways. Each distinct set of these should be considered a partitioning policy. Each of these distinct policies should be encoded into a set of L2PAR*x* and L2PWR*x* registers. The

L2PAR*x* register contains the types of accesses that are allowed to allocate for this policy, and the L2PWR*x* register contains the list of L2 cache ways to which these accesses can allocate. For example, a policy that only allows allocation to L2 cache ways 0 and 1 for loads and stores is encoded as:

L2PAR*x* = 0x00000440
L2PWR*x* = 0xC0000000

The bits set in the L2PAR*x* register reflect the types of access. To change this policy to also have instruction fetches and stashes use this policy, L2PAR*x* changes to 0x000004C1.

Once policies are established, the designation of which processor uses those policies should be encoded in L2PIR*x* registers, where the ID of the processor is used to index a bit in the L2PIR*x* register. Using the previous example, to have core 0, thread 1 and core 1, thread 0 use the policy to only allow allocation to L2 cache ways 0 and 1 for loads and stores the L2PIRx, L2PAR*x*, L2PWR*x* register triple should be set to:

L2PIR*x* = 0x40800000
L2PAR*x* = 0x00000440
L2PWR*x* = 0xC0000000

The bit indexing using the ID is done directly on the 32 bits of the register (when all bits in the register are numbered from 0 to 31). Because all registers are documented using 64-bit notation, the 64-bit index is ID+32.

Note that each distinct policy uses a set of L2PIR*x*, L2PAR*x*, L2PWR*x* registers. L2PIR0, L2PAR0, and L2PWR0 define a policy and which processors use that policy. Similarly, there are 7 more policies that can be defined using the other L2PIR*x*, L2PAR*x*, L2PWR*x* registers.

Note also that a policy does not explicitly deny allocation into L2 cache ways, but allows allocation into L2 cache ways. The full allocation policy for a given transaction from a given processor is the logical OR of all the policies that have the appropriate processor ID bit set in the L2PIR*x* for the policy. Care should be taken to ensure that all processor IDs have at least one policy that allows them to allocate into L2 cache ways unless it is desired that those processors should not allocate any lines in the L2 cache.

2.12.4.1 L2 cache partitioning identification registers (L2PIR*n*)

L2PIR*n*, shown in Figure 2-20, provides controls for partitioning the L2 cache based on identifiers attached to the L2 cache transactions from processors. L2PIR*n* is a set of registers, each containing a bit vector of 32 bits. The identifier sent with each transaction to the L2 cache is used to select the same relative bit (*id* + 32) in each of the L2PIR*n* registers. If a bit is set in a L2PIR*n* register, then that register number is used to index among the allocation policies represented by L2PAR*n* and L2PWR*n*.

If more than one bit for each identifier is set among the group of L2PIR*n* registers, the allocation policy used is the logical OR of the corresponding L2PAR*n* registers, and the ways available for allocation is the logical OR of the L2PWR*n* registers for which the corresponding L2PAR*n* registers allow allocation. For example, if L2PIR0[35] = 1 and L2PIR1[35] = 1, then the allocation policy is L2PAR0 | L2PAR1 and the ways available for allocation are defined by the OR of the L2PWR registers that correspond to the L2PAR register that allows the allocation.

The e6500 implements L2PIR0—L2PIR7 as defined by the architecture and described in *EREF*.

Writing to these registers requires synchronization.

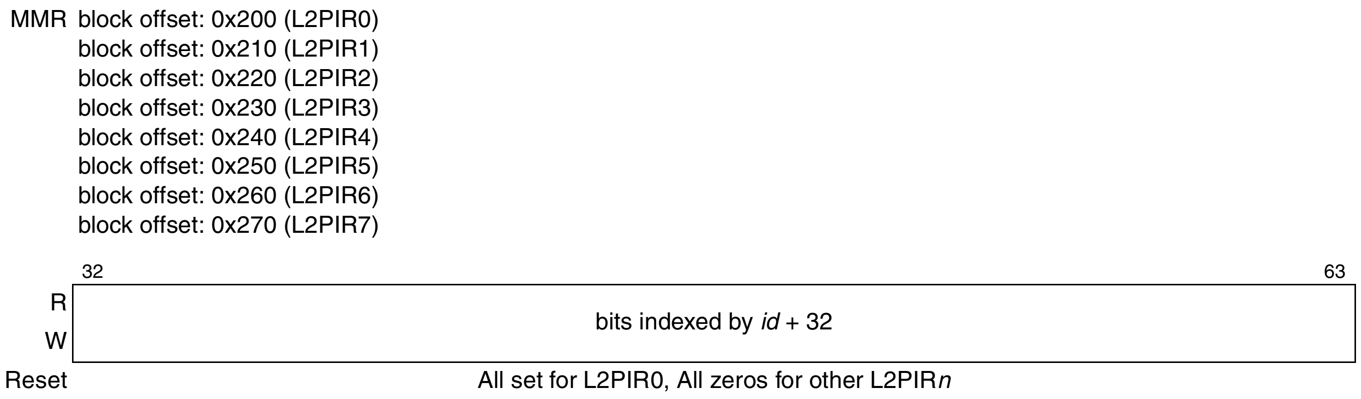

MMR block offset: 0x200 (L2PIR0)
block offset: 0x210 (L2PIR1)
block offset: 0x220 (L2PIR2)
block offset: 0x230 (L2PIR3)
block offset: 0x240 (L2PIR4)
block offset: 0x250 (L2PIR5)
block offset: 0x260 (L2PIR6)
block offset: 0x270 (L2PIR7)
32
63
R
W
bits indexed by *id* + 32
Reset
All set for L2PIR0, All zeros for other L2PIR*n*

Figure 2-20. L2 cache partitioning identification registers (L2PIR*n*)

2.12.4.2 L2 cache partitioning allocation registers (L2PAR*n*)

L2PAR*n*, shown in Figure 2-21, provides controls for partitioning the L2 cache based on which allocation policy is determined from the L2PIR*n* registers. If the bit associated with an *id* of a transaction sent to the L2 cache is set in one of the L2PIR*n* registers, then that register number (0 - *n*) is used to index among the allocation policies represented by L2PAR*n* and L2PWR*n*.

L2PAR*n* controls whether a line should be allocated based on the type of transaction to be performed by the L2 cache. The types of distinguished transactions are:

- Store type operations (store, store conditional, or **dcbz**[**l**][**ep**])
- Load type operations (load, touch, and lock set)
- Instruction fetch
- Stash operations targeted to the L2 cache

The e6500 core implements L2PAR0—L2PAR7 as defined by the architecture and described in *EREF*.

Writing to these registers requires synchronization.

MMR block offset: 0x208 (L2PAR0)
block offset: 0x218 (L2PAR1)
block offset: 0x228 (L2PAR2)
block offset: 0x238 (L2PAR3)
block offset: 0x248 (L2PAR4)
block offset: 0x258 (L2PAR5)
block offset: 0x268 (L2PAR6)
block offset: 0x278 (L2PAR7)

	32–47
R/W	—
Reset	All zeros

	48–52	53	54–55	56	57	58–62	63
R/W	—	DSTALLOC	—	IRDALLOC	DRDALLOC	—	STALLOC
Reset	All zeros	1	All zeros	1	1	All zeros	1

Figure 2-21. L2 cache partitioning allocation registers (L2PAR*n*)

This table describes the L2PAR*n* fields.

Table 2-22. L2PAR*n* field descriptions

Bits	Name	Description
32–52	—	Reserved, should be 0
53	DSTALLOC	Data store allocation control 0 Cacheable store and store conditional instructions that miss in the L2 do not allocate unless enabled by another L2PAR*n*[DSTALLOC]. 1 Cacheable store and store conditional instructions that miss in the L2 attempt to allocate in one of the ways defined by L2PWR*n*[WAY].
54–55	—	Reserved, should be 0
56	IRDALLOC	Instruction read (fetch) allocation control 0 Cacheable instruction fetches that miss in the L2 do not allocate unless enabled by another L2PAR*n*[IRDALLOC]. 1 Cacheable instruction fetches that miss in the L2 attempt to allocate in one of the ways defined by L2PWR*n*[WAY].
57	DRDALLOC	Data read allocation control 0 Cacheable load and touch instructions that miss in the L2 do not allocate unless enabled by another L2PAR*n*[DRDALLOC]. 1 Cacheable load and touch instructions that miss in the L2 attempt to allocate in one of the ways defined by L2PWR*n*[WAY]. **Note:** Any cache locking operation with CT = 2 that has DRDALLOC = 0 will not have the line locked because the L2 does not attempt to allocate the line.

Table 2-22. L2PAR*n* field descriptions (continued)

Bits	Name	Description
58–62	—	Reserved, should be 0
63	STALLOC	Stashing allocation control 0 Stash requests that miss in the L2 do not allocate unless enabled by another L2PAR*n*[STALLOC]. 1 Stash requests that miss in the L2 attempt to allocate in one of the ways defined by L2PWR*n*[WAY]. **Note:** Stash requests do not supply *id* values that index into L2PIR*n* registers, but instead examine all L2PARn registers to determine the allocation policy.

2.12.4.3 L2 cache partitioning way registers (L2PWR*n*)

L2PWR*n*, shown in Figure 2-22, provides controls for partitioning of the L2 cache based on which allocation policy is determined from the L2PIR*n* registers. If the bit associated with an *id* of a transaction sent to the L2 cache is set in one of the L2PIR*n* registers, then that register number (0 - *n*) is used to index among the allocation policies represented by L2PAR*n* and L2PWR*n*.

L2PWR*n* controls which ways are available for a line to allocate into should allocation for the transaction be allowed by L2PAR*n*. The ways are represented as a bit vector where way *x* is represented by bit *x* + 32. Only bits 32:32+(*w*-1) are implemented in each L2PWR*n* register, where *w* represents the number of ways in the L2 cache that are implemented.

The e6500 core implements L2PWR0—L2PWR7 as defined by the architecture and described in *EREF,* except only 16 bits of the WAY field are implemented in each register.

Writing to these registers requires synchronization.

MMR block offset: 0x20C (L2PWR0)
block offset: 0x21C (L2PWR1)
block offset: 0x22C (L2PWR2)
block offset: 0x23C (L2PWR3)
block offset: 0x24C (L2PWR4)
block offset: 0x25C (L2PWR5)
block offset: 0x26C (L2PWR6)
block offset: 0x27C (L2PWR7)

	32 … 47	48 … 63
R W	WAY	—
Reset	All set for L2PWR0, all zeros for other L2PWR*n*	All zeros

Figure 2-22. L2 cache partitioning way registers (L2PWR*n*)

This table describes the L2PWR*n* fields.

Table 2-23. L2PWR*n* field descriptions

Bits	Name	Description
32–47	WAY	Each bit that is set represents a way number into which the transaction can allocate. Multiple bits can be set, representing multiple ways that are available for allocation using this allocation policy. 0 The ways corresponding to bits set to 0 are not available for allocation. 1 The ways corresponding to bits set to 1 are available for allocation.
48–63	—	Reserved, should be cleared.

2.12.5 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 e6500 core implements the L2 error detection registers as they are defined by the architecture and described in *EREF*. Deviations from the architecture are described in this section.

2.12.5.1 L2 Cache Error Disable (L2ERRDIS) register

L2ERRDIS, shown in Figure 2-23, provides general control for disabling error detection in the L2 cache of the processor. The e6500 implements L2ERRDIS as defined by the architecture and described in *EREF,* with the following exceptions:

- It does not implement the TPARDIS and PARDIS fields.
- It implements the implementation-specific field TMHITDIS.

MMR block offset: 0xe44

	32–55	56	57	58	59	60	61	62	63
R W	—	TMHITDIS	TMBECCDIS	TSBECCDIS	—	MBECCDIS	SBECCDIS	—	L2CFGDIS
Reset	All Zeros								

Figure 2-23. L2 Cache Error Disable (L2ERRDIS) register

This table describes the L2ERRDIS field descriptions.

Table 2-24. L2ERRDIS field descriptions

Bits	Name	Description
32–56	—	Reserved
56	TMHITDIS	Tag/status multi-way hit error disable 0 Tag multi-way hit detection is enabled. 1 Tag multi-way hit error detection is disabled. **Note:** This field is not part of *EREF*. **Note:** While error injection is performed, the values 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.

Table 2-24. L2ERRDIS field descriptions (continued)

Bits	Name	Description
57	TMBECCDIS	Tag multiple-bit ECC error disable 0 Tag Multiple-bit ECC error detection is enabled. 1 Tag Multiple-bit ECC error detection is disabled. **Note:** While error injection is performed, TMBECCDIS = 0 and L2CSR0[L2PE] = 1 should always be configured to ensure that errors are always detected. If they are not set when error injection is performed, the result is undefined.
58	TSBECCDIS	Tag ECC error disable 0 Tag Single-bit ECC error detection is enabled. 1 Tag Single-bit ECC error detection is disabled. **Note:** While error injection is performed, TSBECCDIS = 0 and L2CSR0[L2PE] = 1 should always be configured to ensure that errors are always detected. If they are not set when error injection is performed, the result is undefined.
59	(TPARDIS)	Tag parity error disable. This field is not implemented in the e6500 core and always reads as 0.
60	MBECCDIS	Data multiple-bit ECC error disable 0 Data Multiple-bit ECC error detection is enabled. 1 Data Multiple-bit ECC error detection is disabled. **Note:** While error injection is performed, the values 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 single-bit ECC error disable 0 Data Single-bit ECC error detection is enabled. 1 Data Single-bit ECC error detection is disabled. **Note:** While error injection is performed, the values 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.This field is not implemented in the e6500 core and always reads as 0.
63	L2CFGDIS	L2 configuration error disable 0 L2 configuration error detection is enabled. 1 L2 configuration error detection is disabled.

2.12.5.2 L2 Cache Error Detect (L2ERRDET) register

L2ERRDET, shown in Figure 2-24, provides general status and information for errors detected in the L2 cache of the processor. The e6500 core implements L2ERRDET as defined by the architecture and described in *EREF*, with the following exceptions:

- It does not implement the TPARERR and PARERR fields.
- It implements the implementation-specific fields MULL2ERR and TMHITERR.

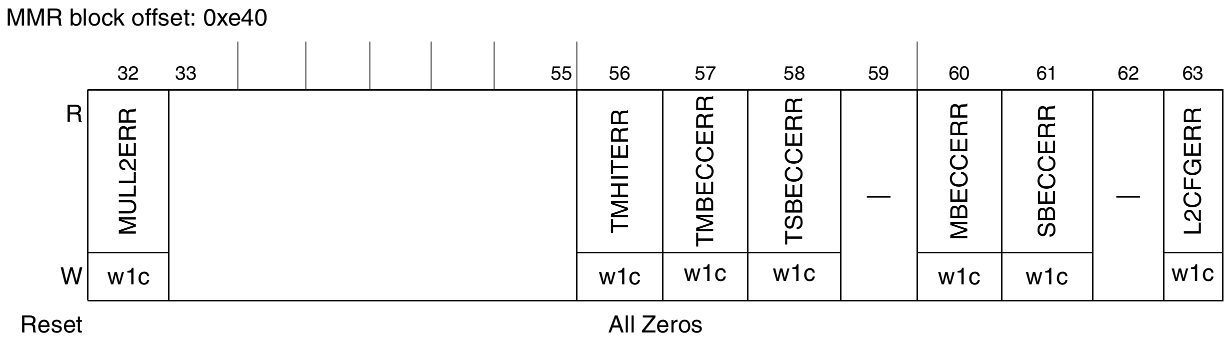


Figure 2-24. L2 Cache Error Detect (L2ERRDET) register

This table describes the L2ERRDET fields.

Table 2-25. 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 are not detected. 1 Multiple L2 errors of the same type are detected. **Note:** This field is not part of *EREF*.
33–55	—	Reserved
56	TMHITERR	Tag multi-way hit error detected. Writing a 1 to this bit location resets the bit. 0 Tag multi-way hit is not detected. 1 Tag multi-way hit is detected. **Note:** This field is not part of *EREF*.
57	TMBECCERR	Tag multiple-bit ECC error detected. Writing a 1 to this bit location resets the bit. 0 Tag Multiple-bit ECC error is not detected. 1 Tag Multiple-bit ECC error is detected.
58	TSBECCERR	Tag single-bit ECC error detected. Writing a 1 to this bit location resets the bit. 0 Tag Single-bit ECC is not detected. 1 Tag Single-bit ECC error is detected.
59	(TPARERR)	Tag parity error detected. This field is not implemented in the e6500 core and always reads as 0.
60	MBECCERR	Data multiple-bit ECC error detected. Writing a 1 to this bit location resets the bit. 0 Tag Multiple-bit ECC error is not detected. 1 Tag Multiple-bit ECC error is detected.
61	SBECCERR	Data single-bit ECC error detected. Writing a 1 to this bit location resets the bit. 0 Tag Single-bit ECC error is not detected. 1 Tag Single-bit ECC error is detected.
62	(PARERR)	Data parity error detected. This field is not implemented in the e6500 core and always reads as 0.
63	L2CFGERR	L2 configuration error detected. Writing a 1 to this bit location resets the bit. 0 L2 configuration error not detected. 1 L2 configuration error detected.

2.12.5.3 L2 Cache Error Interrupt Enable (L2ERRINTEN) register

L2ERRINTEN, shown in Figure 2-25, provides general status and information for errors detected in the L2 cache of the processor. The e6500 core implements L2ERRINTEN as defined by the architecture and described in *EREF*, with the following exceptions:

- It does not implement the TPARINTEN and PARINTEN fields.
- It does implement the implementation-specific field TMHITINTEN.

MMR block offset: 0xe48

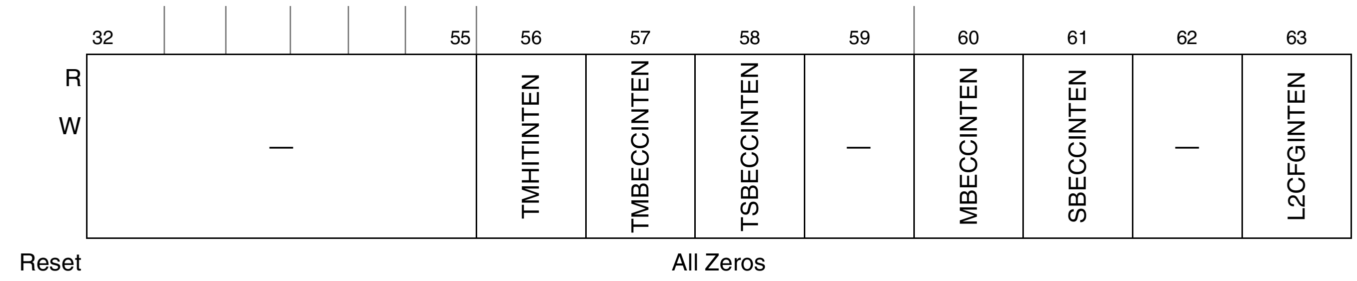


Figure 2-25. L2 Cache Error Interrupt Enable (L2ERRINTEN) register

This table describes the L2ERRINTEN fields.

Table 2-26. 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 is disabled. 1 Tag multi-way hit interrupt reporting is enabled. **Note:** this field is not part of *EREF*.
57	TMBECCINTEN	Tag multiple-bit ECC error interrupt reporting enable 0 Tag multiple-bit ECC error interrupt reporting is disabled. 1 Tag multiple-bit ECC error interrupt reporting is enabled.
58	TSBECCINTEN	Tag ECC interrupt reporting enable 0 Tag single-bit ECC error interrupt reporting is disabled. 1 Tag single-bit ECC error interrupt reporting is enabled.
59	(TPARINTEN)	Tag parity error interrupt reporting enable. This field is not implemented in the e6500 core and always reads as 0.
60	MBECCINTEN	Data multiple-bit ECC error interrupt reporting enable. Valid only if L2CFG0[L2CDEHA] = 0b10. 0 Data Multiple-bit ECC error interrupt reporting is disabled. 1 Data Multiple-bit ECC error interrupt reporting is enabled.
61	SBECCINTEN	Data ECC error interrupt reporting enable. Valid only if L2CFG0[L2CDEHA] = 0b10. 0 Data Single-bit ECC error interrupt reporting is disabled. 1 Data Single-bit ECC error interrupt reporting is enabled.

Table 2-26. L2ERRINTEN field descriptions (continued)

Bits	Name	Description
62	(PARINTEN)	Data parity error interrupt reporting enable. This field is not implemented in the e6500 core and always reads as 0.
63	L2CFGINTEN	L2 configuration error interrupt reporting enable 0 L2 configuration interrupt reporting is disabled. 1 L2 configuration error interrupt reporting is enabled.

2.12.5.4 L2 Cache Error Control (L2ERRCTL) register

L2ERRCTL, shown in Figure 2-26, provides thresholds and counts for errors detected in the L2 cache of the processor. The e6500 core implements L2ERRCTL as defined by the architecture and described in *EREF*.

MMR block offset: 0xe58

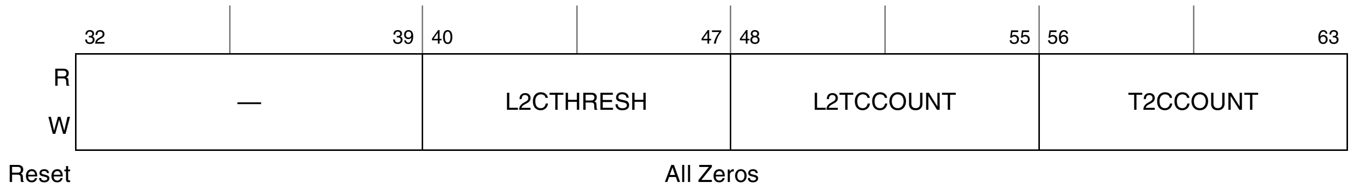


Figure 2-26. L2 Cache Error Control (L2ERRCTL) register

This table describes the L2ERRCTL fields.

Table 2-27. 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 L2TCCOUNT and L2CCOUNT each time a single-bit ECC error is detected. A value of 0 in this field causes the reporting of a single-bit ECC error upon the first occurrence of such an error.
48–55	L2TCCOUNT	L2 tag ECC single-bit error count. L2TCCOUNT counts the number of single-bit errors in the L2 tags which are detected. If L2TCCOUNT equals the ECC single-bit error trigger threshold (L2CTHRESH), an error is reported if single-bit error reporting for tags is enabled. Software should clear this value when such an error is reported to reset the count. e6500 always increments this count when a single-bit ECC error is detected in the tags, regardless of whether single-bit error reporting for tags is enabled.
56–63	L2CCOUNT	L2 data ECC single-bit error count. L2CCOUNT counts the number of single-bit errors in the L2 data which are 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. e6500 always increments this count when a single-bit ECC error is detected in the data, regardless of whether single-bit error reporting for data is enabled.

2.12.5.5 L2 cache error capture address registers (L2ERRADDR and L2ERREADDR)

L2ERRADDR and L2ERREADDR provide the real address of a captured error detected in the L2 cache of the processor. The e6500 core implements these registers as defined by the architecture and described in *EREF*. The real address is 40 bits.

2.12.5.6 L2 cache error capture data registers (L2CAPTDATALO and L2CAPTDATAHI)

L2CAPTDATALO and L2CAPTDATAHI provide 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 e6500 core implements these registers as defined by the architecture and described in *EREF*.

If the captured error is a data ECC error, then these registers contain the data associated with the error. If the captured error is a tag/status ECC error, then L2CAPTDATALO contains the following:

L2CAPTDATALO = low-order 19 bits of the tag || 0b00000 || status[0:7]

L2CAPTDATAHI = 0x000000 || high-order 8 bits of the tag

2.12.5.7 L2 Cache Capture ECC Syndrome (L2CAPTECC) register

L2CAPTECC provides both the calculated and stored ECC syndrome of a captured error detected in the L2 cache of the processor. The e6500 core implements this register as defined by the architecture and described in *EREF*. Tag and status ECC syndromes are left-padded with the appropriate number of zeros.

2.12.5.8 L2 Cache Error Attribute (L2ERRATTR) register

L2ERRATTR, shown in Figure 2-27, provides extended information for errors detected in the L2 cache of the processor. The e6500 implements L2ERRATTR as defined by the architecture and described in *EREF*. It also implements the implementation-specific fields DWNUM, TRANSSRC, TRANSTYPE, and CORE.

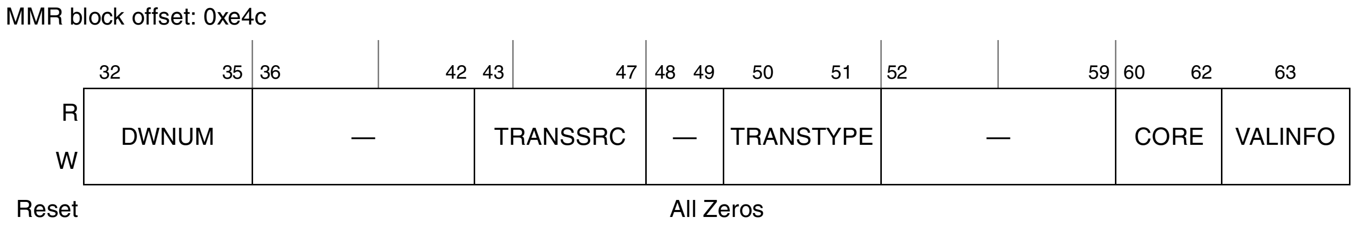


Figure 2-27. L2 Cache Error Attribute (L2ERRATTR) register

This table describes the L2ERRATTR fields.

Table 2-28. L2ERRATTR field descriptions

Bits	Name	Description
32-35	DWNUM	For data ECC errors, contains the doubleword number of the detected error. For tag/status ECC errors, contains which way of the tag/status encountered the error. **Note:** This field is not part of *EREF*.
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*.
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*.
52–59	—	Reserved
60–62	CORE	Core ID that issued TRANSTYPE. If the transaction was from a snoop, this field is undefined. **Note:** This field is not part of *EREF*.
63	VALINFO	L2 capture registers valid 0 L2 capture registers contain no valid information or no enabled errors are detected. 1 L2 capture registers contain information of the first detected error that 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.

2.12.5.9 L2 Cache Error Injection Control (L2ERRINJCTL) register

L2ERRINJCTL, shown in Figure 2-28, 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 *EREF* are implementation dependent, and all fields of this register are e6500 implementation specific.

NOTE

While error injection is performed, the values 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.

MMR block offset: 0xe08

	32 — 46	47	48 — 54	55	56 — 63
R W	—	TERRIEN	—	DERRIEN	ECCERRIM
Reset	All Zeros				

Figure 2-28. L2 Cache Error Injection Control (L2ERRINJCTL) register

This table describes the L2ERRINJCTL fields.

Table 2-29. L2ERRINJCTL field descriptions

Bits	Name	Description
32–46	—	Reserved, should be 0
47	TERRIEN	L2 tag error injection 0 No tag errors are injected. 1 All subsequent entries written to the L2 tag array have the tag ECC bits inverted as specified in the ECC error injection masks. Tag error injection is determined by L2ERRINJHI[59:63] and L2ERRINJLO[32:63]. **Note:** This field is not part of *EREF*.
48–54	—	Reserved, should be 0
55	DERRIEN	L2 data error injection 0 No data errors are injected. 1 Subsequent entries written to the L2 data array have data or data ECC bits inverted as specified in the data and ECC error injection masks. Data error injection is determined by L2ERRINJHI[32:63] and L2ERRINJLO[32:63]. **Note:** This field is not part of *EREF*.
56-63	ECCERRIM	Error injection mask for the ECC syndrome bits. When DERRIEN = 1, the eight ECCERRIM bits map to the eight data ECC bits for each 64 bits of data. When TERRIEN = 1, the low-order seven ECCERIM bits map to the seven tag ECC bits. **Note:** This field is not part of *EREF*.

2.12.5.10 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 or tag array writes when L2ERRINJCTL[TERRIEN] = 1.

The contents of L2ERRINJLO and L2ERRINJHI as defined by the architecture and described in *EREF* are implementation dependent, and all fields of these registers are e6500 implementation specific.

2.13 MMU registers

This section describes the following MMU registers and their fields:

- Logical Partition ID (LPIDR) register
- Process ID (PID) register
- MMU Control and Status 0 (MMUCSR0) register
- MMU Configuration (MMUCFG) register
- TLB configuration registers (TLB*n*CFG)
- TLB page size registers (TLB*n*PS)
- Embedded Page Table Configuration (EPTCFG) register
- MMU assist registers (MAS0–MAS8)
- LRAT Configuration (LRATCFG0) register
- LRAT Page Size (LRATPS) register
- Logical Page Exception (LPER) register

Note that the e6500 core supports MMU architecture version 2 and some fields within registers are different from MMU architecture version 1 and previous cores.

2.13.1 Logical Partition ID (LPIDR) register

LPIDR is implemented for each thread as described in *EREF*.

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 TLPID field in the TLB entry to determine a matching TLB entry.

Only the low-order 6 bits of LPIDR are implemented on the e6500 core.

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

2.13.2 Process ID (PID) register

PID is implemented for each thread as described in *EREF*.

The architecture specifies that the value of PID 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 e6500 core implements all 14 bits for PID values. Writing to PID requires synchronization, as described in Section 3.3.3, “Synchronization requirements.”

2.13.3 MMU Control and Status 0 (MMUCSR0) register

MMUCSR0, shown in Figure 2-29, is used to control the L2 MMUs. The e6500 core implements the L2TLB0_FI, L2TLB1_FI TLB flash invalidate bits, and TLB_EI as defined in *EREF*.

The e6500 core has one MMUCSR0 shared among both threads.

MMUCSR0 synchronization is described in Section 3.3.3, “Synchronization requirements.”

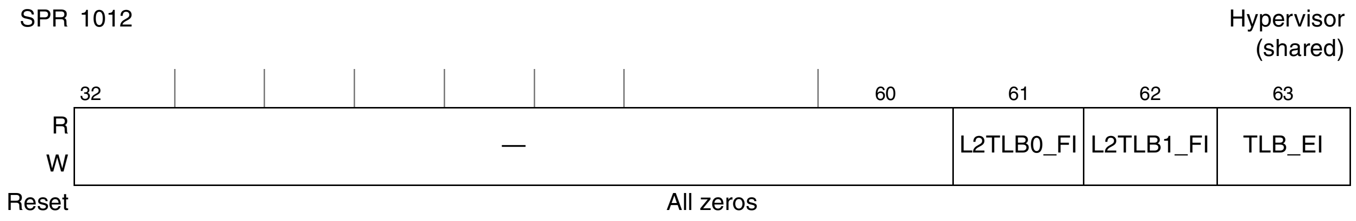


Figure 2-29. MMU Control and Status 0 (MMUCSR0) register

This table describes the MMUCSR0 fields.

Table 2-30. 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 TLB0 invalidation operation. Hardware initiates a TLB0 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 one cycle.
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	TLB_EI	TLB error injection enable. If set, any writes that occur to TLB entries in TLB0 will inject errors. 0 TLB0 error injection is disabled (normal operation) 1 TLB0 error injection is enabled. Any writes to TLB0 have errors injected.

2.13.4 MMU Configuration (MMUCFG) register

MMUCFG, shown in Figure 2-30, provides configuration information about the e6500 MMU and is implemented as defined in *EREF*.

The e6500 core has one MMUCFG shared among both threads.

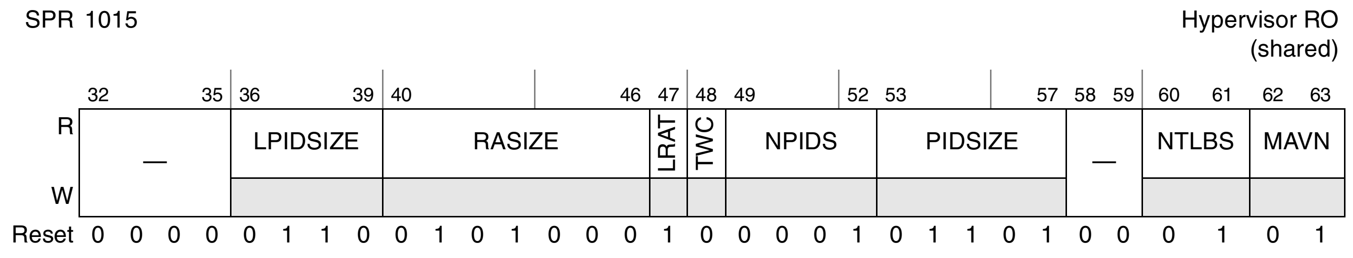


Figure 2-30. MMU Configuration (MMUCFG) register

This table describes MMUCFG fields.

Table 2-31. 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. (0b0101000 indicates 40 physical address bits.)
47	LRAT	LRAT present. (0b1 indicates that LRAT translation is supported for guest supervisor writes to the TLB0 array.)
48	TWC	TLB write conditional. Indicates whether the TLB write conditional and **tlbsrx.** instruction are supported. 0 TLB write conditional and **tlbsrx.** instruction are not supported. 1 TLB write conditional and **tlbsrx.** instruction are supported.
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. (0b01101 indicates PID is 14 bits, PID[50–63].)
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. (0b01 indicates Version 2.0.)

2.13.5 TLB configuration registers (TLB*n*CFG)

TLB*n*CFG, shown in Figure 2-31, are shared by threads and implemented as defined in *EREF*. TLB*n*CFG registers provide configuration information for TLB0 and TLB1 of the L2 MMU.

The e6500 core has one set of TLB*n*CFG registers shared among both threads.

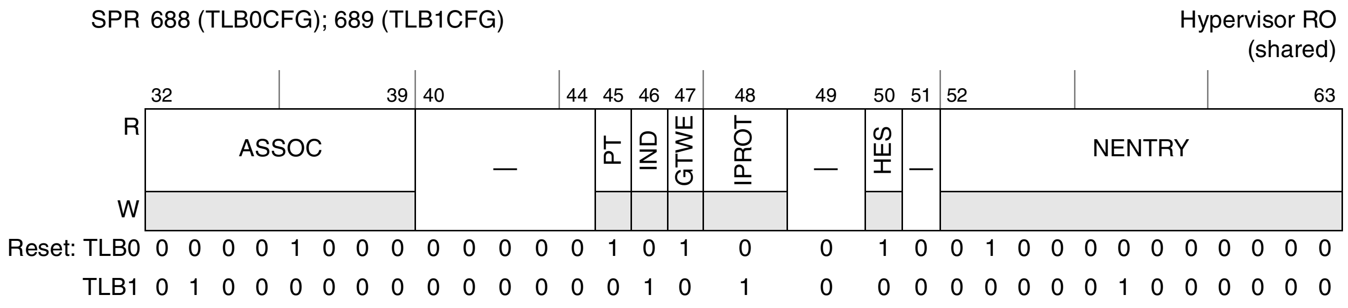

SPR 688 (TLB0CFG); 689 (TLB1CFG)
Hypervisor RO (shared)
32
39
40
44
45
46
47
48
49
50
51
52
63
R
W
ASSOC
—
PT
IND
GTWE
IPROT
—
HES
—
NENTRY
Reset: TLB0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0
TLB1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0

Figure 2-31. TLB configuration registers (TLB0CFG, TLB1CFG)

This table describes the TLB*n*CFG fields and shows the values for the e6500 core.

Table 2-32. TLB*n*CFG field descriptions

Bits	Name	Description
32–39	ASSOC	Associativity of TLB*n* TLB0: 0x08Indicates associativity is 8-way set associative. TLB1: 0x40 Indicates TLB1 is fully associative (because ASSOC = NENTRY).
40–44	—	Reserved
45	PT	Page table. Indicates that the TLB array can be loaded as a result of a hardware tablewalk. 0 The TLB array can not be loaded from a hardware page tablewalk. 1 The TLB array can be loaded from a hardware page tablewalk. TLB0: 1 Indicates TLB0 can be loaded from a hardware page tablewalk. TLB1: 0 Indicates TLB1 can not be loaded from a hardware page tablewalk.
46	IND	Indirect. Indicates that the TLB array can be loaded with an indirect TLB entry and that there is a corresponding EPTCFG register that defines the page size and subpage size. 0 The TLB array can not be loaded with an indirect TLB entry. 1 The TLB array can be loaded with an indirect TLB entry. TLB0: 0 Indicates TLB0 can not be loaded with an indirect TLB entry. TLB1: 1 Indicates TLB1 can be loaded with an indirect TLB entry.
47	GTWE	Guest TLB write entry supported. Indicates that the TLB array can be written (with LRAT translation) by a guest-supervisor **tlbwe** instruction. 0 The TLB array can not be written by a guest-supervisor **tlbwe** instruction. 1 The TLB array can be written by a guest-supervisor **tlbwe** instruction. TLB0: 1 Indicates TLB0 can be written using a guest-supervisor **tlbwe** instruction. TLB1: 0 Indicates TLB1 can not be written using a guest-supervisor **tlbwe** instruction. A **tlbwe** instruction causes a hypervisor privilege exception if it targets an array that does not support GTWE or if EPCR[DGTMI] = 1.
48	IPROT	Invalidate protect capability of TLB*n* 0 The TLB array does not support invalidate protection capability. 1 The TLB array supports invalidate protection capability. TLB0: 0 Indicates that TLB0 does not support invalidate protection capability. TLB1: 1 Indicates that TLB1 supports invalidate protection capability.
49	—	Reserved
50	HES	Hardware entry select. Indicates that the TLB array supports MAS0[HES] where hardware determines which TLB entry is written based on MAS2[EPN]. 0 The TLB array does not support hardware entry select. 1 The TLB array supports hardware entry select. TLB0: 1 Indicates that TLB0 supports hardware entry select. TLB1: 0 Indicates that TLB1 does not supports hardware entry select.
51	—	Reserved
52–63	NENTRY	Number of entries in TLB*n* TLB0: 0x400 TLB0 contains 1024 entries. TLB1: 0x040 TLB1 contains 64 entries.

2.13.6 TLB page size registers (TLB*n*PS)

TLB*n*PS, shown in Figure 2-32, gives configuration information about which page sizes are supported in each TLB array. TLB*n*PS consists of 32 bits, each of which represents whether a page size is supported. If bit *k* is set, then page size 2^{63-k} KB is supported. Page sizes of 4 KB to 1 TB are supported (in power of 2 increments).

This register is hypervisor privileged.

The e6500 core has one set of TLB*n*PS registers shared among both threads.

Note that the method of how page size information is encoded is the same as LRATPS.

SPR 344 (TLB0PS); 345 (TLB1PS)

Hypervisor RO (shared)

	32	33																												61	62	63
	—	TLB*n* page size supported bits																													—	
W																																
TLB0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
TLB1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0

Figure 2-32. TLB page size registers (TLB*n*PS)

This table describes the TLB*n*PS fields.

Table 2-33. TLB*n*PS field descriptions

Bits	Name	Description
32	—	Reserved, should be 0.
33–61	TLB page size supported bits	Page size supported bits for TLB array *n*. When bit *k* of TLB*n*PS is set, page size 2^{63-k} KB is supported by TLB array *n*. For any bit *k* in the register: 0 Page size 2^{63-k} KB is not supported for TLB array *n*. 1 Page size 2^{63-k} KB is supported for TLB array *n*. TLB0: 0x00000004 indicates only 4 KB pages are supported. TLB1: 0x7FFFFFFC indicates 4 KB to 1 TB pages are supported.
62–63	—	Reserved, should be 0.

2.13.7 Embedded Page Table Configuration (EPTCFG) register

EPTCFG, shown in Figure 2-33, gives configuration information about the hardware tablewalk implementation. Each pair of PS*n* and SPS*n* fields describes a page size and sub-page size pair for which the implementation supports. Any SPS*n* field that contains 0 denotes that the pair (and the associated PS*n*) has no information supplied by the pair. A PS*n* value that contains 0 and is paired with a non-zero SPS*n* value denotes that any valid page size supported by the TLB array is allowed for that sub-page size.

PS*n* describes the page size of the indirect TLB entry and the resulting virtual address space that the indirect entry covers. SPS*n* describes the page size of a TLB entry that is written as a result of a successful page table translation. The e6500 core only supports 4 KB sub-page sizes (that is, each PTE doubleword in a page table always describes a 4 KB page.)

This register is hypervisor privileged.

The e6500 core has one EPTCFG shared among both threads.

Note that the notion that a PS*n* can be 0 and be paired with a non-zero SPSn is not part of Power ISA 2.06 and takes advantage of the fact that indirect TLB entries are written to a TLB array that supports variable sizes.

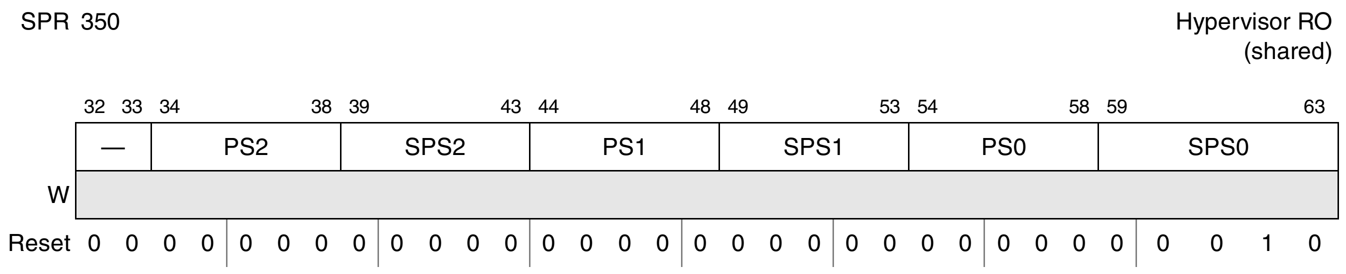


Figure 2-33. Embedded Page Table Configuration (EPTCFG) register

This table describes the EPTCFG fields.

Table 2-34. EPTCFG field descriptions

Bits	Name	Description
32–33	—	Reserved
34–38 44–48 54–58	PS2 PS1 PS0	Page size supported for an indirect TLB entry when paired with the corresponding sub-page size (SPS*n*). If both PS*n* and SPS*n* are zero, the pairing conveys no information. If PS*n* is 0 and SPS*n* is non-zero, then all page sizes supported by the TLB array are supported for indirect TLB entries for that array with the corresponding sub-page size.
39–43 49–53 59–63	SPS2 SPS1 SPS0	Sub-page size supported for an indirect TLB entry when paired with the corresponding page size (PS*n*). If both PS*n* and SPS*n* are zero, the pairing conveys no information. If PS*n* is 0 and SPS*n* is non-zero, then all page sizes supported by the TLB array are supported for indirect TLB entries for that array with the corresponding sub-page size.

2.13.8 Logical to Real Address Translation Configuration (LRATCFG) register

LRATCFG, shown in Figure 2-34, gives configuration information about the implementation’s LRAT and is implemented as defined in *EREF*.

This register is hypervisor privileged and shared by the threads.

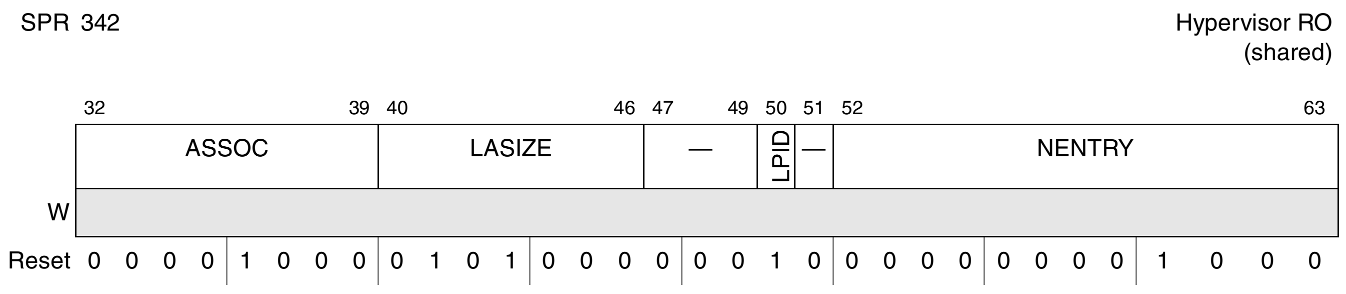


Figure 2-34. Logical to Real Address Translation Configuration (LRATCFG) register

This table describes the LRATCFG fields.

Table 2-35. LRATCFG field descriptions

Bits	Name	Description
32–39	ASSOC	LRAT associativity. Number of ways of associativity of the LRAT array. 0b00001000 indicates fully associative (as it equals NENTRY).
40–46	LASIZE	Logical address size. Number of bits in a logical address supported by the implementation. 0b0101000 indicates 40 bits of logical address.
47–49	—	Reserved, should be 0.
50	LPID	LPID supported. Indicates whether the LPID field in the LRAT is supported. 0b1 indicates that the LPID field in the LRAT is supported.
51	—	Reserved, should be 0.
52–63	NENTRY	Number of entries. Number of entries in the LRAT array. 0b000000001000 indicates 8 entries.

2.13.9 Logical to Real Address Translation Page Size (LRATPS) register

LRATPS, shown in Figure 2-35, gives configuration information about which page sizes an implementation's LRAT supports and is implemented as defined in *EREF*. LRATPS consists of 32 bits, each of which represents whether a page size is supported. If bit *k* is set, then page size 2^{63-k} KB is supported. Page sizes of 4 KB to 1 TB are supported (in power of 2 increments).

This register is hypervisor privileged and is shared by the threads.

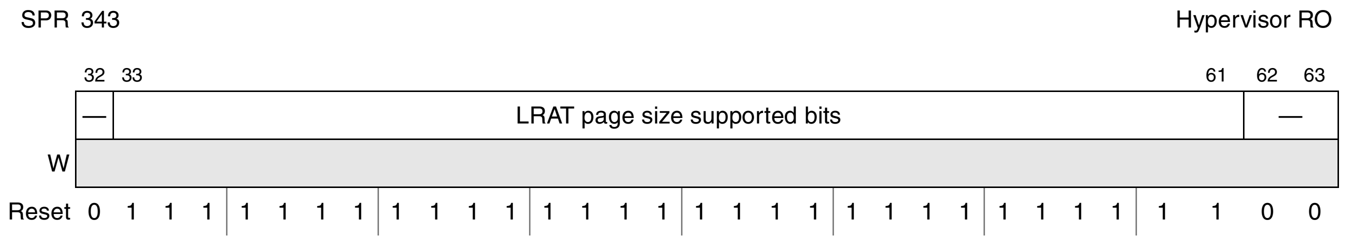

SPR 343
Hypervisor RO
32 33
61 62 63
—
LRAT page size supported bits
—
W
Reset 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0

Figure 2-35. Logical to Real Address Translation Page Size (LRATPS) register

This table describes the LRATPS fields.

Table 2-36. LRATPS field descriptions

Bits	Name	Description
32	—	Reserved, should be 0.
33–61	LRAT page size supported bits	Page size supported bits for the LRAT. For any bit *k* in the register: 0 Page size 2^{63-k} KB is not supported by the LRAT. 1 Page size 2^{63-k} KB is supported by the LRAT. 0x7FFFFFFC indicates 4 KB to 1 TB pages are supported.
62–63	—	Reserved, should be 0.

2.13.10 MMU assist registers (MAS0–MAS8)

MAS*n* registers are used to manage TLBs and the LRAT.

Each thread has a private set of MAS*n* registers.

MAS register contents are written to the TLBs when MAS0[ATSEL] = 0 and a TLB Write Entry (**tlbwe**) instruction executes and are read from the TLBs when MAS0[ATSEL] = 0 and a TLB Read Entry (**tlbre**) instruction or a TLB Search (**tlbsx**) instruction executes. MAS register contents are written to the LRAT when MAS0[ATSEL] = 1 and a TLB Write Entry (**tlbwe**) instruction executes and are read from the LRAT when MAS0[ATSEL] = 1 and a TLB Read Entry (**tlbre**) instruction.

Writing to any MAS register requires synchronization prior to executing a TLB manipulation instruction (**tlbwe**, **tlbre**, **tlbilx**) that 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 the e6500 core, these fields are used as described in the following table:

Table 2-37. TLB selection fields

TLB Write MAS0[ATSEL]	TLB Array MAS0[TLBSEL]	MAS0[ESEL]	MAS2[EPN]	MAS0[NV]
0	0	MAS0[45:47] selects way (low-order 3 bits of ESEL). If MAS0[HES] = 1, these bits are ignored and hardware selects the way.	MAS2[45:51] selects set (low-order 7 bits of EPN).	MAS0[61:63] indicates Next Victim (NV) value for ESEL (low order 3 bits of NV). If MAS0[HES] = 1, these bits are ignored and hardware selects the NV value.
0	1	MAS0[42:47] selects entry (low-order 6 bits of ESEL).	Not used because TLB1 is fully associative	NV field not defined for this TLB array.

TLB read (**tlbre**) and TLB write (**tlbwe**) instructions use MAS0[ESEL] and MAS2[EPN] to select which LRAT entry to read from or write to. On the e6500 core, these fields are used as described in the following table:

Table 2-38. LRAT selection fields

TLB Write MAS0[ATSEL]	TLB Array MAS0[TLBSEL]	MAS0[ESEL]	MAS2[EPN]	MAS0[NV]
1	Not used for LRAT	MAS0[45:47] selects entry (low-order 3 bits of ESEL).	Not used because LRAT is fully associative.	Not used for LRAT.

2.13.10.1 MMU Assist 0 (MAS0) register

MAS0, shown in Figure 2-36, 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 3 bits of NV are implemented. The WQ field is not implemented.

Writing to MAS0 requires synchronization, as described in Section 3.3.3, “Synchronization requirements.”

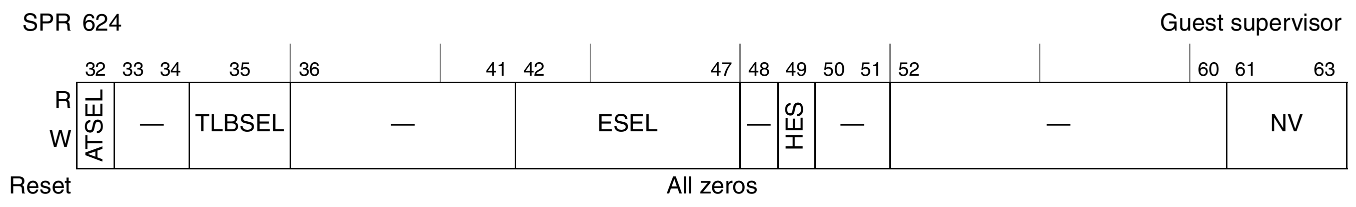


Figure 2-36. MMU Assist 0 (MAS0) register

This table describes the MAS0 fields.

Table 2-39. MAS0 field descriptions — MMU read/write and replacement control

Bit	Name	Description
32	ATSEL	Array type select. Selects LRAT or TLB for access by **tlbwe** or **tlbre** instructions. This field is always treated as 0 in guest state (MSR[GS] = 1). 0 TLB. **tlbwe** and **tlbre** write and read entries out of the TLB arrays. 1 LRAT. **tlbwe** and **tlbre** write and read entries out of the LRAT array.
33–34	—	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. Entry selection selects one of the entries defined by the set selected by TLBSEL and MAS2[EPN]. ESEL is ignored if TLB*n*CFG[HES] = 1 and MAS0[HES] = 1 and entry selection within the set defined by TLBSEL and MAS2[EPN] is performed by hardware. (**Note:** The *n* of TLB*n*CFG is defined by TLBSEL.
48	—	Reserved
49	HES	Hardware entry select. Valid only for **tlbwe** whenTLB*n*CFG[HES] = 1 and ATSEL = 0. **Note:** The *n* of TLB*n*CFG is defined by TLBSEL. Hardware selects which entry in the TLB to write from the set selected by TLBSEL and MAS2[EPN]. 0 Entry selection within the set selected by TLBSEL and MAS2[EPN] is determined by ESEL. 1 Entry selection within the set selected by TLBSEL and MAS2[EPN] is determined by hardware and ESEL is ignored. **Note:** hardware entry select occurs only when: • A TLB write occurs due to a successful hardware page tablewalk, or • *n* = TLBSEL, a **tlbwe** occurs, and TLB*n*CFG[HES] = 1, HES = 1 and ATSEL = 0.
50–60	—	Reserved
61–63	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 e6500 core, 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:** This field is not defined for operations that specify TLB1 (when TLBSEL = 1).

2.13.10.2 MMU Assist 1 (MAS1) register

MAS1, shown in Figure 2-37, is implemented as defined in *EREF*. Only 4 KB through 1 TB page sizes are supported for TSIZE (when using TLB1). Only LRAT 4 KB through 1 TB page sizes are supported for TSIZE.

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

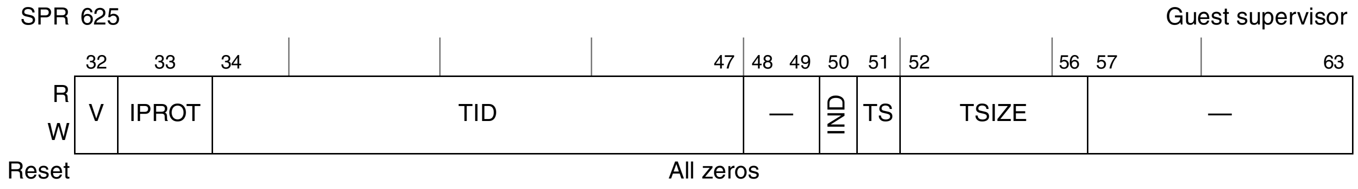


Figure 2-37. MMU Assist 1 (MAS1) register

This table describes the MAS1 fields.

Table 2-40. 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.
34–47	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.
48–49	—	Reserved
50	IND	Indirect. Defines this TLB entry as an indirect entry that is used to find a page table when a hardware page tablewalk is performed. IND is ignored and assumed to be 0 if TLB*n*CFG[IND] = 0. 0 This TLB entry is not an indirect entry and is used for normal translation. 1 This TLB entry is an indirect entry and is used for page table translation. **Note:** The *n* of TLB*n*CFG is defined by TLBSEL.
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.
52–56	TSIZE	Translation size. Defines the page size of the TLB entry and defines the page size of the LRAT entry. For TLB arrays with fixed-size TLB entries, TSIZE is ignored. For variable-size arrays, the page size is 2^{TSIZE} KB. The e6500 core supports TLB page sizes from 4 KB to 1 TB (0b00010 to 0b11110). For LRAT entries, the page size is 2^{TSIZE} KB. The e6500 core supports LRAT page sizes defined by *EREF* from 4 KB to 1 TB (0b00010 to 0b11110).
57–63	—	Reserved

2.13.10.3 MMU Assist 2 (MAS2) register

MAS2, shown in Figure 2-38, is implemented as defined in *EREF*. MAS2 is a 64-bit register. The ACM and VLE fields are not implemented.

Writing to MAS2 requires synchronization, as described in Section 3.3.3, "Synchronization requirements."

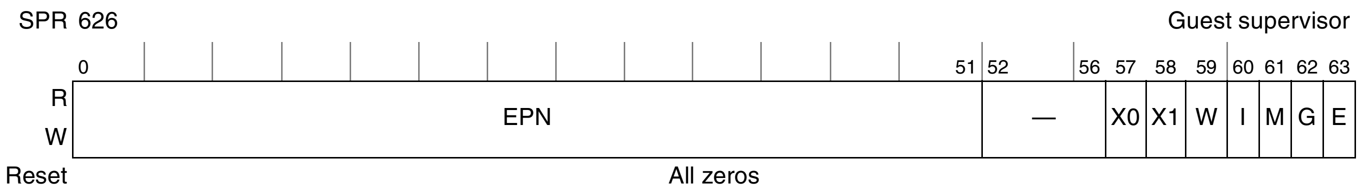


Figure 2-38. MMU Assist 2 (MAS2) register

This table describes the MAS2 fields.

Table 2-41. MAS2 field descriptions — EPN and page attributes

Bits	Name	Description
0–51	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.
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 cache, but the transaction is always performed over CoreNet, ignoring the hit (although the hit may have other unarchitected side effects). **Note:** Cache-inhibited (non-decorated, and non-guarded) loads execute speculatively on the e6500 core.
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 will execute 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 the original PowerPC architecture. 0 The page is accessed in big-endian byte order. 1 The page is accessed in true little-endian byte order.

2.13.10.4 MMU Assist 3 (MAS3) register

MAS3, shown in Figure 2-39, is implemented as defined in *EREF*.

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 or the LRAT will write a logical to real translated 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."

There are two definitions for MAS3 depending on whether an indirect entry is being read or written (MAS1[IND] = 1). If the entry is not an indirect entry (MAS1[IND] = 0), then bits 58–63 are defined as permission bits. Otherwise, bits 58–62 are defined as the SPSIZE field and bit 63 is undefined.

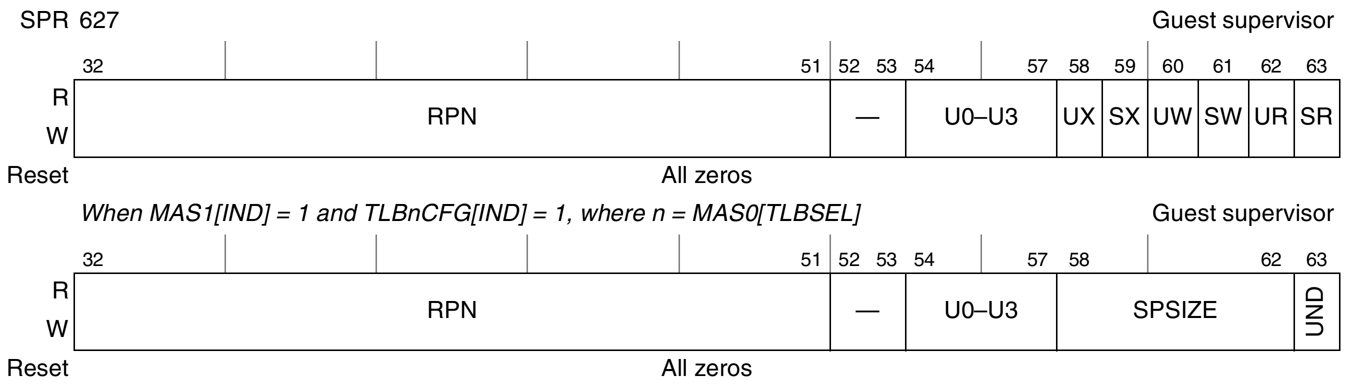


Figure 2-39. MMU Assist 3 (MAS3) register

This table describes the MAS3 fields.

Table 2-42. 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 2.13.10.8, "MAS Register 7 (MAS7)," for more information. For indirect entries, the valid RPN bits are a function of the page size of the indirect entry. The page size as it applies to the RPN in this case is the page size of the indirect entry - 9 (TSIZE - 9).
52–53	—	Reserved

Table 2-42. MAS3 field descriptions — RPN and access control (continued)

Bits	Name	Description
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. User and supervisor read, write, and execute permission bits. See *EREF* for more information on the page permission bits as they are defined by the architecture.
58–62	SPSIZE	Sub-page size. The sub-page size, if the entry is an indirect entry. The entry is only an indirect entry if MAS1[IND] = 1 and TLB*n*CFG[IND] = 1, where *n* = MAS0[TLBSEL]. The e6500 core only supports sub-page sizes of 4 KB. An attempt to write an indirect entry with a value other than 0b00010 for SPSIZE causes the entry to be created with a SPSIZE of 0b00010. A read of an indirect entry always returns 0b00010 in SPSIZE.
63	UND	Contains an undefined value when the entry is an indirect entry.

2.13.10.5 MMU Assist 4 (MAS4) register

MAS4, shown in Figure 2-40, is implemented as defined in *EREF*. Only the low-order bit of TLBSELD is implemented and only 4 KB through 1 TB page sizes 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."

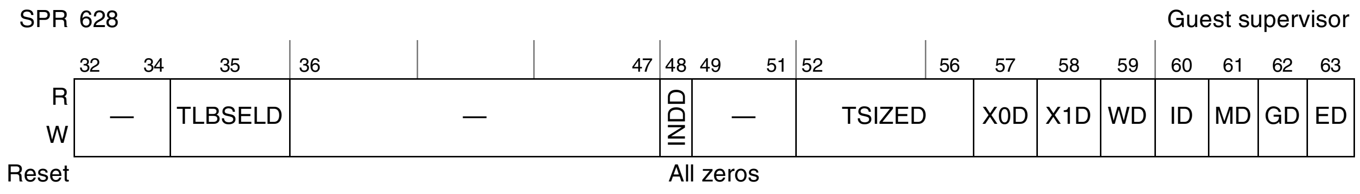


Figure 2-40. MMU Assist 4 (MAS4) register

This table describes the MAS4 fields.

Table 2-43. 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.
36–47	—	Reserved
48	INDD	IND default value. Specifies the default value to be loaded in MAS1[IND] and MAS6[IND] on a TLB miss exception.
49–51	—	Reserved
52–56	TSIZED	Default TSIZE value. Specifies the default value to be loaded into MAS1[TSIZE] on a TLB miss exception.
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.
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.

Table 2-43. MAS4 field descriptions — Hardware replacement assist configuration (continued)

Bits	Name	Description
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.

2.13.10.6 MMU Assist 5 (MAS5) register

MAS5, shown in Figure 2-41, is implemented as defined in *EREF*. 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."

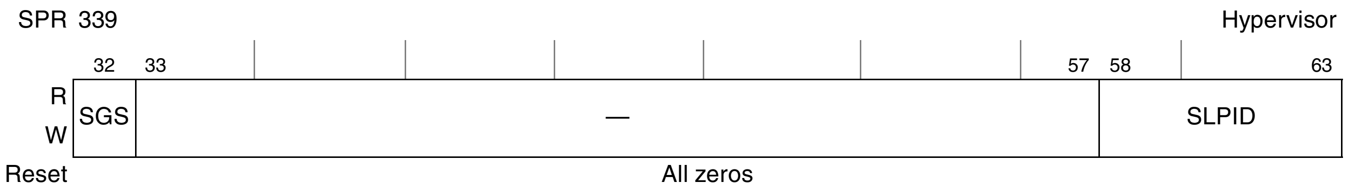


Figure 2-41. MMU Assist 5 (MAS5) register

This table describes the MAS5 fields.

Table 2-44. MAS5 field descriptions

Bits	Name	Description
32	SGS	Search GS. Specifies the GS value used when searching the TLB during execution of **tlbsx**. The SGS field is compared with the TGS field of each TLB entry to find a matching entry.
33–55	—	Reserved
56–63	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.

2.13.10.7 MMU Assist 6 (MAS6) register

MAS6, shown in Figure 2-42, is implemented as defined in *EREF*.

Note that 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."

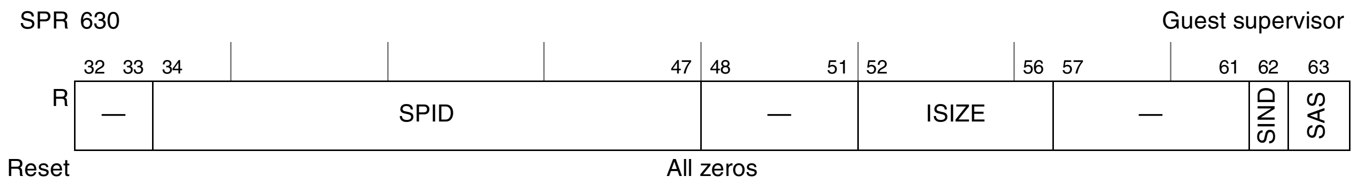


Figure 2-42. MMU Assist 6 (MAS6) register

This table describes the MAS6 fields.

Table 2-45. MAS6 field descriptions

Bits	Name	Description
32–33	—	Reserved
34–47	SPID	Search PID. Specifies the value of PID used when searching the TLB during execution of **tlbsx**.
48–51	—	Reserved
52–56	ISIZE	Invalidation size. Specifies the page size when a **tlbilx** T = 3 or **tlbivax** is executed. The e6500 core does not require ISIZE to be set and ignores it when performing invalidations. Software should set ISIZE to maintain portability with other implementations.
57–61	—	Reserved
62	SIND	Indirect (IND) value for searches and invalidates. Specifies the value of IND used when searching the TLB during execution of **tlbsx** or performing invalidations during execution of **tlbilx** T=3 or **tlbivax**.
63	SAS	Address space (AS) value for searches. Specifies the value of AS used when searching the TLB during execution of **tlbsx**.

2.13.10.8 MAS Register 7 (MAS7)

MAS7, shown in Figure 2-43, is implemented as defined by the *EREF*. MAS7 contains the high-order 32-bits of the real (physical) page number. Since e6500 supports 40 bits of physical address, only the low-order 8 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 or the LRAT will write a logical to real translated 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.”

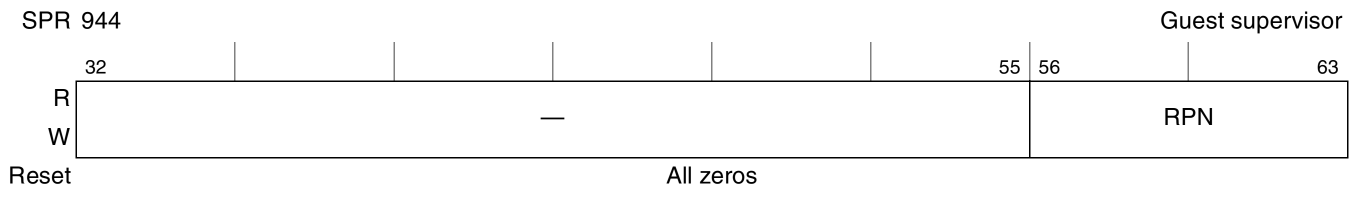


Figure 2-43. MMU Assist 7 (MAS7) register

This table describes the MAS7 fields.

Table 2-46. MAS7 field descriptions — High-order RPN

Bits	Name	Description
32–55	—	Reserved
56–63	RPN	Real page number, 8 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.

2.13.10.9 MMU Assist 8 (MAS8) register

MAS8, shown in Figure 2-44, is implemented as defined in *EREF*. 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."

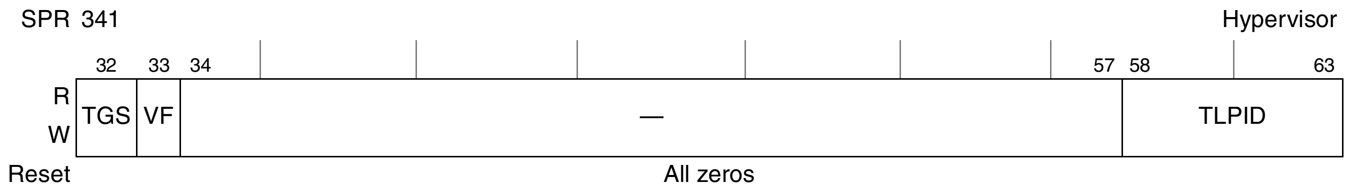


Figure 2-44. MMU Assist 8 (MAS8) register

This table describes the MAS8 fields.

Table 2-47. MAS8 field descriptions

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. If set, data accesses that translate through this TLB entry cause a virtualization fault and subsequent DSI, which is directed to the hypervisor, regardless of the permission bit settings. Instruction accesses that translate through this TLB entry are not affected by this bit. If set in an indirect TLB entry that performs a page table translation, a virtualization fault occurs if the translation is for a data access, and an instruction virtualization fault occurs if the translation is for an instruction fetch access. 0 Accesses translated by this TLB entry occur normally. 1 Accesses translated by this TLB entry always cause a virtualization fault or an instruction virtualization fault (for indirect TLB entry only) and subsequent data or instruction storage interrupt.
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.

2.13.10.10 64-bit access to MAS register pairs

Certain MAS registers can be accessed in pairs with single **mfspr** or **mtspr** instruction. The register pairs are listed in Table 2-48. Software should take special consideration when using MAS register pairs because the programming model is only available on 64-bit implementations. For **mtspr**, all 64 bits are written from the source GPR to the MAS pair. For **mfspr**, all 64 bits are read from the MAS pair and are written to the GPR, regardless of computation mode. If compatibility with 32-bit implementations is desired, MAS register pairs should not be used, and the MAS registers should be addressed individually.

Table 2-48. MAS register pairs

Name	SPR Number	Privilege	Bits 0–31	Bits 32–63
MAS0_MAS1	373	Guest supervisor	MAS0	MAS1
MAS5_MAS6	348	Hypervisor	MAS5	MAS6
MAS7_MAS3	372	Guest supervisor	MAS7	MAS3
MAS8_MAS1	349	Hypervisor	MAS8	MAS1

2.13.11 External PID registers

Each e6500 thread implements private, external PID load and store context registers (EPLC and EPSC) as defined in *EREF*.

2.13.11.1 External PID Load Context (EPLC) register

EPLC, shown in Figure 2-45, contains fields to provide the context for external PID load instructions. Only the low-order 6 bits of the ELPID field are implemented.

Writing to EPLC requires synchronization, as described in Section 3.3.3, “Synchronization requirements.”

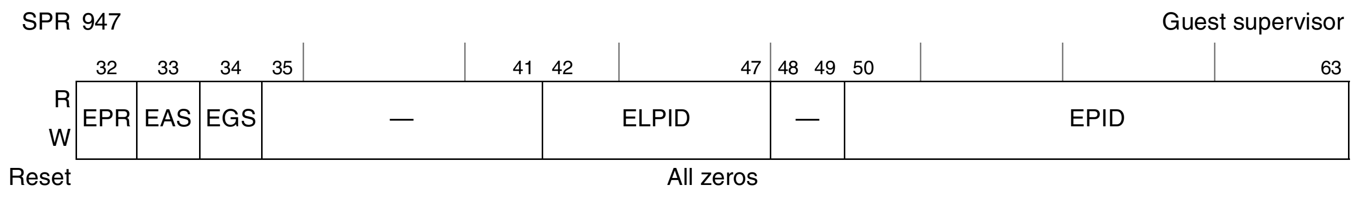


Figure 2-45. External PID Load Context (EPLC) register

This table describes the EPLC fields.

Table 2-49. EPLC field descriptions — External PID load context

Bits	Name	Descriptions
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 the 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 the hypervisor state (MSR[PR] = 0 and MSR[GS] = 0).
48–49	—	Reserved
50–63	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.

2.13.11.2 External PID Store Context (EPSC) register

EPSC, shown in Figure 2-46, 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 are implemented.

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

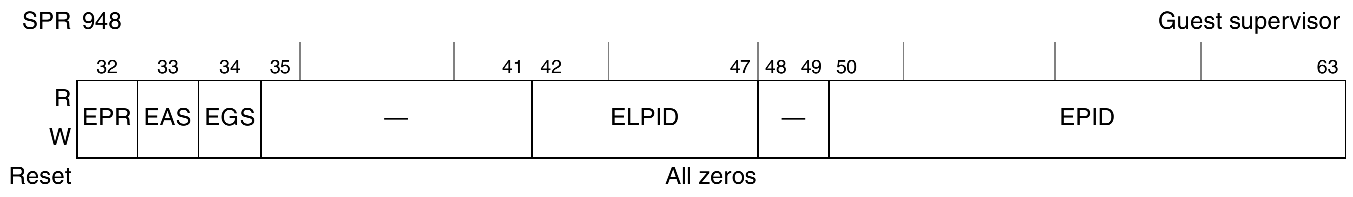


Figure 2-46. External PID Store Context (EPSC) register

This table describes the EPSC fields.

Table 2-50. EPSC field descriptions — External PID store context

Bits	Name	Descriptions
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 load translation when an external PID store instruction is executed. Compared with TLB[TGS] during translation.This field is only writable in the 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 load translation when an external PID store instruction is executed. Compared with TLB[TLPID] during translation. This field is only writable in the hypervisor state (MSR[PR] = 0 and MSR[GS] = 0).
48–49	—	Reserved
50–63	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.

2.14 Internal debug registers

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

The register descriptions listed in this section show the register organization, addressing information, and offer a small amount of detail for the bits implemented. For a more comprehensive description of the debug facilities, including how these registers interact with other debug registers and components, see Chapter 9, “Debug and Performance Monitor Facilities.”

Each thread in the e6500 core has a private set of debug registers, including nexus related and instruction and data address compare registers.

The e6500 core implements the category Embedded Enhanced Debug from *EREF*, 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.

2.14.1 Unimplemented internal debug registers

The e6500 core does not implement the following internal debug registers defined in *EREF*:

- DBCR3
- DVC1, DVC2

2.14.2 Debug Resource Request 0 (DBRR0) register

DBRR0, shown in Figure 2-47, allows an internal software debug agent to request debug resources. After writing this register to request debug resources, reading this register back indicates which ones were granted. This register does not affect the actual allocation of debug resources. Allocation is handled by the EDBRAC0 register.

SPR 700 Hypervisor

	32–33	34	35	36	37	38	39	40	41	42	43	44	45–47	48	49	50	51	52–53	54	55	56	57	58	59	60	61	62	63
R[1] W[2]	—	RST	UDE	ICMP	BRT	IRPT	TRAP	IAC1/2	—	IAC3/4	0	DAC1/2	—	RET	IAC5/6	0	IAC7/8	—	TRACE	PM	EVTO0	CIRPT	CRET	DNI	EVTO1	EVTO2	EVTO3	EVTO4
Reset	All zeros																											

Figure 2-47. Debug Resource Request 0 (DBRR0) register

[1] Reads 0 for resources not granted to the software debug agent (either not requested or previously owned and released). Reads 1 for resources that are granted to the software debug agent.

[2] Write 0 for resources not being requested or to release previously owned resources. Write 1 to request a resource be granted to the software debug agent.

This table describes the DBRR0 fields.

Table 2-51. DBRR0 field descriptions

Bits	Name	Description
32–33	—	Reserved
34	RST	Reset Field Control (DBCR0[RST])
35	UDE	Unconditional Debug Event
36	ICMP	Instruction Complete Debug Event (DBCR0[ICMP])
37	BRT	Branch Taken Debug Event (DBCR0[BRT])
38	IRPT	Interrupt Taken Debug Condition Enable (DBCR0[IRPT])
39	TRAP	Trap Debug Event (DBCR0[TRAP])
40	IAC1/2	Instruction Address Compare 1 and 2
41	—	Reserved
42	IAC3/4	Instruction Address Compare 3 and 4
43	—	Reserved
44	DAC1/2	Data Address Compare 1 and 2
45–47	—	Reserved

Table 2-51. DBRR0 field descriptions (continued)

Bits	Name	Description
48	RET	Return Debug Event (DBCR0[RET])
49	IAC5/6	Instruction Address Compare 5 and 6
50	—	Reserved
51	IAC7/8	Instruction Address Compare 7 and 8
52–53	—	Reserved
54	TRACE	e6500 Nexus Trace
55	PM	Performance Monitor
56	EVTO0[1]	Event Output 0
57	CIRPT	Critical Interrupt Taken Debug Event (DBCR0[CIRPT])
58	CRET	Return From Critical Interrupt Debug Event (DBCR0[CRET])
59	DNI	Debug Notify Interrupt (**dni**) instruction
60	EVTO1[2]	Event Output 1
61	EVTO2[2]	Event Output 2
62	EVTO3[2]	Event Output 3
63	EVTO4[2]	Event Output 4

[1] Does not have a corresponding bit in the External Debug Resource Allocation Control (EDBRAC0) register and, thus, is only provided to identify if the resource is currently in use. Allocation of this resource is not possible and, thus, internal and external debuggers should make every effort to not overwrite the configuration once the resource has been granted.

[2] Does not have a corresponding bit in the External Debug Resource Allocation Control (EDBRAC0) register and, thus, is only provided to identify if the resource is currently being used. Allocation of this resource is not possible and, thus, internal and external debuggers should make every effort to not overwrite the configuration once the resource has been granted. The configuration of multiple event output bits is located in one register (DC3). If these events are shared, it is recommended that the external debugger configure these resources only when the core is halted. Otherwise, read-modify-write problems arise when both the internal and external debugger write to this register, and the results are unpredictable.

2.14.3 External Debug Resource Allocation Control 0 (EDBRAC0) register

EDBRAC0, shown in Figure 2-48, allows an external host debugger to allocate debug resources for its usage.

When EDM mode is not enabled (EDBCR0[EDM] = 0), this register is ignored and an internal software debug agent can use any debug resources.

When EDM mode is enabled (EDBCR0[EDM]=1), debug resources that are allocated to the internal software debug agent (that is, the corresponding bit in this register is set) are usable by software only and are not accessible by the external host debugger. Similarly, debug resources that are allocated to the external host debugger (that is, the corresponding bit in this register is clear) are usable by the external host debugger only and are not accessible by an internal software debug agent.

Only the external host debugger can write to this register. However, this register is readable by the software via the SPR interface.

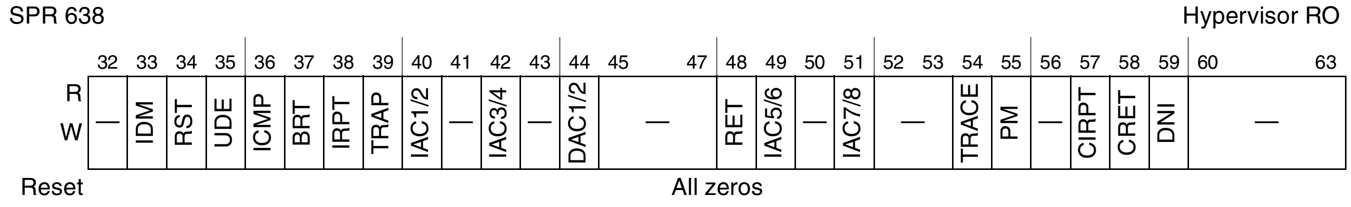


Figure 2-48. External Debug Resource Allocation Control 0 (EDBRAC0) register

This table describes the EDBRAC0 fields.

Table 2-52. EDBRAC0 field descriptions

Bits	Name	Description
32	—	Reserved
33	IDM	Internal Debug Mode Control (DBCR0[IDM]) 0 Internal Debug Mode enable/disable is owned exclusively by an external debug host. 1 Internal Debug Mode enable/disable is owned exclusively by an internal debug agent.
34	RST	Reset Field Control (DBCR0[RST]) 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
35	UDE	Unconditional Debug Event (DBCR0[UDE]) 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
36	ICMP	Instruction Complete Debug Event (DBCR0[ICMP]) 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
37	BRT	Branch Taken Debug Event (DBCR0[BRT]) 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
38	IRPT	Interrupt Taken Debug Condition Enable (DBCR0[IRPT]) 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
39	TRAP	Trap Debug Event (DBCR0[TRAP]) 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
40	IAC1/2	Instruction Address Compare 1 and 2 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
41	—	Reserved
42	IAC3/4	Instruction Address Compare 3 and 4 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
43	—	Reserved
44	DAC1/2	Data Address Compare 1 and 2 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.

Table 2-52. EDBRAC0 field descriptions (continued)

Bits	Name	Description
45–47	—	Reserved
48	RET	Return Debug Event (DBCR0[RET]) 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
49	IAC5/6	Instruction Address Compare 5 and 6 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
50	—	Reserved
51	IAC7/8	Instruction Address Compare 7 and 8 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
52–53	—	Reserved
54	TRACE[1]	e6500 Processor Nexus Trace 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
55	PM	Performance Monitor 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
56	—	Reserved
57	CIRPT	Critical Interrupt Taken Debug Event (DBCR0[CIRPT]) 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
58	CRET	Return From Critical Interrupt Debug Event (DBCR0[CRET]) 0 Debug resource is owned exclusively by an external debug host. 1 Debug resource is allocatable/usable by an internal debug agent.
59	DNI	Debug Notify Interrupt (**dni**) instruction 0 Debug resource is owned exclusively by an external debug host. Execution of the **dni** instruction results in entry into debug halt mode (if EDBSRMSK0[DNIM] = 0). 1 Debug resource is allocatable/usable by an internal debug agent. Execution of the **dni** instruction results in either a debug interrupt (DBCR0[IDM] = 1 and MSR[DE] = 1) or a nop (DBCR0[IDM] = 0 or MSR[DE] = 0).
60–63	—	Reserved

[1] Does not include ownership of the DC2 and DC3 registers (for EVTO0-4 configuration). Those two registers are unprotected.

2.14.4 Debug Control 0 (DBCR0) register

DBCR0, shown in Figure 2-49, is used to enable debug modes and control which debug events are allowed to set DBSR or EDBSR0 flags, reset the thread, and control timer operation during debug events.

All bits in this register (except EDM and FT) are writable only by the owner (as defined in the EDBRAC0 register when EDM is enabled). When EDM is disabled, the internal software debug agent has access to all resources, but should make sure to request the resources it is using via the DBRR0 register prior to configuring them.

EDM is read only in this register and FT is writable via an SPR access only. The external debugger must halt the core and access FT via SPR.

When DBCR0[EDM] or DBCR0[IDM] are enabled, the debug resources in this register that are not enabled (excluding RST and FT) generate a watchpoint when the debug event occurs. Debug resources that are enabled (excluding RST and FT) cause the appropriate status register bit to be set (DBSR for debug events allocated to IDM, and EDBSR0 for debug events allocated to EDM) and will cause exception processing to begin (for debug events allocated to IDM) or the core to halt (for debug events allocated to EDM).

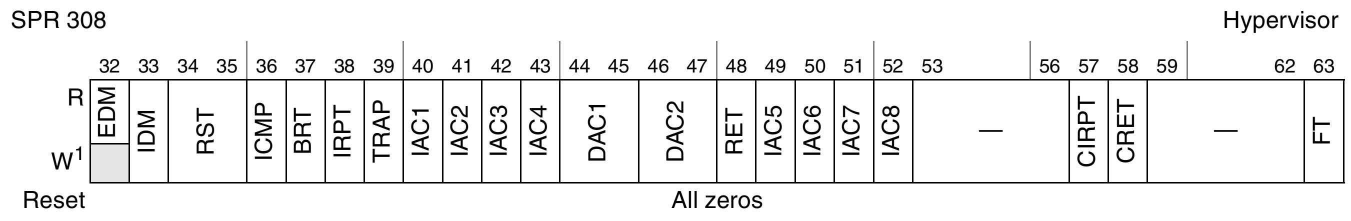


Figure 2-49. Debug Control 0 (DBCR0) register

[1] Individual bits in DBCR0 that are owned by an external debugger (as configured in the EDBRAC0 register) are unaffected by a write via **mtspr**. Similarly, individual bits in DBCR0 that are owned by a software debug agent (as configured in the EDBRAC0 register) are unaffected by a write via the memory-mapped interface.

This table describes the DBCR0 fields.

Table 2-53. 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 External debug mode is disabled. Internal debug events are not mapped into external debug events. 1 External debug mode is enabled. Hardware-owned debug events do not cause the CPU to vector to interrupt code. Software is not permitted to write to debug registers {DBCR0-5, IAC1-8, DAC1-2} unless permitted by settings in DBRR0. Hardware-owned events set status bits in EDBSR0.
33	IDM	Internal Debug Mode 0 Debug exceptions are disabled. Debug events do not affect DBSR. 1 Debug exceptions are enabled. Enabled debug events owned by software update the corresponding bit in the DBSR. If MSR[DE] = 1, the occurrence of a debug event, or the recording of an earlier UDE debug event in the DBSR when MSR[DE] was cleared, causes a debug interrupt.
34–35	RST	Reset. The e6500 core implements these bits as follows: 0*x* Default (no action) 1*x* Core reset. Requests a core hard reset. When owned by an internal software debug agent (EDBRAC0[RST] = 1), a write of DBCR0[RST] = 1x 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.

Table 2-53. DBCR0 field descriptions (continued)

Bits	Name	Description
W 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	IAC3	Instruction Address Compare 3 Debug Condition Enable 0 IAC3 debug conditions are disabled. 1 IAC3 debug conditions are enabled.
43	IAC4	Instruction Address Compare 4 Debug Condition Enable 0 IAC4 debug conditions are disabled. 1 IAC4 debug conditions are enabled.
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.
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	IAC5	Instruction Address Compare 5 Debug Condition Enable 0 IAC5 debug conditions are disabled. 1 IAC5 debug conditions are enabled.
50	IAC6	Instruction Address Compare 6 Debug Condition Enable 0 IAC6 debug conditions are disabled. 1 IAC6 debug conditions are enabled.
51	IAC7	Instruction Address Compare 7 Debug Condition Enable 0 IAC7 debug conditions are disabled. 1 IAC7 debug conditions are enabled.

Table 2-53. DBCR0 field descriptions (continued)

Bits	Name	Description
52	IAC8	Instruction Address Compare 8 Debug Condition Enable 0 IAC8 debug conditions are disabled. 1 IAC8 debug conditions are enabled.
53–56	—	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]). **Note:** The FT bit applies to all timers, including the shared TB and ATB, and each thread’s DEC, FIT, and watchdog timers.

2.14.5 Debug Control 1 (DBCR1) register

DBCR1, shown in Figure 2-50, is implemented as defined by the architecture and described in *EREF*, with the following exceptions:

- IAC1- IAC4 comparisons must be based on effective addresses. Comparisons based on real addresses are not supported
- When IAC12M != 00, IAC2US and IAC2ER settings are ignored and IAC1US and IAC1ER values are used.
- When IAC34M != 00, IAC4US and IAC4ER settings are ignored and IAC3US and IAC3ER values are used.

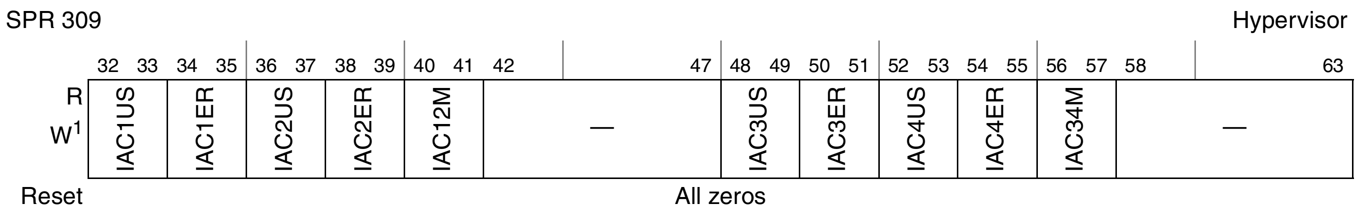


Figure 2-50. Debug Control 1 (DBCR1) register

[1] Software writes are not allowed to EDM-owned resources (as configured in the EDBRAC0 register) and are ignored.

This table describes the DBCR1 fields.

Table 2-54. DBCR1 field descriptions

Bits	Name	Description
32–33	IAC1US	Instruction Address Compare 1 User/Supervisor Mode 00 IAC1 debug conditions are unaffected by MSR[PR], MSR[GS]. 01 Reserved on e6500 10 IAC1 debug conditions can occur only if MSR[PR] = 0 (supervisor mode). 11 IAC1 debug conditions can occur only 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 e6500 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 are unaffected by MSR[PR], MSR[GS]. 01 Reserved on e6500 10 IAC2 debug conditions can occur only if MSR[PR] = 0 (supervisor mode). 11 IAC2 debug conditions can occur only 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 e6500 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[1] 00 Exact address compare. IAC1 debug conditions can occur only if the address of the instruction fetch is equal to the value specified in IAC1. IAC2 debug conditions can occur only 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. IAC1US and IAC1ER are used to define the comparison, and IAC2US and IAC2ER are ignored. 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[2]. IAC2 debug conditions do not occur. The DBCR0[IAC1] setting is used. The value of DBCR0[IAC2] is ignored. IAC1US and IAC1ER are used to define the comparison, and IAC2US and IAC2ER are ignored. 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[3]. IAC2 debug conditions do not occur. The DBCR0[IAC1] setting is used. The value of DBCR0[IAC2] is ignored. IAC1US and IAC1ER are used to define the comparison, and IAC2US and IAC2ER are ignored. The e6500 core sets both DBSR[IAC1] and DBSR[IAC2] if IAC12M is set to anything other than 0b00 and an instruction address compare 1 or 2 event occurs.
42–47	—	Reserved
48–49	IAC3US	Instruction Address Compare 3 User/Supervisor Mode 00 IAC3 debug conditions unaffected by MSR[PR], MSR[GS]. 01 Reserved on e6500 10 IAC3 debug conditions can occur only if MSR[PR] = 0 (supervisor mode). 11 IAC3 debug conditions can occur only if MSR[PR] = 1 (user mode).

Table 2-54. DBCR1 field descriptions (continued)

Bits	Name	Description
50–51	IAC3ER	Instruction Address Compare 3 Effective/Real Mode 00 IAC3 debug conditions are based on effective addresses 01 Reserved on e6500 10 IAC3 debug conditions are based on effective addresses and can occur only if MSR[IS] = 0 11 IAC3 debug conditions are based on effective addresses and can occur only if MSR[IS] = 1
52–53	IAC4US	Instruction Address Compare 4 User/Supervisor Mode 00 IAC4 debug conditions unaffected by MSR[PR], MSR[GS]. 01 Reserved on e6500 10 IAC4 debug conditions can occur only if MSR[PR] = 0 (supervisor mode). 11 IAC4 debug conditions can occur only if MSR[PR] = 1 (user mode).
54–55	IAC4ER	Instruction Address Compare 4 Effective/Real Mode 00 IAC4 debug conditions are based on effective addresses. 01 Reserved on e6500 10 IAC4 debug conditions are based on effective addresses and can occur only if MSR[IS] = 0. 11 IAC4 debug conditions are based on effective addresses and can occur only if MSR[IS] = 1.
56–57	IAC34M	Instruction Address Compare 3/4 Mode[4] 00 Exact address compare. IAC3 debug conditions can occur only if the address of the instruction fetch is equal to the value specified in IAC3. IAC4 debug conditions can occur only if the address of the instruction fetch is equal to the value specified in IAC4. IAC3US, IAC3ER, and DBCR0[IAC3] are used for IAC3 conditions. IAC4US, IAC4ER, and DBCR0[IAC4] are used for IAC4 conditions. 01 Address bit match. IAC3 debug conditions can occur only if the address of the instruction fetch ANDed with the contents of IAC4 is equal to the contents of IAC3, also ANDed with the contents of IAC4. IAC4 debug conditions do not occur. The DBCR0[IAC3] setting is used. The value of DBCR0[IAC4] is ignored. IAC3US and IAC3ER are used to define the comparison, and IAC4US and IAC4ER are ignored. 10 Inclusive address range compare. IAC3 debug conditions can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC3 and less than the value specified in IAC4[5]. IAC4 debug conditions do not occur. The DBCR0[IAC3] setting is used. The value of DBCR0[IAC4] is ignored. IAC3US and IAC3ER are used to define the comparison, and IAC4US and IAC4ER are ignored. 11 Exclusive address range compare. IAC3 debug conditions can occur only if the address of the instruction fetch is less than the value specified in IAC3 or is greater than or equal to the value specified in IAC4[6]. IAC4 debug conditions do not occur. The DBCR0[IAC3] setting is used. The value of DBCR0[IAC4] is ignored. IAC3US and IAC3ER are used to define the comparison, and IAC4US and IAC4ER are ignored. The e6500 core sets both DBSR[IAC3] and DBSR[IAC4] if IAC34M is set to anything other than 0b00 and an instruction address compare 3 or 4 event occurs.
58–63	—	Reserved

[1] When MSR[CM] = 0, IAC*n*[0:31] are treated as zero for the purpose of comparison with the fetch effective address. When MSR[CM] = 1, bits 0-61 of the fetch effective address are compared to IAC*n*[0:61].

[2] If IAC1 > IAC2 or IAC1 = IAC2, a valid condition never occurs.

[3] If IAC1 > IAC2 or IAC1 = IAC2, a valid condition may occur on every instruction fetch.

[4] When MSR[CM] = 0, IAC*n*[0:31] are treated as zero for the purpose of comparison with the fetch effective address. When MSR[CM] = 1, bits 0-61 of the fetch effective address are compared to IAC*n*[0:61].

[5] If IAC3 > IAC4 or IAC3 = IAC4, a valid condition never occurs.

[6] If IAC3 > IAC4 or IAC3 = IAC4, a valid condition may occur on every instruction fetch.

2.14.6 Debug Control 2 (DBCR2) register

DBCR2, shown in Figure 2-51, is implemented as defined by the architecture and described in *EREF*, with the following exceptions:

- DAC comparisons are based on effective addresses only.
- Data Value Compare is not implemented.
- DACLINK1 and DACLINK2 are implemented.

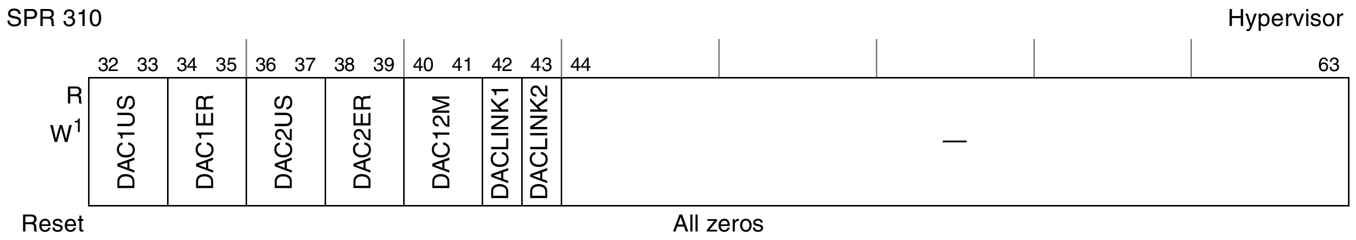


Figure 2-51. Debug Control 2 (DBCR2) register

[1] When EDM is enabled (DBCR0[EDM] = 1), software writes are not allowed to EDM-owned resources (as configured in the EDBRAC0 register) and are ignored.

This table describes the DBCR2 fields.

Table 2-55. DBCR2 field descriptions

Bits	Name	Description
32–33	DAC1US	Data Address Compare 1 User/Supervisor Mode 00 DAC1 debug conditions are unaffected by MSR[PR]. 01 Reserved on e6500 10 DAC1 debug conditions can occur only if MSR[PR] = 0 supervisor mode). 11 DAC1 debug conditions can occur only 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 e6500 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 are unaffected by MSR[PR], MSR[GS]. 01 Reserved on e6500 10 DAC2 debug conditions can occur only if MSR[PR] = 0 (supervisor mode). 11 DAC2 debug conditions can occur only 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 e6500 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 2-55. DBCR2 field descriptions (continued)

Bits	Name	Description
40–41	DAC12M	Data Address Compare 1/2 Mode 00 Exact address compare. DAC1 debug conditions can occur only if the data storage address is equal to the value specified in DAC1. DAC2 debug conditions can occur only 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. The e6500 core sets both DBSR[DAC1] and DBSR[DAC2] if DAC12M is set to anything other than 0b00 and a data address compare 1 or 2 event occurs. DBCR2[DACLINK2] is ignored when DBCR2[DAC12M] is anything other than 0b00.
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 and IAC2 debug events are not generated when DACLINK1 is set irrespective of the DBCR1[IAC12M] setting. 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 3 0 No effect 1 DAC2 debug events are linked to IAC3 debug conditions. IAC3 and IAC4 debug events are not generated when DACLINK2 is set irrespective of the DBCR1[IAC34M] setting. When linked to IAC3, the DAC2 debug event is qualified based on whether the instruction also generated an IAC3 debug condition. DBCR2[DACLINK2] is ignored when DBCR2[DAC12M] is anything other than 0b00.
44–63	—	Reserved

[1] See Section 2.14.7, “Debug Control 4 (DBCR4) register,” 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.

2.14.7 Debug Control 4 (DBCR4) register

DBCR4 is used to extend the data address matching functionality, as described in the following figure.

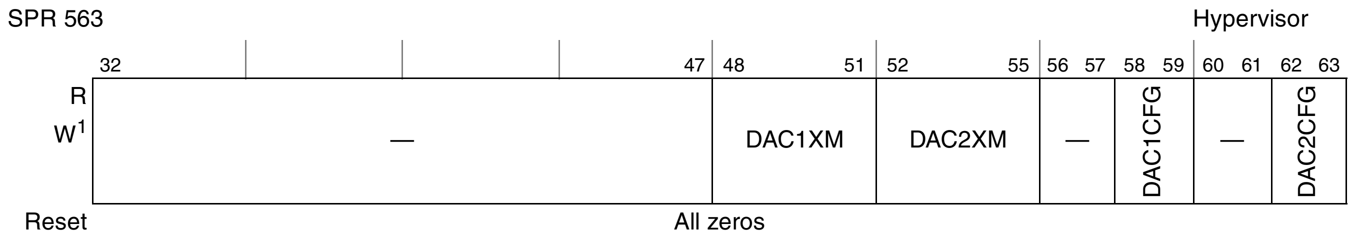


Figure 2-52. Debug Control 4 (DBCR4) register

[1] When EDM is enabled (DBCR0[EDM] = 1), software writes are not allowed to EDM-owned resources (as configured in the EDBRAC0 register) and are ignored.

This table describes the DBCR4 fields.

Table 2-56. 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 e6500 core supports ranges up to 4 KB. 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 e6500 core supports ranges up to 4 KB. 1101–1111 Reserved
56–57	—	Reserved
58–59	DAC1CFG	Data Address Compare 1 Configuration 00 DAC1 debug watchpoints (when DBCR0[DAC1] = 00) are enabled for load-type or store-type storage accesses. 01 DAC1 debug watchpoints (when DBCR0[DAC1RD] = 00) are disabled for load-type storage accesses. 10 DAC1 debug watchpoints (when DBCR0[DAC1WR] = 00) are disabled for store-type storage accesses. 11 DAC1 debug watchpoints (when DBCR0[DAC1] = 00) are disabled.

Bits	Name	Description
60–61	—	Reserved
62–63	DAC2CFG	Data Address Compare 2 Configuration 00 DAC2 debug watchpoints (when DBCR0[DAC2] = 00) are enabled for load-type or store-type storage accesses. 01 DAC2 debug watchpoints (when DBCR0[DAC2RD] = 00) are disabled for load-type storage accesses. 10 DAC2 debug watchpoints (when DBCR0[DAC2WR] = 00) are disabled for store-type storage accesses. 11 DAC2 debug watchpoints (when DBCR0[DAC2] = 00) are disabled.

2.14.8 Debug Control 5 (DBCR5) register

DBCR5, shown in Figure 2-53, is used to configure instruction address compare operation for IAC5-8. This register is implemented as defined by the architecture and described in *EREF*, with the following exceptions:

- IAC5- IAC8 comparisons must be based on effective addresses. Comparisons based on real addresses are not supported.
- When IAC56M != 00, IAC6US and IAC6ER settings are ignored and IAC5US and IAC5ER values are used.
- When IAC78M != 00, IAC8US and IAC8ER settings are ignored and IAC7US and IAC7ER values are used.

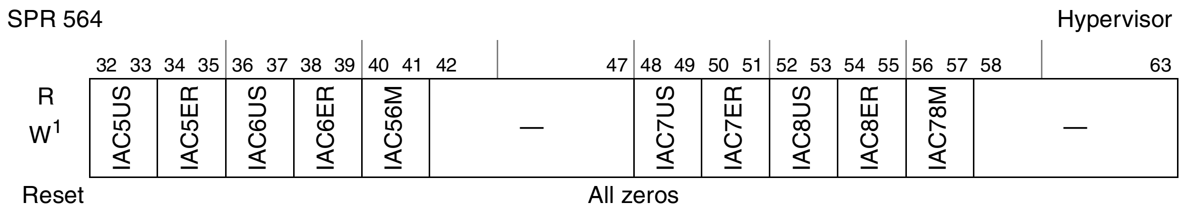


Figure 2-53. Debug Control 5 (DBCR5) register

[1] When EDM is enabled (DBCR0[EDM] = 1), software writes are not allowed to EDM-owned resources (as configured in the EDBRAC0 register) and are ignored.

This table describes the DBCR5 fields.

Table 2-57. DBCR5 field descriptions

Bits	Name	Description
32–33	IAC5US	Instruction Address Compare 5 User/Supervisor Mode 00 IAC5 debug conditions are unaffected by MSR[PR], MSR[GS]. 01 Reserved on e6500 10 IAC5 debug conditions can occur only if MSR[PR] = 0 (supervisor mode). 11 IAC5 debug conditions can occur only if MSR[PR] = 1 (user mode).

Table 2-57. DBCR5 field descriptions (continued)

Bits	Name	Description
34–35	IAC5ER	Instruction Address Compare 5 Effective/Real Mode 00 IAC5 debug conditions are based on effective addresses. 01 Reserved on e6500 10 IAC5 debug conditions are based on effective addresses and can occur only if MSR[IS] = 0. 11 IAC5 debug conditions are based on effective addresses and can occur only if MSR[IS] = 1.
36–37	IAC6US	Instruction Address Compare 6 User/Supervisor Mode 00 IAC6 debug conditions are unaffected by MSR[PR], MSR[GS]. 01 Reserved on e6500 10 IAC6 debug conditions can occur only if MSR[PR]=0 (supervisor mode). 11 IAC6 debug conditions can occur only if MSR[PR]=1 (user mode).
38–39	IAC6ER	Instruction Address Compare 6 Effective/Real Mode 00 IAC6 debug conditions are based on effective addresses. 01 Reserved on e6500 10 IAC6 debug conditions are based on effective addresses and can occur only if MSR[IS] = 0. 11 IAC6 debug conditions are based on effective addresses and can occur only if MSR[IS] = 1.
40–41	IAC56M	Instruction Address Compare 5/6 Mode[1] 00 Exact address compare. IAC5 debug conditions can occur only if the address of the instruction fetch is equal to the value specified in IAC5. IAC6 debug conditions can occur only if the address of the instruction fetch is equal to the value specified in IAC6. IAC5US, IAC5ER, and DBCR0[IAC5] are used for IAC5 conditions. IAC6US, IAC6ER, and DBCR0[IAC6] are used for IAC6 conditions. 01 Address bit match. IAC5 debug conditions can occur only if the address of the instruction fetch ANDed with the contents of IAC6 is equal to the contents of IAC5, also ANDed with the contents of IAC6. IAC6 debug conditions do not occur. The DBCR0[IAC5] setting is used. The value of DBCR0[IAC6] is ignored. IAC6US and IAC6ER settings are ignored, and IAC5US and IAC5ER values are used. 10 Inclusive address range compare. IAC5 debug conditions can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC5 and less than the value specified in IAC6.[2] IAC6 debug conditions do not occur. The DBCR0[IAC5] setting is used. The value of DBCR0[IAC6] is ignored. IAC6US and IAC6ER settings are ignored, and IAC5US and IAC5ER values are used. 11 Exclusive address range compare. IAC5 debug conditions can occur only if the address of the instruction fetch is less than the value specified in IAC5 or is greater than or equal to the value specified in IAC6[3]. IAC6 debug conditions do not occur. The DBCR0[IAC5] setting is used. The value of DBCR0[IAC6] is ignored. IAC6US and IAC6ER settings are ignored, and IAC5US and IAC5ER values are used. The e6500 core sets both DBSR[IAC5] and DBSR[IAC6] bits if IAC56M is set to anything other than 0b00 and an instruction address compare 5 or 6 event occurs.
42–47	—	Reserved
48–49	IAC7US	Instruction Address Compare 7 User/Supervisor Mode 00 IAC7 debug conditions are unaffected by MSR[PR], MSR[GS]. 01 Reserved on e6500 10 IAC7 debug conditions can occur only if MSR[PR] = 0 (supervisor mode). 11 IAC7 debug conditions can occur only if MSR[PR] = 1 (user mode).
50–51	IAC7ER	Instruction Address Compare 7 Effective/Real Mode 00 IAC7 debug conditions are based on effective addresses. 01 Reserved on e6500 10 IAC7 debug conditions are based on effective addresses and can occur only if MSR[IS] = 0. 11 IAC7 debug conditions are based on effective addresses and can occur only if MSR[IS] = 1.

Table 2-57. DBCR5 field descriptions (continued)

Bits	Name	Description
52–53	IAC8US	Instruction Address Compare 8 User/Supervisor Mode 00 IAC8 debug conditions unaffected by MSR[PR], MSR[GS]. 01 Reserved on e6500 10 IAC8 debug conditions can occur only if MSR[PR] = 0 (supervisor mode). 11 IAC8 debug conditions can occur only if MSR[PR] = 1 (user mode.)
54–55	IAC8ER	Instruction Address Compare 8 Effective/Real Mode 00 IAC8 debug conditions are based on effective addresses. 01 Reserved on e6500 10 IAC8 debug conditions are based on effective addresses and can occur only if MSR[IS] = 0. 11 IAC8 debug conditions are based on effective addresses and can occur only if MSR[IS] = 1.
56–57	IAC78M	Instruction Address Compare 7/8 Mode[4] 00 Exact address compare. IAC7 debug conditions can occur only if the address of the instruction fetch is equal to the value specified in IAC7. IAC8 debug conditions can occur only if the address of the instruction fetch is equal to the value specified in IAC8. IAC7US, IAC7ER, and DBCR0[IAC7] are used for IAC7 conditions. IAC8US, IAC8ER, and DBCR0[IAC8] are used for IAC8 conditions. 01 Address bit match. IAC7 debug conditions can occur only if the address of the instruction fetch ANDed with the contents of IAC8 is equal to the contents of IAC7, also ANDed with the contents of IAC8. IAC8 debug conditions do not occur. The DBCR0[IAC7] setting is used. The value of DBCR0[IAC8] is ignored. IAC8US and IAC8ER settings are ignored, and IAC7US and IAC7ER values are used. 10 Inclusive address range compare. IAC7 debug conditions can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC7 and less than the value specified in IAC8.[5] IAC8 debug conditions do not occur. The DBCR0[IAC7] setting is used. The value of DBCR0[IAC8] is ignored. IAC8US and IAC8ER settings are ignored, and IAC7US and IAC7ER values are used 11 Exclusive address range compare. IAC7 debug conditions can occur only if the address of the instruction fetch is less than the value specified in IAC7 or is greater than or equal to the value specified in IAC8.[6] IAC8 debug conditions do not occur. The DBCR0[IAC7] setting is used. The value of DBCR0[IAC8] is ignored. IAC8US and IAC8ER settings are ignored, and IAC7US and IAC7ER values are used. The e6500 core sets both DBSR[IAC7] and DBSR[IAC8] bits if IAC78M is set to anything other than 0b00 and an instruction address compare 7 or 8 event occurs.
58–63	—	Reserved

1 When MSR[CM] = 0, IAC*n*[0:31] are treated as zero for the purpose of comparison with the fetch effective address. When MSR[CM] = 1, bits 0-61 of the fetch effective address are compared to IAC*n*[0:61].

2 If IAC5 > IAC6 or IAC5 = IAC6, a valid condition never occurs.

3 If IAC5 > IAC6 or IAC5 = IAC6, a valid condition may occur on every instruction fetch.

4 When MSR[CM] = 0, IAC*n*[0:31] are treated as zero for the purpose of comparison with the fetch effective address. When MSR[CM] = 1, bits 0-61 of the fetch effective address are compared to IAC*n*[0:61].

5 If IAC7 > IAC8 or IAC7 = IAC8, a valid condition never occurs.

6 If IAC7 > IAC8 or IAC7 = IAC8, a valid condition may occur on every instruction fetch.

2.14.9 Debug Status (DBSR/DBSRWR) register

DBSR provides status information for debug events when DBCR0[IDM] = 1 and the corresponding DBCR0 bit is set, and for the most recent processor reset.

DBSR is implemented as defined by the architecture and described in *EREF*, with the following exception:

- Two additional debug events are possible: CIRPT and CRET

DBSRWR is a write-only register that generally is a write port into DBSR used by the hypervisor to restore delayed debug interrupt state during partition switch. However, the e6500 core does not support delayed debug interrupts, so write capability is not needed. Writes to DBSRWR on the e6500 core are silently dropped and do not affect the value of DBSR.

DBSR is a write-one-to-clear register. Software should normally write DBSR with a mask specifying which bits of DBSR to clear.

This figure shows the Debug Status Register Write (DBSRWR) register.

SPR 306 (DBSRWR) Hypervisor WO

	32–63
R	
W	No effect
Reset	Contents can be read through DBSR only

Figure 2-54. Debug Status Register Write (DBSRWR) register

This figure shows the Debug Status (DBSR) register.

SPR 304 (DBSR) Hypervisor

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53			56	57	58	59	60			63
R/W[1]	—	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1R	DAC1W	DAC2R	DAC2W	RET	IAC5	IAC6	IAC7	IAC8	—				CIRPT	CRET	DNI	—			
Reset	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2-55. Debug Status (DBSR) register

[1] Writing a 1 to DBSR bits that are set clears the bits (write-one-to-clear).

This table describes the DBSR fields.

Table 2-58. DBSR field descriptions

Bits	Name	Description
32	—	Reserved
33	UDE	Unconditional Debug Event Set when an unconditional debug event has occurred while: • DBCR0[IDM] = 1, and • EDBCR0[EDM] = 0 or EDBRAC0[ICMP] = 1 An unconditional debug event can occur when the $\overline{\text{UDE}}$ signal (level sensitive, active low) is asserted to the core. **Note:** Unconditional debug events are not affected by EPCR[DUVD] on the e6500 core.
34–35	MRR	Most Recent Reset. The e6500 implements MRR as follows: 00 No hard reset occurred because this bit was last cleared by software. 01 Reserved 10 The previous reset was a hard reset (default value on power-up). 11 Reserved

Table 2-58. DBSR field descriptions (continued)

Bits	Name	Description
36	ICMP	Instruction Complete Debug Event Set if an instruction complete debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[ICMP] = 1, and DBCR0[ICMP] = 1.
37	BRT	Branch Taken Debug Event Set if a branch taken debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[BRT] = 1, and DBCR0[BRT] = 1.
38	IRPT	Interrupt Taken Debug Event Set if an interrupt taken debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[IRPT] = 1, and DBCR0[IRPT] = 1.
39	TRAP	Trap Instruction Debug Event Set if a trap instruction debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[TRAP] = 1, and DBCR0[TRAP] = 1.
40	IAC1	Instruction Address Compare 1 Debug Event Set if an IAC1 debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[IAC1] = 1, and DBCR0[IAC1] = 1.
41	IAC2	Instruction Address Compare 2 Debug Event Set if an IAC2 debug condition occurs while DBCR0[IDM] = 1, EDBRAC0[IAC2] = 1, and DBCR0[IAC2] = 1.
42	IAC3	Instruction Address Compare 3 Debug Event Set if an IAC3 debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[IAC3] = 1, and DBCR0[IAC3] = 1.
43	IAC4	Instruction Address Compare 4 Debug Event Set if an IAC4 debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[IAC4] = 1, and DBCR0[IAC4] = 1.
44	DAC1R	Data Address Compare 1 Read Debug Event Set if a read-type DAC1 debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[DAC1] = 1, and DBCR0[DAC1] = 0b10 or 0b11.
45	DAC1W	Data Address Compare 1 Write Debug Event Set if a write-type DAC1 debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[DAC1] = 1, and DBCR0[DAC1] = 0b01 or 0b11.
46	DAC2R	Data Address Compare 2 Read Debug Event Set if a read-type DAC2 debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0, or EDBRAC0[DAC2] = 1, and DBCR0[DAC2] = 0b10 or 0b11.
47	DAC2W	Data Address Compare 2 Write Debug Event Set if a write-type DAC2 debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0, or EDBRAC0[DAC2] = 1, and DBCR0[DAC2] = 0b01 or 0b11.
48	RET	Return Debug Event Set if a return debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[RET] = 1, and DBCR0[RET] = 1.
49	IAC5	Instruction Address Compare 5 Debug Event Set if an IAC5 debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[IAC5] = 1, and DBCR0[IAC5] = 1.
50	IAC6	Instruction Address Compare 6 Debug Event Set if an IAC6 debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[IAC6] = 1, and DBCR0[IAC6] = 1.

Table 2-58. DBSR field descriptions (continued)

Bits	Name	Description
51	IAC7	Instruction Address Compare 7 Debug Event Set if an IAC7 debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[IAC7] = 1, and DBCR0[IAC7] = 1.
52	IAC8	Instruction Address Compare 8 Debug Event Set if an IAC8 debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[IAC8] = 1, and DBCR0[IAC8] = 1.
53–56	—	Reserved
57	CIRPT	Critical Interrupt Taken Debug Event Set if a critical interrupt debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[CIRPT] = 1, and DBCR0[CIRPT] = 1.
58	CRET	Critical Return Debug Event Set if a critical return debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[CRET] = 1, and DBCR0[CRET] = 1.
59	DNI	Debug Notify Interrupt (**dni**) instruction Set if a **dni** instruction debug condition occurs while DBCR0[IDM] = 1, EDBCR0[EDM] = 0 or EDBRAC0[DNI] = 1, and MSR[DE] = 1.
60–63	—	Reserved

2.14.10 Instruction address compare registers (IAC1–IAC8)

IAC1–IAC8, shown in Figure 2-56, are implemented as defined by the architecture and described in *EREF*, with one exception: software writes to an IAC*n* register that is owned by an external debugger are ignored while the e6500 core is not halted.

IAC1–IAC8 are 64-bit registers on the e6500 core.

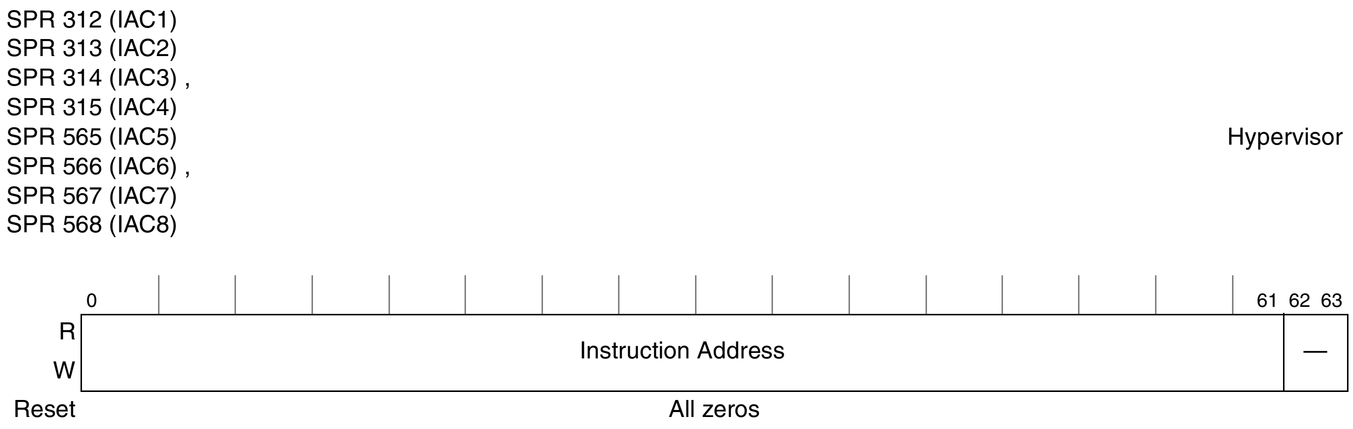


Figure 2-56. Instruction address compare registers (IAC1-IAC8)

2.14.11 Data address compare registers (DAC1–DAC2)

DAC1–DAC2, shown in Figure 2-57, are implemented as defined by the architecture and described in *EREF*, with one exception: software writes to a DAC*n* register that is owned by an external debugger are ignored while the e6500 core is not halted.

DAC1 and DAC2 are 64-bit registers on the e6500 core.

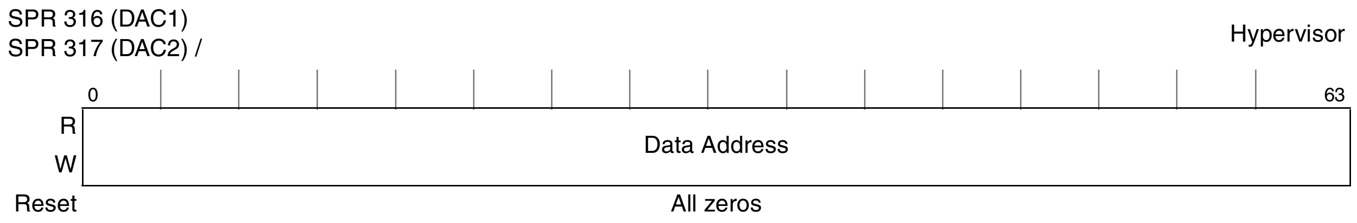


Figure 2-57. Data address compare registers (DAC1–DAC2)

2.14.12 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 9.5, "Nexus registers." The index offset for these registers can be specified in the Nexus SPR Configuration (NSPC) register, after which, access to these registers can be made by using **mtspr** and **mfspr** instructions to read and write the Nexus SPR Data (NSPD) register.

2.14.12.1 Nexus SPR Configuration (NSPC) register

NSPC, shown in Figure 2-58, provides a mechanism for software to access Nexus debug resources through SPR instructions. See Section 9.10.3.2, "Special-purpose register access (Nexus only)," for details on accessing Nexus resources through NSPC.

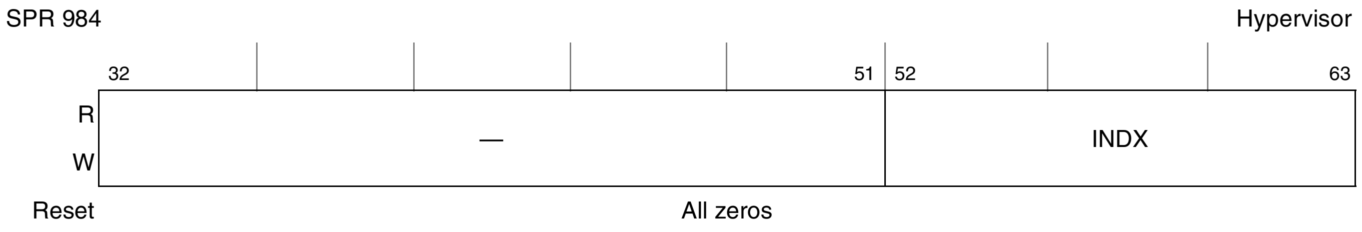


Figure 2-58. Nexus SPR Configuration (NSPC) register

This table describes the NSPC fields. See Table 9-31 for the list of Nexus registers that can be accessed.

Table 2-59. NSPC field descriptions

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

[1] See to Table 9-31 for appropriate index values for accessing Nexus registers.

2.14.12.2 Nexus SPR Data (NSPD) register

NSPD, shown in Figure 2-59, provides a mechanism to transfer data to and from SPR resources. The Nexus resource to be accessed is determined by the programming of 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 NSPD requires an **isync** instruction immediately following the **mtspr** instruction to NSPD to ensure that the write is completed.

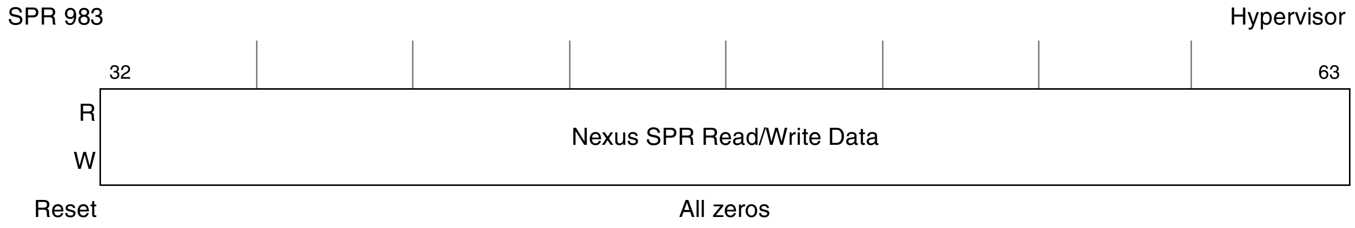


Figure 2-59. Nexus SPR Data (NSPD) register

2.14.13 Debug Event Select (DEVENT) register

DEVENT, shown in Figure 2-60, 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 9.11.17, "Data Acquisition Trace," for more details on the IDTAG.

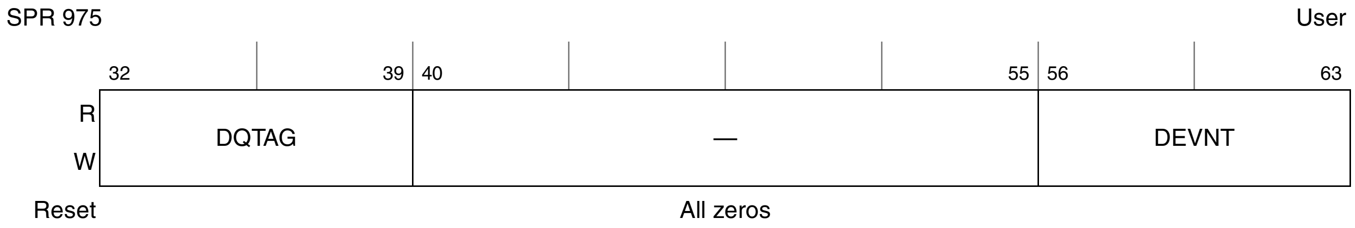


Figure 2-60. Debug Event (DEVENT) register

This table describes the DEVENT fields.

Table 2-60. DEVENT field descriptions

Bits	Name	Description
32–39	DQTAG	IDTAG channel identifier used in Data Acquisition Messages
40–55	—	Reserved

Table 2-60. DEVENT field descriptions (continued)

Bits	Name	Description

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

2.14.14 Debug Data Acquisition Message (DDAM) register

DDAM, shown in Figure 2-61, 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 9.11.17, "Data Acquisition Trace," for details.

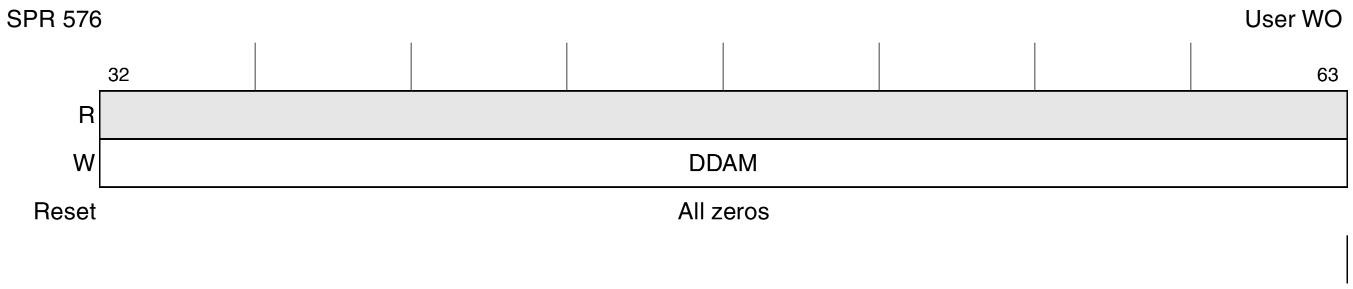


Figure 2-61. Debug Data Acquisition Message (DDAM) register

This table describes the DDAM fields.

Table 2-61. DDAM field descriptions

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

2.14.15 Nexus Process ID (NPIDR) register

NPIDR, shown in Figure 2-62, allows the full process ID utilized by the OS to be transmitted within Nexus Ownership Trace Messages.

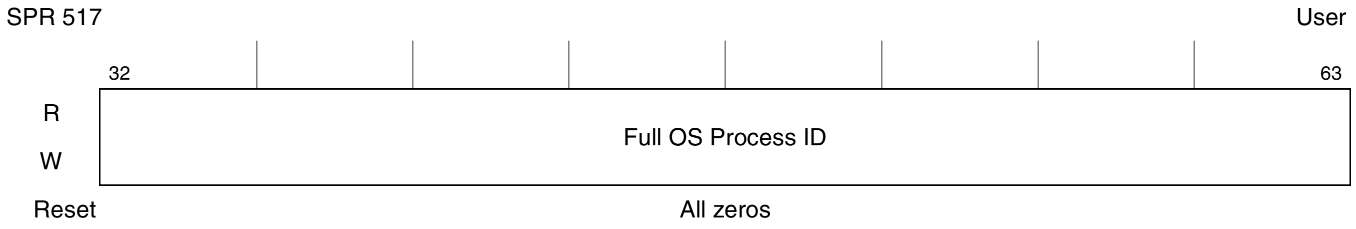


Figure 2-62. Nexus Process ID (NPIDR) register

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.

2.15 Multi-threaded operation management registers

A combination of SPRs and TMRs manages threads and thread execution on the e6500 core.

2.15.1 Thread (processor) management SPRs

The TENSR, TENS, and TENC SPRs provide thread control within the multi-threaded e6500. The TIR SPR register gives a unique thread identifier within a multi-threaded processor. The PPR32 register allows software to assign priorities to the current thread in execution.

2.15.1.1 Thread Identification (TIR) register

TIR, shown in Figure 2-63, contains a read-only, thread-dependent value, with valid values of 0 and 1, that represents a unique thread number within the multi-threaded e6500 core. A thread for which TIR = n is referred to as “thread n.”

Each thread has a private TIR.

Software Note

The value of TIR is the same as the initial value of PIR[THREAD_ID].

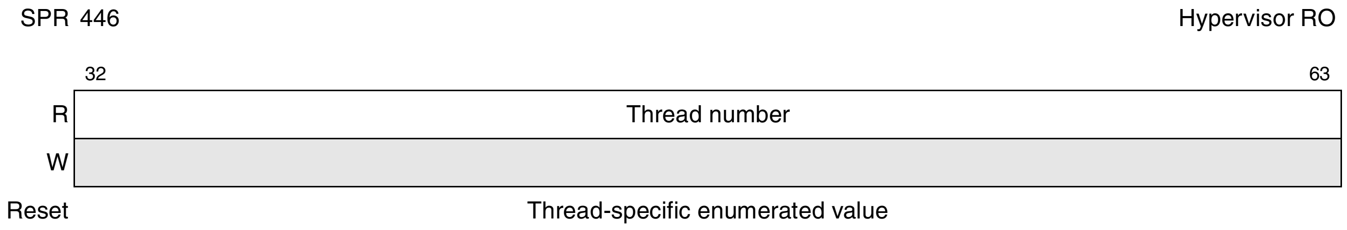


Figure 2-63. Thread Identification (TIR) register

2.15.1.2 Thread Enable (TEN) register

TEN is a 64-bit register, shown in Figure 2-64, that represents which threads are enabled in a multi-threaded processor. For $t < 2$ (e6500 implements two threads), bit 63-t of TEN corresponds to thread t.

The e6500 has a single TEN that is shared by the threads.

TEN is not directly accessible. TEN is written by writing to TENS or the TENC registers, which set or clear bits in TEN, respectively. TEN is read by reading either TENS or TENC. Reading TEN (through TENS or TENC) represents the current set of values from the last writes to TENS and TENC. Reading TEN through the TENSR register represents the current state of the threads. (That is, when a thread is disabled by writing TENC, software can determine when a thread has been disabled by polling TENSR.) Bits representing threads that are not implemented are ignored and always return 0 when read through TENS, TENC, or TENSR.

Software Note

To enable a thread, software sets its associated bit in TENS. More than one thread may be enabled with a single write to TENS if multiple bits are set. To disable a thread, software sets its associated bit in TENC. More than one thread may be disabled with a single write to TENC if multiple bits are set.

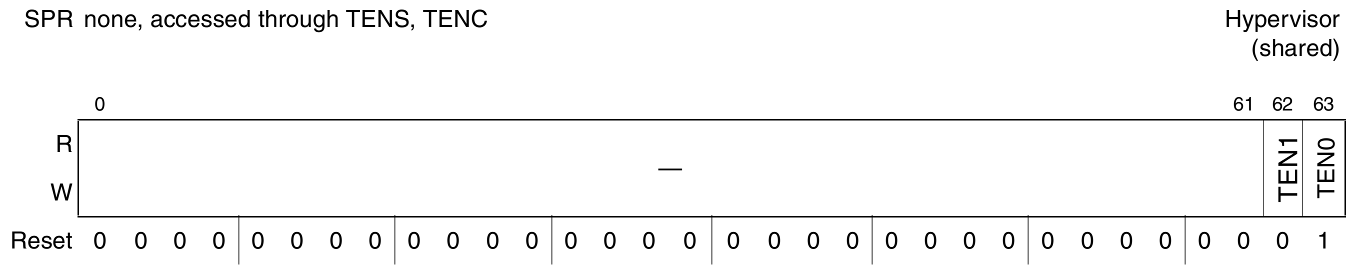

SPR none, accessed through TENS, TENC
Hypervisor (shared)
0
61 62 63
R
W
—
TEN1
TEN0
Reset 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Figure 2-64. Thread Enable (TEN) register

This table describes the TEN fields.

Table 2-62. TEN field descriptions

Bits	Name	Description
0–61	—	Reserved, should be 0.
62	TEN1	Thread 1 is enabled.
63	TEN0	Thread 0 is enabled.

2.15.1.3 Thread Enable Set (TENS) register

TENS is a 64-bit register, shown in Figure 2-65, that allows software to enable threads in the dual-threaded e6500 processor. For $t < 2$ (e6500 implements two threads), bit 63-t corresponds to thread t. When TENS is written, threads for which the corresponding bits in TENS are 1 are enabled; threads for which the corresponding bits in TENS are 0 are unaffected. Reading TEN (through TENS) represents the current set of values from the last writes to TENS and TENC. Bits representing threads that are not implemented are ignored and always return 0 when read through TENS.

The e6500 core has a single TENS that is shared by both threads.

For the e6500 core, software should not attempt to enable a thread that it previously disabled unless the TENSR value of that thread shows that it is completely disabled. A write of 1 to bit *x* of TENS is ignored unless bit *x* of TENSR is 0.

Software Note

To enable a thread, software sets its associated bit in TENS. More than one thread may be enabled with a single write to TENS if multiple bits are set.

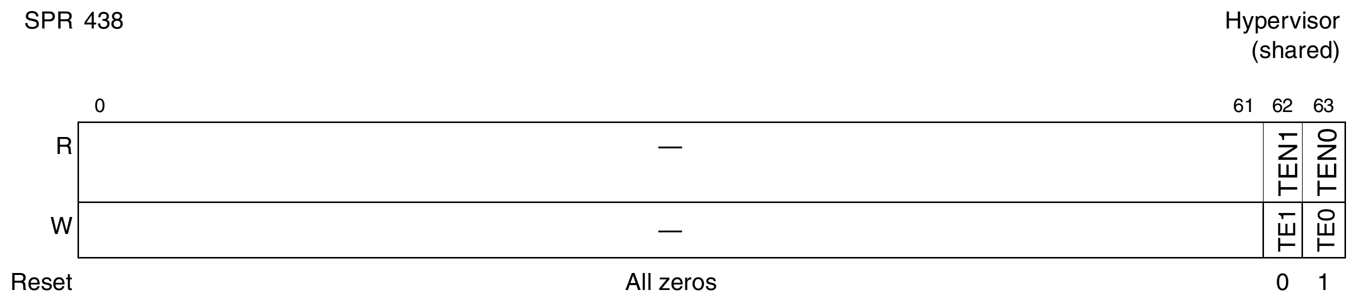


Figure 2-65. Thread Enable Set (TENS) register

This table describes the writable TENS fields. See Table 2-66 for descriptions of the TENS read fields.

Table 2-63. TENS writable field descriptions

Bits	Name	Description
0–61	—	Reserved, should be 0.
62	TE1	Enable (set) thread 1.
63	TE0	Enable (set) thread 0.

2.15.1.4 Thread Enable Clear (TENC) register

TENC is a 64-bit register, shown in Figure 2-66, that allows software to disable threads in the dual-threaded e6500 processor. For $t < 2$ (e6500 implements two threads), bit 63-*t* corresponds to thread *t*. When TENC is written, threads for which the corresponding bits in TENC are 1 are disabled; threads for which the corresponding bits in TENC are 0 are unaffected. Reading TEN (through TENC) represents the current set of values from the last writes to TENS and TENC. Bits representing threads that are not implemented are ignored and always return 0 when read through TENS.

The e6500 core has a single TENC that is shared by both threads.

Software Note

To disable a thread, software sets its associated bit in TENC. More than one thread may be disabled with a single write to TENC if multiple bits are set.

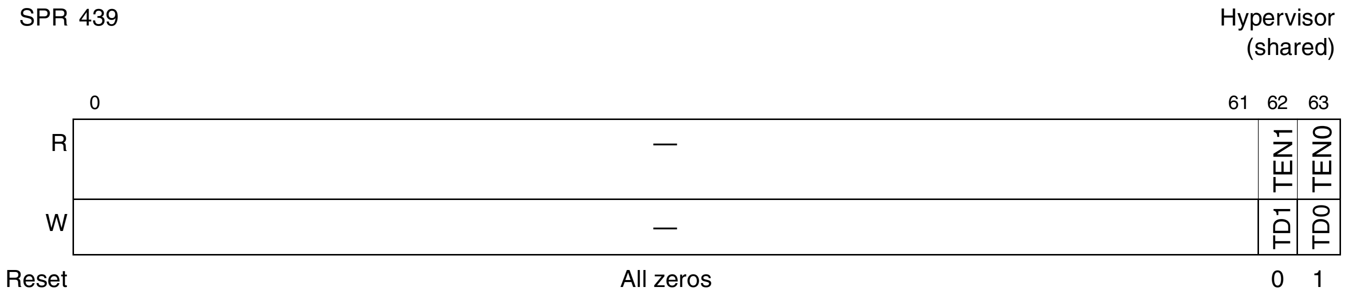


Figure 2-66. Thread Enable Clear (TENC) register

This table describes the writable TENC fields. See Table 2-66 for descriptions of the writable TENC fields.

Table 2-64. TENC writable field descriptions

Bits	Name	Description
0–61	—	Reserved, should be 0.
62	TD1	Disable (clear) thread 1.
63	TD0	Disable (clear) thread 0.

2.15.1.5 Thread Enable Status (TENSR) register

TENSR is a 64-bit register, shown in Figure 2-67, that allows software to determine which threads in a multi-threaded processor are enabled or disabled. For $t < 2$ (e6500 implements 2 threads), bit 63-t corresponds to thread t. When TENSR is read, threads for which the corresponding bits in TENS are 1 are enabled; threads for which the corresponding bits in TENSR are 0 are disabled or unimplemented. When a thread is disabled by writing to TENC, software can determine when the thread actually is disabled by polling the appropriate bit in TENSR.

The e6500 core has a single TENSR that is shared by both threads.

Software Note

When a thread T_x disables other threads, T_z, it writes 1 to the TENC bits corresponding to T_z. In order to ensure that all updates to the shared state among threads in a processor core (SPRs and other state such as caches and TLBs) caused by instructions being performed by threads T_z have been performed with respect to all threads on a processor core, thread T_x reads the TENSR until all the bits corresponding to the disabled threads, T_z are 0s.

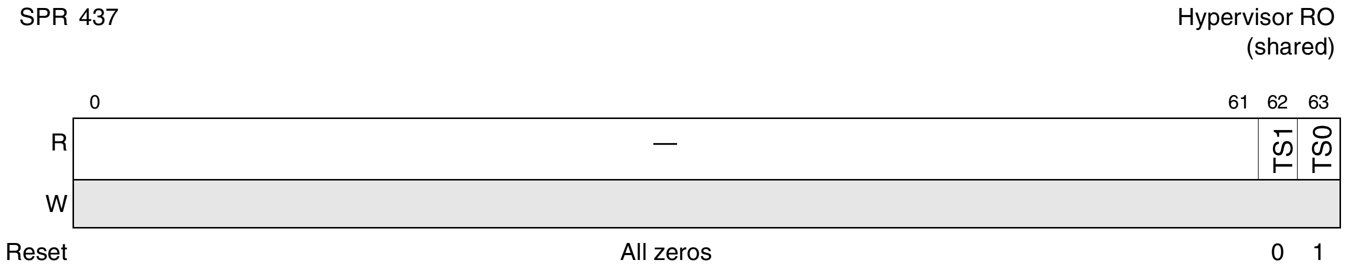


Figure 2-67. Thread Enable Status (TENSR) register

This table describes the TENSR fields.

Table 2-65. TENSR field descriptions

Bits	Name	Description
0–61	—	Reserved, should be 0.
62	TS1	Thread 1 is enabled. A value of 1(0) means that thread 1 is enabled (disabled).
63	TS0	Thread 0 is enabled. A value of 1(0) means that thread 0 is enabled (disabled).

2.15.1.6 Processor Priority (PPR32) register

PPR32, shown in Figure 2-68, specifies what priority a thread has in relation to other threads in a multi-threaded processor. For priority, lower numeric values denote lower priority and higher numeric values denote higher priority. Thread priority may be used to determine which threads have priority when arbitrating for shared resources in a multi-threaded processor. The number of bits implemented in the PRI field is three. The number of bits implemented is defined by TMCFG0[NPRIBITS].

Each of the threads in the e6500 core has a private PPR32.

The PRI field may be set by executing **mtspr** or by executing special forms of the **or** instruction (**or r***x*,**r***x*,**r***x*). In user mode, only values two through four may be set. In the guest-supervisor state, only values one through six can be set. If software attempts to set a value that is not allowed, the PRI field remains unchanged.

The priority of another thread can be changed by writing the TPRI*n* associated with the thread.

The PRI field is an alias for the TMR TPRI*n* register associated with the executing thread.

Note: Thread priorities are not used by the e6500 core multi-threaded processor.

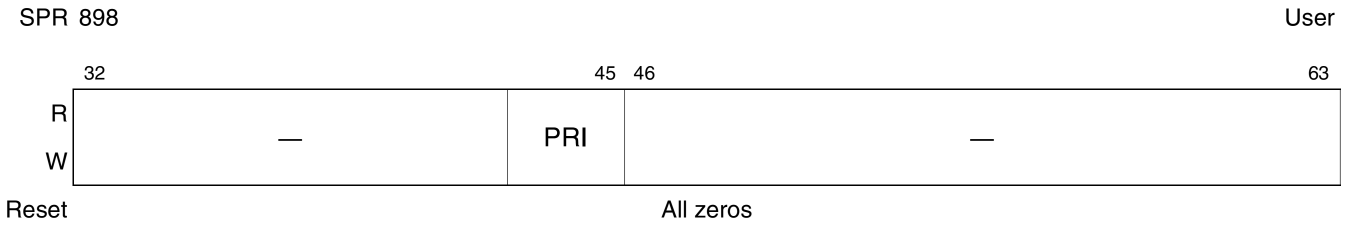


Figure 2-68. Processor Priority (PPR32) register

This table describes the PPR32 fields.

Table 2-66. PPR32 field descriptions

Bits	Name	Description
32–42	—	Reserved, should be 0.
43–45	PRI	Thread priority. The following values are defined: 000 Default (hypervisor only). This value should only be used out of reset. 001 Very low (supervisor or hypervisor only) 010 Low 011 Medium-low 100 Medium 101 Medium-high (supervisor or hypervisor only) 110 High (supervisor or hypervisor only) 111 Very high (hypervisor only)
46–63	—	Reserved, should be 0.

2.15.2 Thread management registers (TMRs)

TMRs are on-chip registers accessed with the **mttmr** and **mftmr** instructions and are used to control the use of threads in the e6500 multi-threaded processor and other architected processor resources related to threads.

The e6500 core has a single set of TMRs, all of which are shared by both threads.

2.15.2.1 Thread Management Configuration 0 (TMCFG0) register

TMCFG0, shown in Figure 2-69, contains read-only configuration information about the multi-threading implementation.

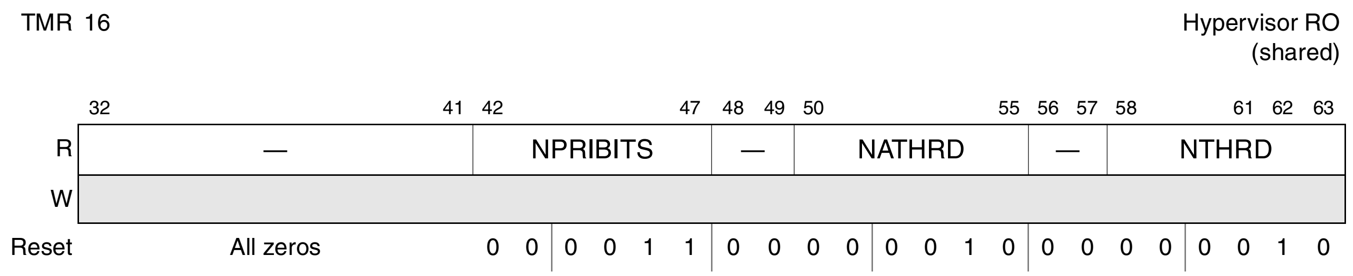


Figure 2-69. Thread Management Configuration 0 (TMCFG0) register

This table describes TMCFG0 fields.

Table 2-67. TMCFG0 field descriptions

Bits	Name	Description
32–41	—	Reserved, should be 0.
42–47	NPRIBITS	Number of bits of thread priority implemented. These are the number of least significant bits of each TPRI*n* register. The e6500 core implements three thread priority bits.
48–49	—	Reserved, should be 0.

Table 2-67. TMCFG0 field descriptions

Bits	Name	Description
50–55	NATHRD	Number of active threads implemented. The e6500 core implements two simultaneously active threads.
56–57	—	Reserved, should be 0.
58–63	NTHRD	Number of threads implemented. The e6500 core implements two threads.

2.15.2.2 Thread Initial Next Instruction Address *n* (INIA*n*) registers

INIA*n* registers, shown in Figure 2-70, are 64-bit registers that contain the current fetch address of a thread, where INIA*n* corresponds to thread *n*. INIA*n* registers specify what address should be used to fetch instructions for each thread. An INIA*n* register may only be written when the associated thread is disabled. Thus, a thread cannot write its own NIA, and writing to the INIA*n* register of any thread *n* that is enabled is ignored on the e6500 core. The facility is expected to be used only in initialization to position threads prior to first execution. Writing to an INIA*n* register in 32-bit mode causes the upper 32 bits in the destination register to be set to 0. Bits 62 and 63 of the INIA*n* registers are not writable and always maintain a value of 0. INIA*n* is an alias for the SPR NIA, which is private to thread *n*.

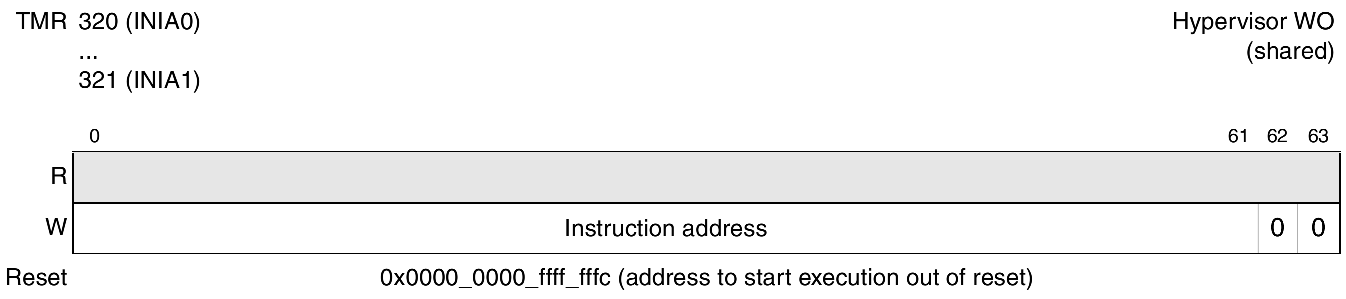


Figure 2-70. Thread Initial Next Instruction Address *n* (INIA*n*) registers

2.15.2.3 Thread Initial Machine State Register *n* (IMSR*n*) registers

IMSR*n* registers, shown in Figure 2-71, contain the current machine state register of a thread, where IMSR*n* corresponds to thread *n*. IMSR*n* registers specify the MSR for each thread. An IMSR*n* register may only be written when the associated thread is disabled. Thus, a thread cannot write its own MSR, and writing to the IMSR*n* register of any thread *n* that is enabled is ignored on the e6500 core. The facility is expected to be used only in initialization to set the initial machine state of threads prior to first execution. IMSR*n* is an alias for the MSR, which is private to thread *n*.

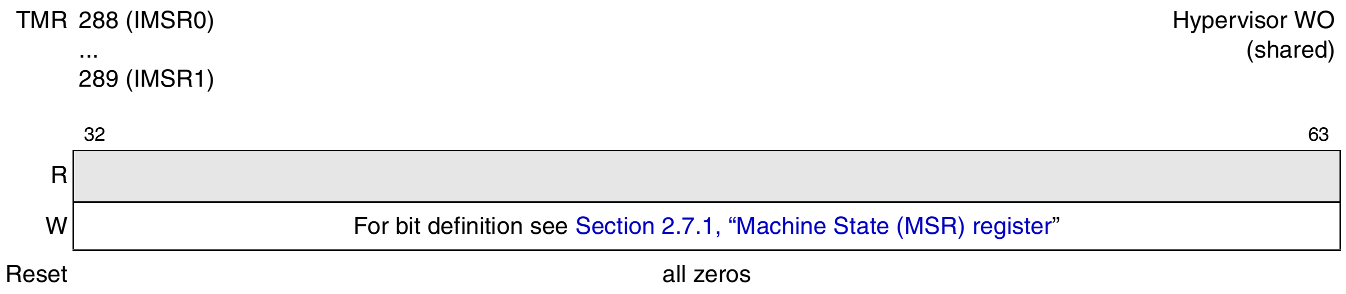


Figure 2-71. Thread Initial Machine State *n* (IMSR*n*) registers

2.15.2.4 Thread Priority *n* (TPRI*n*) registers

TPRI*n* registers, shown in Figure 2-72, allow threads to change the priority of any thread, where TPRI*n* corresponds to thread *n*. TPRI*n* registers specify what priority a thread has in relation to other threads in a multi-threaded processor. For priority, lower numeric values denote lower priority and higher numeric values denote higher priority. Thread priority may be used to determine which threads have priority when arbitrating for shared resources in a multi-threaded processor. Three bits are implemented in the TPRI*n* registers of the e6500 core. The number of bits implemented is defined by TMCFG0[NPRIBITS].

TPRI*n* is an alias for the SPR PPR32[PRI] field, which is private to thread *n*.

Note: Thread priorities are not used by the e6500 multi-threaded processor.

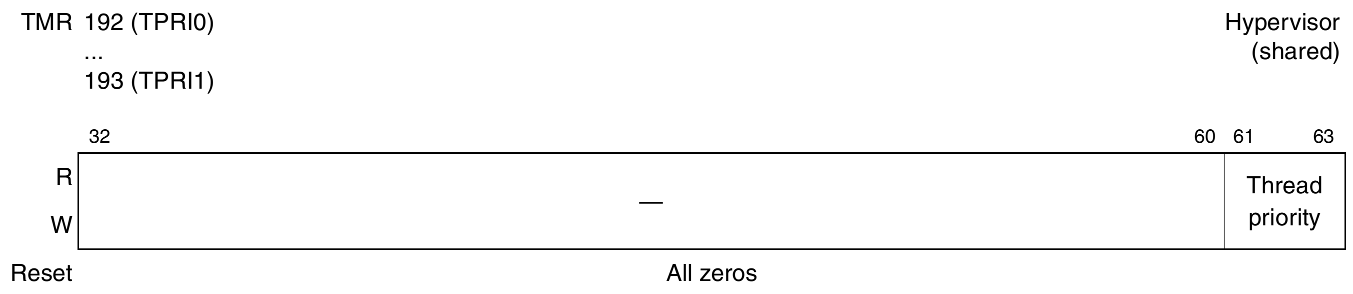


Figure 2-72. Thread Priority *n* (TPRI*n*) registers

2.16 Performance monitor registers (PMRs)

The performance monitor provides a set of performance monitor registers (PMRs) per thread for defining, enabling, and counting conditions that trigger the performance monitor interrupt. PMRs are defined in *EREF*.

The performance monitor also defines (G)IVOR35 (see Section 2.9.5, “(Guest) Interrupt Vector Offset (IVORs/GIVORs) registers”) for providing the address of the performance monitor interrupt vector. (G)IVOR35 is described in the interrupt model chapter of *EREF*. The e6500 core implements a single IVOR35 that is shared by both threads and a private GIVOR35 per thread.

PMRs are similar to SPRs and are accessed using **mtpmr** and **mfpmr** instructions. As shown in the following table, the contents of the PMRs are reflected to a read-only user-level equivalent.

Each thread in the e6500 core has a private set of PMRs.

Table 2-68. Performance monitor registers (PMRs)

Name	Supervisor		User		Section/Page
	Abbreviation	PMR*n*	Abbreviation	PMR*n*	
Performance monitor counter 0	PMC0	16	UPMC0	0	2.16.4/2-118
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 counter 4	PMC4	20	UPMC4	4	
Performance monitor counter 5	PMC5	21	UPMC5	5	
Performance monitor local control a0	PMLCa0	144	UPMLCa0	128	2.16.2/2-112
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 a4	PMLCa4	148	UPMLCa4	132	
Performance monitor local control a5	PMLCa5	149	UPMLCa5	133	
Performance monitor local control b0	PMLCb0	272	UPMLCb0	256	2.16.3/2-114
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 local control b4	PMLCb4	276	UPMLCb4	260	
Performance monitor local control b5	PMLCb5	277	UPMLCb5	261	
Performance monitor global control 0	PMGC0	400	UPMGC0	384	2.16.1/2-112

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. This behavior is described in *EREF*.

2.16.1 Performance Monitor Global Control 0 (PMGC0) and User Performance Monitor Global Control 0 (UPMGC0) registers

PMGC0, shown in Figure 2-73, controls all performance monitor counters. PMGC0 contents are reflected to UPMGC0, which is readable by user-level software. The e6500 core implements these registers as defined in *EREF*, with the exception 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.

PMR PMGC0 (PMR400)UPMGC0 (PMR384) PMGC0: Guest supervisor
UPMGC0: User RO

	32	33	34	35–50	51–52	53–54	55	56–63
R/W	FAC	PMIE	FCECE	—	TBSEL	—	TBEE	—
Reset	All zeros							

Figure 2-73. Performance Monitor Global Control 0 (PMGC0) and User Performance Monitor Global Control 0 (UPMGC0) registers

PMGC0 is cleared by a hard reset. Reading this register does not change its contents. This table describes the e6500-specific PMGC0 and UPMGC0 fields.

Table 2-69. 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[31]) 01 TB[55] (TBL[23]) 10 TB[51] (TBL[19]) 11 TB[47] (TBL[15]) Time base transition events can be used to periodically collect information about processor activity. In multi-processor systems in which TB registers are synchronized among processors, time base transition events can be used to correlate the performance monitor data obtained by 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.

2.16.2 Local control A registers (PMLCa0–PMLCa5/UPMLCa0–UPMLCa5)

PMLCa0–PMLCa5 function as event selectors and give local control for the corresponding performance monitor counters. PMLCa*n* works with the corresponding PMLCb*n*. PMLCa*n* contents are reflected to

UPMLCa*n*. The e6500 core implements these registers as they are defined by the architecture and described in *EREF*, with the following exception:

- The EVENT field only implements the low-order 9 bits of the *EREF*-defined field.

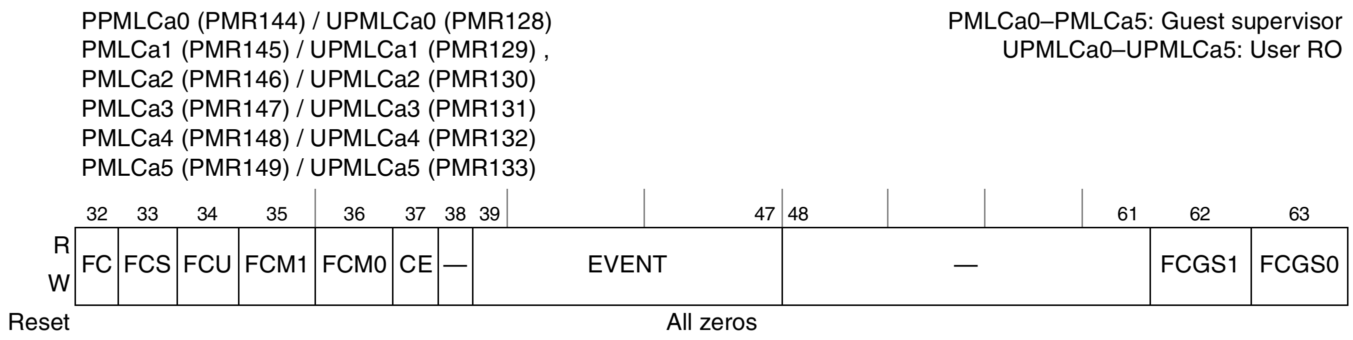

PPMLCa0 (PMR144) / UPMLCa0 (PMR128)
PMLCa1 (PMR145) / UPMLCa1 (PMR129) ,
PMLCa2 (PMR146) / UPMLCa2 (PMR130)
PMLCa3 (PMR147) / UPMLCa3 (PMR131)
PMLCa4 (PMR148) / UPMLCa4 (PMR132)
PMLCa5 (PMR149) / UPMLCa5 (PMR133)
PMLCa0–PMLCa5: Guest supervisor
UPMLCa0–UPMLCa5: User RO
32
33
34
35
36
37
38
39
47
48
61
62
63
R
W
FC
FCS
FCU
FCM1
FCM0
CE
—
EVENT
—
FCGS1
FCGS0
Reset
All zeros

Figure 2-74. Local control A (PMLCa0–PMLCa5/UPMLCa0–UPMLCa5) registers

This table describes the PMLCa fields.

Table 2-70. PMLCa0–PMLCa5/UPMLCa0–UPMLCa5 field descriptions

Bits	Name	Description
32	FC	Freeze counter 0 PMC is incremented (if permitted by other PM control bits). 1 PMC is not incremented.
33	FCS	Freeze counter in supervisor state 0 PMC is incremented (if permitted by other PM control bits). 1 PMC is not incremented if MSR[PR] = 0.
34	FCU	Freeze counter in user state 0 PMC is incremented (if permitted by other PM control bits). 1 PMC is not incremented if MSR[PR] = 1.
35	FCM1	Freeze counter while mark = 1 0 PMC is incremented (if permitted by other PM control bits). 1 PMC is not incremented if MSR[PMM] = 1.
36	FCM0	Freeze counter while mark = 0 0 PMC is incremented (if permitted by other PM control bits). 1 PMC is not incremented if MSR[PMM] = 0.
37	CE	Condition enable 0 PMC*x* overflow conditions cannot occur. (PMC*x* cannot cause interrupts or 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	—	Reserved
39–47	EVENT	Event selector. Up to 511 events are selectable. When this field is 0, the PMC is not incremented.
48–61	—	Reserved

Table 2-70. PMLCa0–PMLCa5/UPMLCa0–UPMLCa5 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.

2.16.3 Local control b registers (PMLCb0–PMLCb5/UPMLCb0–UPMLCb5)

PMLCb0–PMLCb5, shown in Figure 2-75, specify a threshold value and a multiple to apply to a threshold event selected for the corresponding performance monitor counter. For the e6500 core, thresholding is supported only for PMC0 and PMC1. PMLCb*n* works with the corresponding PMLCa*n*. PMLCb*n* contents are reflected to UPMLCa*n*. The e6500 core implements these registers as they are defined in *EREF*, except for the following e6500-specific fields:

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

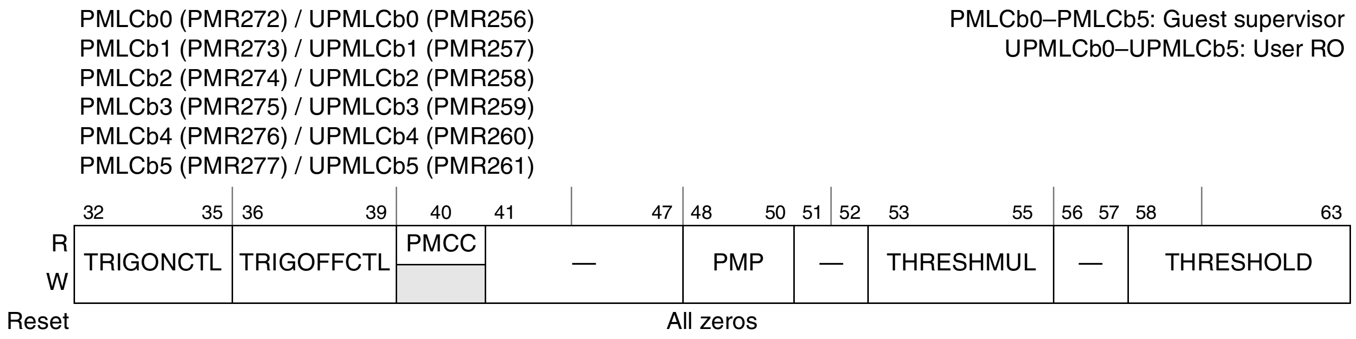

PMLCb0 (PMR272) / UPMLCb0 (PMR256)
PMLCb1 (PMR273) / UPMLCb1 (PMR257)
PMLCb2 (PMR274) / UPMLCb2 (PMR258)
PMLCb3 (PMR275) / UPMLCb3 (PMR259)
PMLCb4 (PMR276) / UPMLCb4 (PMR260)
PMLCb5 (PMR277) / UPMLCb5 (PMR261)
PMLCb0–PMLCb5: Guest supervisor
UPMLCb0–UPMLCb5: User RO
32
35
36
39
40
41
47
48
50
51
52
53
55
56
57
58
63
R
W
TRIGONCTL
TRIGOFFCTL
PMCC
—
PMP
—
THRESHMUL
—
THRESHOLD
Reset
All zeros

Figure 2-75. Local control b registers (PMLCb0–PMLCb5/UPMLCb0–UPMLCb5)

Table 2-71 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.

NOTE

For the purposes of this section, “frozen” means the counter is frozen by means of either PMLCa*x*[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*x* if counting is enabled by all other performance monitor controls. If the trigger state is OFF, counting is disabled for PMC*x*. For both controls, the following applies to how the trigger state is determined:

- When the counter is frozen by means of either PMLCa*x*[FC] = 1 or PMGC0[FAC] = 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 make triggers inactive and all other performance monitor controls determine whether events are counted. Note that if PMLCa*x*[EVENT] = 0, the counter is considered frozen.
- 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 PMC*x* from a counter other than PMLCa*x*[FC] or PMGC0[FAC] have no effect on the trigger state. However, such methods can prevent the counter from counting. That is, the trigger state may be ON, but PMC*x* is not counting events because it is frozen from some other method.

Table 2-71. PMLCb0–PMLCb5/UPMLCb0–UPMLCb5 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 is 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* is asserted. 1111 Reserved **Note:** DVT*n* (DVT0, DVT1, .. DVT7) are asserted by writing to the DEVENT register. See Section 2.14.13, "Debug Event Select (DEVENT) register." The counter trigger ON control uses certain conditions in the processor as signals 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, but still causes counting to begin if the appropriate trigger ON control is set. For a graphic representation of performance monitor counter controls, see Figure 9-34.
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* is asserted. 1111 Reserved **Note:** DVT*n* (DVT0, DVT1, .. DVT7) are asserted by writing to the DEVENT register. See Section 2.14.13, "Debug Event Select (DEVENT) register." The counter trigger OFF control uses certain conditions in the processor as signals 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, but still causes counting to stop if the appropriate trigger OFF control is set. For a graphic representation of performance monitor counter controls, see Figure 9-34.
40	PMCC	PMC*x* trigger state 0 PMC*x* trigger state is OFF. 1 PMC*x* trigger state is ON. **Note:** This is a status bit that shows the trigger state controlled by TRIGONCTL and TRIGOFFCTL. When PMCC = 1, PMC*x* may still not be counting if it is frozen by other means such as PMLCa*x*[FC] or PMGC0[FAC].
41–47	—	Reserved

Table 2-71. PMLCb0–PMLCb5/UPMLCb0–UPMLCb5 field descriptions (continued)

Bits	Name	Description
48–50	PMP	Performance Monitor Overflow Periodicity Select [1] 000 Performance Monitor Watchpoint (PMW*x*) triggers on any change to counter bit 32 (period = 2^{31}). 001 Performance Monitor Watchpoint (PMW*x*) triggers on any change to counter bit 43 (period = 2^{20}). 010 Performance Monitor Watchpoint (PMW*x*) triggers on any change to counter bit 49 (period = 2^{14}). 011 Performance Monitor Watchpoint (PMW*x*) triggers on any change to counter bit 55 (period = 2^{8}). 100 Performance Monitor Watchpoint (PMW*x*) triggers on any change to counter bit 59 (period = 2^{4}). 101 Performance Monitor Watchpoint (PMW*x*) triggers on any change to counter bit 61 (period = 2^{2}). 110 Performance Monitor Watchpoint (PMW*x*) triggers on any change to counter bit 62 (period = 2^{1}). 111 Performance Monitor Watchpoint (PMW*x*) 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*x*[THRESHOLD] × 1). 001 Threshold field is multiplied by 2 (PMLCb*x*[THRESHOLD] × 2). 010 Threshold field is multiplied by 4 (PMLCb*x*[THRESHOLD] × 4). 011 Threshold field is multiplied by 8 (PMLCb*x*[THRESHOLD] × 8). 100 Threshold field is multiplied by 16 (PMLCb*x*[THRESHOLD] × 16). 101 Threshold field is multiplied by 32 (PMLCb*x*[THRESHOLD] × 32). 110 Threshold field is multiplied by 64 (PMLCb*x*[THRESHOLD] × 64). 111 Threshold field is multiplied by 128 (PMLCb*x*[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 (PMWx) that can be used for triggering or to generate Watchpoint Messages (if enabled).

2.16.4 Performance monitor counter registers (PMC0–PMC5/UPMC0–UPMC5)

PMCs and UPMCs, shown in Figure 2-76, are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count 255 events. The e6500 core implements these registers as defined in *EREF*.

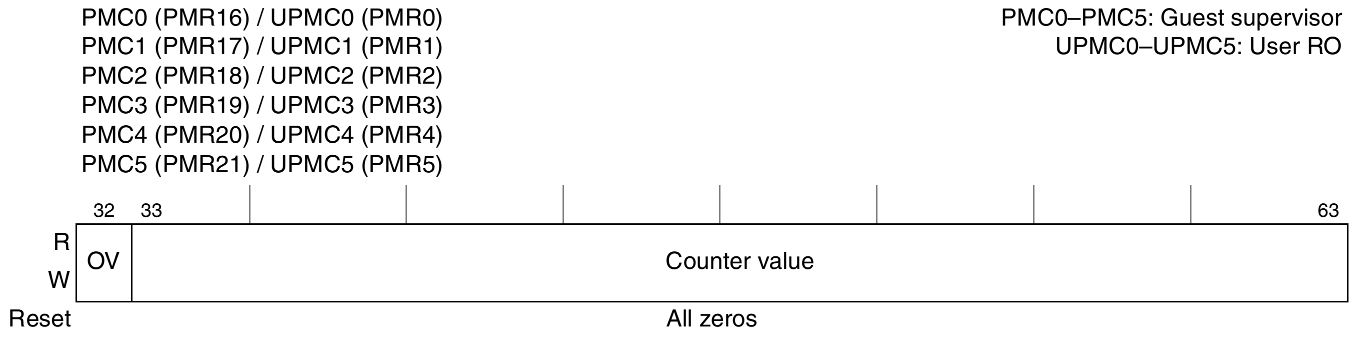

PMC0 (PMR16) / UPMC0 (PMR0)
PMC1 (PMR17) / UPMC1 (PMR1)
PMC2 (PMR18) / UPMC2 (PMR2)
PMC3 (PMR19) / UPMC3 (PMR3)
PMC4 (PMR20) / UPMC4 (PMR4)
PMC5 (PMR21) / UPMC5 (PMR5)
PMC0–PMC5: Guest supervisor
UPMC0–UPMC5: User RO
32
33
63
R
W
OV
Counter value
Reset
All zeros

Figure 2-76. Performance monitor counter (PMC0–PMC5/UPMC0–UPMC5) registers

This table describes the PMC register fields.

Table 2-72. PMC0–PMC5/UPMC0–UPMC5 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.

NOTE

The counters will stop counting (freeze) when the core enters debug halt mode, or when the core enters a low-power mode where the core clock is disabled. The counters will resume counting when debug halt mode is exited, or when the clocks are turned back on as the low power mode is exited.

A counter enters the 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 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. If the performance monitor interrupt is directed to the guest state, interrupts are masked when MSR[EE] = 0 or MSR[GS] = 0. If the performance monitor interrupt is directed to the hypervisor, interrupts are masked when MSR[EE] = 0 and MSR[GS] = 0. An exception may be signaled while the interrupt is masked, 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 the interrupt masking condition remains until after the counter leaves the overflow state (msb becomes 0), or 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. Initialize counters and configure control registers using **mtpmr** instructions.
3. Release the counters by clearing PMGC0[FAC] with a final **mtpmr** instruction.

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 e6500 processor core, 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 *EREF*.

3.1 Overview

This chapter provides information about the instruction set as implemented on the e6500 core, which is an implementation of the 64-bit Power ISA.The e6500 core 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. Those details are described in this chapter.

3.1.1 Supported Power ISA categories and unsupported instructions

The e6500 core implements the following categories, as defined in *EREF*:

- Base
- Embedded
- Alternate Time Base
- Cache Specification
- Cache Stashing
- Data Cache Block Extended Operations
- Decorated Storage
- Embedded.Enhanced Debug
- Embedded.External PID
- Embedded.Hypervisor
- Embedded.Hypervisor.LRAT
- Embedded.Page Table
- Embedded.Little-Endian
- Embedded.Multi-Threading
- Embedded.Performance Monitor
- Embedded.Processor Control
- Embedded.Cache Locking
- Enhanced Reservations
- External Proxy

- Floating Point and Floating Point.Record
- Vector
- Wait
- 64-Bit

The following table lists Power ISA 2.06 instructions defined in the previous Power ISA categories list that are not supported on the e6500 core. 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
64	**divde[o][.]**	Divide Doubleword Extended	—
64	**divdeu[o][.]**	Divide Doubleword Extended Unsigned	—
Base	**divwe[o][.]**	Divide Word Extended	—
Base	**divweu[o][.]**	Divide Word Extended Unsigned	—
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
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	**fctidu[.]**	Floating Convert to Integer Doubleword Unsigned	—
Floating Point	**fctiduz[.]**	Floating Convert to Integer Doubleword Unsigned with Round Toward Zero	—
Floating Point	**fctiwu[.]**	Floating Convert to Integer Word Unsigned	—
Floating Point	**fctiwuz[.]**	Floating Convert to Integer Word Unsigned with Round Toward 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	—

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

Category	Mnemonic	Name	Notes
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.

3.2 Computation mode

EREF defines two major computation modes selectable through the state of the Computation Mode bit in the MSR (MSR[CM]). The e6500 core supports both 32-bit mode (MSR[CM] = 0) and 64-bit mode (MSR[CM] = 1). *EREF* defines two methods of a 64-bit implementation providing 32-bit mode. The e6500 core provides 32-bit mode in a manner compatible with Power Architecture processors, which implement the Server category. EIS calls this "hybrid 32-bit mode". In both 32-bit and 64-bit mode, instructions that set a 64-bit register affect all 64 bits. The computational mode controls how the effective address is interpreted, how condition register bits and XER bits are set, how the Link (LR) register is set by branch instructions in which LK = 1, and how the Count (CTR) register is tested by branch conditional instructions. In both modes, effective address computations use all 64 bits of the relevant registers and produce a 64-bit result. However, in 32-bit mode, the high-order 32 bits of the computed effective address are ignored for the purpose of addressing storage.

When executing in 32-bit mode, the upper 32 bits of the fetch address, effective addresses, DAC*x*, IAC*x*, IVPR, and GIVPR are ignored. Record forms of instructions (commonly called "dot" forms because they are specified with a "**.**" at the end of the mnemonic) produce different Condition (CR) register results for an instruction that sets a GPR based on whether the thread is in 32-bit mode or 64-bit mode. In 32-bit mode, the CR result is set based on the signed comparison of the low-order 32 bits of the result to 0. In 64-bit mode, the CR result is set based on the signed comparison of all 64 bits of the result to 0.

3.3 Instruction set summary

The e6500 core 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 logical instructions. See Section 3.4.4, "Floating-point execution model."
- AltiVec instructions—These include vector integer, logical, and single-precision floating-point arithmetic instructions. See Section 3.4.5, "AltiVec instructions."
- Load and store instructions—These include integer, floating-point, AltiVec, external PID, and decorated storage load and store instructions, along with memory synchronization 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 3.4.6, "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, and so on. See Section 3.4.7, "Processor control instructions."
- Memory synchronization instructions—These instructions are used for memory synchronizing. See Section 3.4.9, "Memory synchronization instructions."
- Memory control instructions—These instructions provide control of caches and TLBs. See Section 3.4.11, "Memory control instructions," and Section 3.4.12.4, "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 4 bytes long and are word-aligned. Byte, halfword, word, and doubleword loads and stores occur between memory and a set of thirty-two 64-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. AltiVec instructions operate on quad-word operands, which may contain integer byte, halfword, word, single-precision floating-point vector elements, and use thirty-two 128-bit vector registers (VRs) 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 A, "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 the e6500 core. 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 thread 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**). A context-synchronizing interrupt (that is, any interrupt) can be used instead of a context-synchronizing instruction, in which case 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.

When modifying registers shared between threads, shared resource synchronization may be required as described in Section 3.3.3.1, “Shared resource synchronization.”

Care must be taken when altering context associated with instruction fetch and instruction address translation. Altering INIA, IMSR, MSR[IS], MSR[GS], MSR[CM], LPIDR, or PID can cause an implicit branch, where the change in translation or how instructions are fetched causes the thread 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. It is recommended that MSR[IS], MSR[GS], and MSR[CM] 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. Because the INIA and IMSR associated with a thread can only be written when that thread is disabled, it is recommended that altering the context of that thread via its INIA or IMSR be performed only after that thread is known to be disabled by polling the appropriate TENSR bit.

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 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.)

This table 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 (CM)	None	CSI	—
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**	3
mtspr (L1CSR2)	**sync** followed by **isync**	**isync** followed by **sync**[4]	3
mtspr (LPIDR)	CSI	CSI	—
mtspr (PID)	CSI	CSI	—
tlbivax	CSI	**sync** followed by CSI	5,6,7
tlbilx	CSI	CSI	5,6
tlbwe	CSI	CSI	5,6

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 Isolated shared synchronization is required. See Section 3.3.3.1, “Shared resource synchronization.”

4 The additional **sync** following the **isync** after the **mtspr** is done is required if software is turning off stashing by writing 0 to the stash ID field of the register. The **sync** ensures that any pending stash operations have finished.

[5] 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.

[6] 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 a **sync** and by a context-synchronizing instruction. Note that such synchronization does not guarantee these completions for other threads in the processor core. If these completions must be ensured on other threads on the processor core, either **tlbsync** must be used for **tlbivax** invalidations or **tlbilx**; **isync**; **sync** must be executed on the other threads.

[7] To ensure 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 e6500 core, this code should be protected by a mutual exclusion lock such that only one processor at a time is executing this sequence because multiple **tlbsync** operations on the CoreNet interface may cause the integrated device to hang.

This table identifies the software synchronization requirements for instruction fetch and/or execution for context-altering instructions that require synchronization.

Table 3-3. Instruction fetch and/or execution synchronization requirements

Context-Altering Instruction or Event	Required Before	Required After	Notes
mtmsr (CM)	None	CSI	—
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 (CDCSR0)	None	CSI	1
mtspr (IVOR*n*)	None	CSI	1
mtspr (IVPR*n*)	None	CSI	1
mtspr (L1CSR0, L1CSR1, L1CSR2)	**sync** followed by **isync**	**isync**	2
mtspr (LPIDR)	None	CSI	—
mtspr (MAS*n*)	None	CSI	3
mtspr (PID)	None	CSI	—
mtspr (PWRMGTCR0)	None	CSI	1
mttmr (INIA*n*)	**mtspr** TENC confirmed by **mfspr** TENSR	**mtspr** TENS confirmed by **mfspr** TENSR[4]	—
tlbivax	None	CSI	5,6
tlbilx	None	CSI	5
tlbwe	None	CSI	5

[1] Shared synchronization is required to synchronize the change in the other threads.

[2] Isolated shared synchronization is required. See Section 3.3.3.1, "Shared resource synchronization."

[3] MAS register changes require a CSI before subsequent instructions that use those updated values, such as a **tlbwe**, **tlbre**, **tlbilx**, **tlbsx**, and **tlbivax**. Typically, software performs several MAS updates and then performs a single **isync** prior to executing the TLB management instruction.

[4] This sequence is required to enable the thread once the new NIA has been written.

[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 a **sync** and by a context-synchronizing instruction.

[6] To ensure 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 e6500 core, this code should be protected by a mutual exclusion lock such that only one processor at a time is executing this sequence because multiple **tlbsync** operations on the CoreNet interface may cause the integrated device to hang.

This table identifies the software synchronization requirements for non-context-altering instructions that require synchronization.

Table 3-4. Special synchronization requirements

Non-Context-Altering Instruction or Event	Required Before	Required After	Notes
mtspr (BUCSR)	None	**isync**	—
mtspr (DAC*n*)	None	CSI and changing MSR[DE] from 0 to 1	[1]
mtspr (DBCR*n*)	None	CSI and changing MSR[DE] from 0 to 1	[1]
mtspr (DBSR)	None	CSI and changing MSR[DE] from 0 to 1	[1]
mtspr (DBSRWR)	None	CSI and changing MSR[DE] from 0 to 1	[1]
mtspr (EPCR[DUVD])	None	CSI and changing MSR[DE] from 0 to 1	[1,2]
mtspr (HID*n*)	**msync** followed by **isync**	**isync**	[3]
mtspr (IAC*n*)	None	CSI and changing MSR[DE] from 0 to 1	[1]
mtspr (MMUCSR0)	None	**isync**	[3]
mtspr (NSPD)	None	**isync**	—
mtspr (PPR32)	None	CSI	—
mttmr (TPRI*n*)	None	executing thread: **isync**; other thread: shared register synchronization	[3]

[1] Synchronization requirements for changing any debug facility registers require that the changes be followed by a CSI 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 occur in the debug interrupt routine when MSR[DE] = 0, and the subsequent return via **rfdi** from the debug routine is likely to write MSR[DE] back to 1, which accomplishes the required synchronization. Software should only make changes to the debug facility registers when MSR[DE] = 0. Note that results of changing debug registers may be seen at any time after the debug facility is changed, but are not guaranteed until the required synchronization is performed. This means that changing debug resources that cause debug events to trigger in the current instruction stream is an unreliable construct for software to use.

[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 earlier data or instruction execution tables.

[3] Shared synchronization is required to synchronize the change in the other threads.

Synchronization requirements for updating memory-mapped registers (MMRs) are described in Section 2.2.3.1, “Synchronization requirements for memory-mapped registers.”

3.3.3.1 Shared resource synchronization

When modifying SPRs or TMRs, which are shared between threads (shown as shared in Table 2-2), if the change in the register must be synchronized in other threads, shared resource synchronization must be performed after any required synchronization operations are performed in the executing thread. One thread can context-synchronize any other thread within in the e6500 core by first disabling and then re-enabling the other thread using the following process:

1. Write to the appropriate bit of the TENC SPR.
2. Poll the corresponding TENSR bit.
3. Write the TENS bit associated with the other thread.

Some registers require that the thread performing the **mtspr** must be the only enabled thread during instruction execution and synchronization. This is called "isolated shared synchronization." To perform isolated shared synchronization, the thread first disables all other threads by writing the TENC register and polls the TENSR register to determine that all other threads are disabled. It then can perform the **mtspr** and appropriate synchronization before re-enabling the other threads by writing to the TENS register.

3.3.3.2 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 a **tlbwe**, **tlbilx**, or **tlbivax** instruction, which 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**). However to see that all the effects of a **tlbilx** are seen in other threads in the same processor, software should arrange to execute **tlbilx** in the other thread.

3.3.3.3 Context synchronization

Context-synchronizing operations include instructions **isync**, **sc**, **rfi**, **rfci**, **rfmci**, **rfdi**, **rfgi**, **ehpriv,** and most interrupts. An instruction or event is context synchronizing if it satisfies the following requirements:

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 *EREF*.)

As described in Section 3.3.3.4, "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.

EREF describes context synchronization in detail.

3.3.3.4 Execution synchronization

An instruction is execution synchronizing if it satisfies items 1 and 2 in the context synchronization requirements list (see Section 3.3.3.3, "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.

3.3.3.5 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 e6500 core instructions are described in Section 4.6, "Exceptions."

3.4 Instruction set overview

This section provides a overview of the instructions implemented in the e6500 core and highlights any special information with respect to how the e6500 core implements a particular instruction.

3.4.1 Record and overflow forms

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. In 32-bit mode, the results of the lower 32-bits of the result are compared to 0. In 64-bit mode, the results of all 64-bits are compared 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. In 64-bit mode, overflow (XER[OV]) is set if the carryout of

bit 0 is not equal to the carryout of bit 1 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 AltiVec instructions—The dot (**.**) suffix on the mnemonic for vector comparison instructions enables the update of the CR6 field. CR6 is updated with the result of the comparison operation whether the relation holds true or false for all elements or whether all the elements are within bounds (for **vcmpbfp**).
- 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 (EAs) used to determine the address where a storage operation is to be performed. There are several different forms of EA generation and some instructions, such as integer load and store instructions, provide all such forms. The EA 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 **r**0, 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 if 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.

If operating in 32-bit mode, the upper 32 bits of the EA are treated as 0. In 64-bit mode, all 64 bits of the EA are used.

See *EREF* for more information.

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 XER and CR fields.

3.4.3.1.1 Integer arithmetic instructions

This table lists the integer arithmetic instructions implemented on the e6500 core.

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 Doubleword	**divd** (**divd. divdo divdo.**)	**r**D,**r**A,**r**B
Divide Doubleword Unsigned	**divdu divdu. divduo divduo.**	**r**D,**r**A,**r**B
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 Doubleword	**mulhd** (**mulhd.**)	**r**D,**r**A,**r**B
Multiply High Doubleword Unsigned	**mulhdu** (**mulhdu.**)	**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 Doubleword	**mulld** (**mulld. mulldo mulldo.**)	**r**D,**r**A,**r**B
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
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 A, "Simplified Mnemonics."

An implementation that executes instructions with the Overflow Exception Enable (OE) bit set or that sets the Carry (CA) bit can either execute these instructions slowly or prevent execution of the next instruction until the operation completes. Chapter 10, "Execution Timing," describes how the e6500 core 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 or a 64-bit result based on computation mode, 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**.

This table lists integer compare instructions. The L bit can be either 0 (for a 32-bit compare) or 1 (for a 64-bit compare), regardless of the computation mode.

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 A, "Simplified Mnemonics."

3.4.3.1.3 Integer logical instructions

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

Table 3-7. Integer logical instructions

Name	Mnemonic	Syntax	Implementation Notes
AND	**and (and.)**	rA,rS,rB	—
AND Immediate	**andi.**	rA,rS,UIMM	—
AND Immediate Shifted	**andis.**	rA,rS,UIMM	—
AND with Complement	**andc** (**andc.**)	rA,rS,rB	—
Bit Permute Doubleword	**bpermd**	rA,rS,rB	—
Compare Bytes	**cmpb**	rA,rS,rB	—
Count Leading Zeros Word	**cntlzw (cntlzw.**)	rA,rS	—
Count Leading Zeros Doubleword	**cntlzd (cntlzd.**)	rA,rS	—
Equivalent	**eqv** (**eqv.**)	rA,rS,rB	—

Table 3-7. Integer logical instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Extend Sign Byte	**extsb** (**extsb.**)	rA,rS	—
Extend Sign Half Word	**extsh** (**extsh.**)	rA,rS	—
Extend Sign Word	**extsw** (**extsw.**)	rA,rS	—
NAND	**nand** (**nand.**)	rA,rS,rB	—
NOR	**nor** (**nor.**)	rA,rS,rB	—
OR	**or** (**or.**)	rA,rS,rB	—
OR Immediate	**ori**	rA,rS,UIMM	**ori r0,r0,0** is the preferred form for a no-op. At dispatch it may enter the completion queue but not to an execution unit.
OR Immediate Shifted	**oris**	rA,rS,UIMM	—
OR with Complement	**orc** (**orc.**)	rA,rS,rB	—
Parity Doubleword	**prtyd**	rA,rS	—
Parity Word	**prtyw**	rA,rS	—
Population Count Byte	**popcntb**	rA,rS	—
Population Count Doubleword	**popcntd**	rA,rS	—
Population Count Word	**popcntw**	rA,rS	—
XOR	**xor** (**xor.**)	rA,rS,rB	—
XOR Immediate	**xori**	rA,rS,UIMM	—
XOR Immediate Shifted	**xoris**	rA,rS,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 results are 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 A, "Simplified Mnemonics," lists simplified mnemonics that allow simpler coding of frequently 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 Doubleword then Clear Left	**rldcl** (**rldcl.**)	rA,rS,rB,MB
Rotate Left Doubleword then Clear Right	**rldcr** (**rldcr.**)	rA,rS,rB,ME
Rotate Left Doubleword Immediate then Clear	**rldic** (**rldic.**)	rA,rS,SH,MB
Rotate Left Doubleword Immediate then Clear Left	**rldicl** (**rldicl.**)	rA,rS,SH,MB

Table 3-8. Integer rotate instructions (continued)

Name	Mnemonic	Syntax
Rotate Left Doubleword Immediate then Clear Right	**rldicr** (**rldicr.**)	**r**A,**r**S,SH,ME
Rotate Left Doubleword Immediate then Mask Insert	**rldimi**(**rldimi.**)	**r**A,**r**S,SH,MB
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 A, “Simplified Mnemonics,” shows how to simplify coding of such shifts. Multiple-precision shifts can be programmed as described in *EREF*.

Table 3-9. Integer shift instructions

Name	Mnemonic	Syntax
Shift Left Doubleword	**sld** (**sld.**)	**r**A,**r**S,**r**B
Shift Left Word	**slw** (**slw.**)	**r**A,**r**S,**r**B
Shift Right Doubleword	**srd** (**srd.**)	**r**A,**r**S,**r**B
Shift Right Word	**srw** (**srw.**)	**r**A,**r**S,**r**B
Shift Right Algebraic Doubleword Immediate	**sradi** (**sradi.**)	**r**A,**r**S,SH
Shift Right Algebraic Word Immediate	**srawi** (**srawi.**)	**r**A,**r**S,SH
Shift Right Algebraic Doubleword	**srad**(**srad.**)	**r**A,**r**S,**r**B
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. The e6500 core 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
- AltiVec load instructions
- AltiVec store instructions
- Memory synchronization instructions

- External PID load and store instructions, which are described in Section 3.4.12.3, "External PID load and store instructions"
- Decorated storage load and store instructions, which are described in Section 3.4.3.2.10, "Decorated load and store instructions

Implementation notes:

The following notes describe how the e6500 core handles misalignment:

1. The e6500 core 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.
2. A word or halfword memory access requires multiple accesses if it crosses a doubleword boundary but not if it crosses a natural boundary.
3. Frequent use of misaligned memory accesses can greatly degrade performance.
4. Any load doubleword, word, or load halfword that crosses a doubleword boundary is interruptible, and, therefore, can restart. If the first access is 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.
5. 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 instructions, as well as floating-point load and store instructions, contain update forms that 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**	rD,d(rA)
Load Byte and Zero Indexed	**lbzx**	rD,rA,rB
Load Byte and Zero with Update	**lbzu**	rD,d(rA)
Load Byte and Zero with Update Indexed	**lbzux**	rD,rA,rB
Load Doubleword	**ld**	rD,d(rA)
Load Doubleword Indexed	**ldx**	rD,rA,rB

Table 3-10. Integer load instructions (continued)

Name	Mnemonic	Syntax
Load Doubleword with Update	**ldu**	**r**D,**d**(**r**A)
Load Doubleword with Update Indexed	**ldux**	**r**D,**r**A,**r**B
Load Half Word and Zero	**lhz**	**r**D,**d**(**r**A)
Load Half Word and Zero Indexed	**lhzx**	**r**D,**r**A,**r**B
Load Half Word and Zero with Update	**lhzu**	**r**D,**d**(**r**A)
Load Half Word and Zero with Update Indexed	**lhzux**	**r**D,**r**A,**r**B
Load Half Word Algebraic	**lha**	**r**D,**d**(**r**A)
Load Half Word Algebraic Indexed	**lhax**	**r**D,**r**A,**r**B
Load Half Word Algebraic with Update	**lhau**	**r**D,**d**(**r**A)
Load Half Word 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
Load Word Algebraic	**lwa**	**r**D,**d**(**r**A)
Load Word Algebraic Indexed	**lwax**	**r**D,**r**A,**r**B
Load Word Algebraic with Update Indexed	**lwaux**	**r**D,**r**A,**r**B

Some implementations execute the load algebraic (**lha**, **lhax**, **lhau**, **lhaux**, **lwa**, **lwax**, **lwaux**) instructions with greater latency than other types of load instructions. The e6500 core executes these instructions with the same latency as other load instructions.

The e6500 core also contains load and store instructions for atomic memory accesses. These are described in Section 3.4.9, "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 Doubleword	**std**	**r**S,**d**(**r**A)
Store Doubleword Indexed	**stdx**	**r**S,**r**A,**r**B
Store Doubleword with Update	**stdu**	**r**S,**d**(**r**A)

Table 3-11. Integer store instructions (continued)

Name	Mnemonic	Syntax
Store Doubleword with Update Indexed	**stdux**	rS,rA,rB
Store Half Word	**sth**	rS,d(rA)
Store Half Word Indexed	**sthx**	rS,rA,rB
Store Half Word with Update	**sthu**	rS,d(rA)
Store Half Word with Update Indexed	**sthux**	rS,rA,rB
Store Word	**stw**	rS,d(rA)
Store Word Indexed	**stwx**	rS,rA,rB
Store Word with Update	**stwu**	rS,d(rA)
Store Word with Update Indexed	**stwux**	rS,rA,rB

3.4.3.2.4 Integer load and store with byte-reverse instructions

The following table describes integer load and store with byte-reverse instructions. *EREF* supports true little-endian on a per-page basis.

Table 3-12. Integer load and store with byte-reverse instructions

Name	Mnemonic	Syntax
Load Doubleword Byte-Reverse Indexed	**ldbrx**	rD,rA,rB
Load Halfword Byte-Reverse Indexed	**lhbrx**	rD,rA,rB
Load Word Byte-Reverse Indexed	**lwbrx**	rD,rA,rB
Store Doubleword Byte-Reverse Indexed	**stdbrx**	rD,rA,rB
Store Halfword Byte-Reverse Indexed	**sthbrx**	rS,rA,rB
Store Word Byte-Reverse Indexed	**stwbrx**	rS,rA,rB

Some implementations run the load/store byte-reverse instructions with greater latency than other types of load/store instructions. The e6500 core 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 will be 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 that are 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, causing 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) 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 *EREF*.

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 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 *EREF*.

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

Table 3-15. Floating-point store instructions (continued)

Name	Mnemonic	Operand Syntax
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 AltiVec load instructions

Separate AltiVec load instructions are used for vector operands. The load instructions load either a quad-word vector or an element of a vector. For load to left and load to right instructions, only a partial number of bytes in the quad-word vector are loaded and the remaining bytes in the VR are set to zero. These instructions are more fully defined in *AltiVec Technology Programming Environments Manual for Power ISA Processors*.

This table provides a list of the AltiVec load instructions.

Table 3-16. AltiVec load instructions

Name	Mnemonic	Operand Syntax
Load Vector Element Byte Indexed	**lvebx**	**v**D,**r**A,**r**B
Load Vector Element Halfword Indexed	**lvehx**	**v**D,**r**A,**r**B
Load Vector Element Word Indexed	**lvewx**	**v**D,**r**A,**r**B
Load Vector Element Indexed Byte Indexed	**lvexbx**	**v**D,**r**A,**r**B
Load Vector Element Indexed Halfword Indexed	**lvexhx**	**v**D,**r**A,**r**B
Load Vector Element Indexed Word Indexed	**lvexwx**	**v**D,**r**A,**r**B
Load Vector to Left Indexed	**lvtlx**	**v**D,**r**A,**r**B
Load Vector to Left Indexed LRU	**lvtlxl**	**v**D,**r**A,**r**B
Load Vector to Right Indexed	**lvtrx**	**v**D,**r**A,**r**B
Load Vector to Right Indexed LRU	**lvtrxl**	**v**D,**r**A,**r**B
Load Vector for Shift Left	**lvsl**	**v**D,**r**A,**r**B
Load Vector for Swap Merge	**lvsm**	**v**D,**r**A,**r**B
Load Vector for Shift Right	**lvsr**	**v**D,**r**A,**r**B
Load Vector with Left-Right Swap Indexed	**lvswx**	**v**D,**r**A,**r**B
Load Vector with Left-Right Swap Indexed LRU	**lvswxl**	**v**D,**r**A,**r**B
Load Vector Indexed	**lvx**	**v**D,**r**A,**r**B
Load Vector Indexed LRU	**lvxl**	**v**D,**r**A,**r**B

Implementation notes:

For **lvtlx**, **lvtlxl**, **lvtrx**, and **lvtrxl**, if the load is to caching-inhibited memory, all bytes in the referenced quad-word are accessed from memory, even if the number of bytes to be loaded is less than a quad-word.

3.4.3.2.9 AltiVec store instructions

Separate AltiVec store instructions are used for vector operands. The store instructions store either a quad-word vector or an element of a vector. For store to left and store from right instructions, only a partial number of bytes in the quad-word vector are stored. These instructions are more fully defined in the *AltiVec Technology Programming Environments Manual for Power ISA Processors*.

This table lists the AltiVec store instructions.

Table 3-17. AltiVec store instructions

Name	Mnemonic	Operand Syntax
Store Vector Element Byte Indexed	**stvebx**	vS,rA,rB
Store Vector Element Halfword Indexed	**stvehx**	vS,rA,rB
Store Vector Element Word Indexed	**stvewx**	vS,rA,rB
Store Vector Element Indexed Byte Indexed	**stvexbx**	vS,rA,rB
Store Vector Element Indexed Halfword Indexed	**stvexhx**	vS,rA,rB
Store Vector Element Indexed Word Indexed	**stvexwx**	vS,rA,rB
Store Vector from Left Indexed	**stvflx**	vS,rA,rB
Store Vector from Left Indexed LRU	**stvflxl**	vS,rA,rB
Store Vector from Right Indexed	**stvfrx**	vS,rA,rB
Store Vector from Right Indexed LRU	**stvfrxl**	vS,rA,rB
Store Vector with Left-Right Swap Indexed	**stvswx**	vS,rA,rB
Store Vector with Left-Right Swap Indexed LRU	**stvswxl**	vS,rA,rB
Store Vector Indexed	**stvx**	vS,rA,rB
Store Vector Indexed LRU	**stvxl**	vS,rA,rB

Implementation notes:

For **stvflx**, **stvflxl**, **stvfrx**, and **stvfrxl**, if the store is to caching-inhibited or write-through-required memory and the number of bytes to be stored is greater than 8, an alignment interrupt is taken.

3.4.3.2.10 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, see the reference manual for the integrated device.

The architecture defines the decorated instructions listed in the following table, which provide the EA in **r**B and the decoration in **r**A.

Table 3-18. Decorated load and store instructions

Instruction	Mnemonic	Syntax	Description
Load Byte with Decoration Indexed	**lbdx**	rD,rA,rB	The byte, halfword, word, doubleword, or floating-point doubleword addressed by EA (in **r**B) using the decoration supplied by **r**A is loaded into the target GPR **rD**.
Load Halfword with Decoration Indexed	**lhdx**	rD,rA,rB	
Load Word with Decoration Indexed	**lwdx**	rD,rA,rB	
Load Doubleword with Decoration Indexed	**lddx**	rD,rA,rB	
Load Floating-Point Doubleword with Decoration Indexed	**lfddx**	frD,rA,rB	
Store Byte with Decoration Indexed	**stbdx**	rS,rA,rB	The contents of **r**S and the decoration supplied by GPR(**r**A) are stored into byte, halfword, word, doubleword, or floating-point doubleword in storage addressed by EA (r**B**).
Store Halfword with Decoration Indexed	**sthdx**	rS,rA,rB	
Store Word with Decoration Indexed	**stwdx**	rS,rA,rB	
Store Doubleword with Decoration Indexed	**stddx**	rS,rA,rB	
Store Floating-Point Doubleword with Decoration Indexed	**stfddx**	frS,rA,rB	
Decorated Storage Notify	**dsn**	rA,rB	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 caching 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 e6500 core requires that decorated load instructions (**lbdx**, **lhdx**, **lwdx**, **lddx**, **lfddx**) have write permissions when the target data address of the instruction is in storage that is caching inhibited because decorated load operations defined by integrated devices that contain the e6500 core can modify memory.

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 the e6500 core, only the low-order 4 bits of the decoration in **r**A are implemented.

3.4.4 Floating-point execution model

The e6500 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, see the “Operand Conventions” chapter in *EREF*.

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 and/or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands.

The floating-point instructions follow 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 pre-normalized. 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.

EREF supports a floating-point system as defined in the IEEE 754 standard, but requires software support to conform with that standard. All floating-point operations conform to the IEEE 754 standard, except if software sets the non-IEEE mode bit (NI) in FPSCR. The e6500 core is in the non-denormalized mode when the NI bit is set in 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.

3.4.4.1.1 Floating-point arithmetic instructions

This table lists the floating-point arithmetic instructions.

Table 3-19. 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.

This table lists the floating-point multiply-add instructions.

Table 3-20. 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

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 *EREF*.

This table lists the floating-point rounding and conversion instructions.

Table 3-21. Floating-point rounding and conversion instructions

Name	Mnemonic	Operand Syntax
Floating Convert from Integer Doubleword	**fcfid** (**fcfid.**)	**fr**D,**fr**B
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 Convert to Integer Doubleword	**fctid** (**fctid.**)	**fr**D,**fr**B
Floating Convert to Integer Doubleword with Round Toward Zero	**fctidz** (**fctidz.**)	**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).

This table lists the floating-point compare instructions.

Table 3-22. 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 (FPSCR) register instructions

Every FPSCR instruction appears to synchronize the effects of all floating-point instructions executed by a given thread. Executing an FPSCR instruction ensures that all floating-point instructions previously initiated by the given thread appear to have completed before the FPSCR instruction is initiated and that no subsequent floating-point instructions appear to be initiated by the given thread until the FPSCR instruction of the thread has completed.

This table lists the FPSCR instructions.

Table 3-23. Floating-Point Status and Control (FPSCR) 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 e6500 core.

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.

This table lists the floating-point move instructions.

Table 3-24. 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 AltiVec instructions

AltiVec instructions use vector registers (VRs) to provide single instruction multiple data (SIMD) computation using byte, halfword, and word elements. Depending on the instruction, computation can be performed using unsigned or signed and modulo or saturating integer arithmetic, as well as single-precision floating-point operations.

The e6500 core implements the AltiVec instruction set as described in *AltiVec Technology Programming Environments Manual for Power ISA Processors*. AltiVec instructions are listed here by function.

3.4.5.1 AltiVec integer instructions

Most integer instructions have both signed and unsigned versions and many have both modulo (wrap-around) and saturating clamping modes. Saturation occurs whenever the result of a saturating instruction does not fit in the result field. Unsigned saturation clamps results to zero on underflow and to the maximum positive integer value ($2^n – 1$, for example, 255 for byte fields) on overflow. Signed saturation clamps results to the smallest representable negative number ($–2^{n–1}$, for example, –128 for byte fields) on underflow, and to the largest representable positive number ($2^{n–1}–1$, for example, +127 for byte fields) on overflow. When a modulo instruction is used, the resultant number truncates overflow or underflow for the length (byte, halfword, word, quad-word) and type of operand (unsigned, signed). AltiVec provides a way to detect saturation and sets the SAT bit in the Vector Status and Control (VSCR[SAT]) register in a saturating instruction.

Borderline cases that generate results equal to saturation values, for example unsigned $0 + 0 \rightarrow 0$ and unsigned byte $1 + 254 \rightarrow 255$, are not considered saturation conditions and do not cause VSCR[SAT] to be set.

This table lists the AltiVec integer arithmetic instructions.

Table 3-25. AltiVec integer arithmetic instructions

Name	Mnemonic	Operand Syntax
Vector Absolute Differences Unsigned [Byte, Halfword, Word]	**vabsdub** **vabsduh** **vabsduw**	vD,vA,vB
Vector Add Unsigned Integer [b,h,w] Modulo	**vaddubm** **vadduhm** **vadduwm**	vD,vA,vB
Vector Add Unsigned Integer [b,h,w] Saturate	**vaddubs** **vadduhs** **vadduws**	vD,vA,vB
Vector Add Signed Integer [b,h,w] Saturate	**vaddsbs** **vaddshs** **vddsws**	vD,vA,vB
Vector Add and Write Carry-Out Unsigned Word	**vaddcuw**	vD,vA,vB
Vector Subtract Unsigned Integer Modulo [b,h,w]	**vsububm** **vsubuhm** **vsubuwm**	vD,vA,vB
Vector Subtract Unsigned Integer Saturate [b,h,w]	**vsububs** **vsubuhs** **vsubuws**	vD,vA,vB
Vector Subtract Signed Integer Saturate [b,h,w]	**vsubsbs** **vsubshs** **vsubsws**	vD,vA,vB
Vector Subtract and Write Carry-Out Unsigned Word	**vsubcuw**	vD,vA,vB

Table 3-25. AltiVec integer arithmetic instructions (continued)

Name	Mnemonic	Operand Syntax
Vector Multiply Odd Unsigned Integer [b,h] Modulo	**vmuloub** **vmulouh**	vD,vA,vB
Vector Multiply Odd Signed Integer [b,h] Modulo	**vmulosb** **vmulosh**	vD,vA,vB
Vector Multiply Even Unsigned Integer [b,h] Modulo	**vmuleub** **vmuleuh**	vD,vA,vB
Vector Multiply Even Signed Integer [b,h] Modulo	**vmulesb** **vmulesh**	vD,vA,vB
Vector Multiply-High and Add Signed Halfword Saturate	**vmhaddshs**	vD,vA,vB,vC
Vector Multiply-High Round and Add Signed Halfword Saturate	**vmhraddshs**	vD,vA,vB,vC
Vector Multiply-Low and Add Unsigned Halfword Modulo	**vmladduhm**	vD,vA,vB,vC
Vector Multiply-Sum Unsigned Integer [b,h] Modulo	**vmsumubm** **vmsumuhm**	vD,vA,vB,vC
Vector Multiply-Sum Signed Halfword Saturate	**vmsumshs**	vD,vA,vB,vC
Vector Multiply-Sum Unsigned Halfword Saturate	**vmsumuhs**	vD,vA,vB,vC
Vector Multiply-Sum Mixed Sign Byte Modulo	**vmsummbm**	vD,vA,vB,vC
Vector Multiply-Sum Signed Halfword Modulo	**vmsumshm**	vD,vA,vB,vC
Vector Sum Across Signed Word Saturate	**vsumsws**	vD,vA,vB
Vector Sum Across Partial (1/2) Signed Word Saturate	**vsum2sws**	vD,vA,vB
Vector Sum Across Partial (1/4) Unsigned Byte Saturate	**vsum4ubs**	vD,vA,vB
Vector Sum Across Partial (1/4) Signed Integer Saturate	**vsum4sbs** **vsum4shs**	vD,vA,vB
Vector Average Unsigned Integer [b,h,w]	**vavgub** **vavguh** **vavguw**	vD,vA,vB
Vector Average Signed Integer [b,h,w]	**vavgsb** **vavgsh** **vavgsw**	vD,vA,vB
Vector Maximum Unsigned Integer [b,h,w]	**vmaxub** **vmaxuh** **vmaxuw**	vD,vA,vB
Vector Maximum Signed Integer [b,h,w]	**vmaxsb** **vmaxsh** **vmaxsw**	vD,vA,vB

Table 3-25. AltiVec integer arithmetic instructions (continued)

Name	Mnemonic	Operand Syntax
Vector Minimum Unsigned Integer [b,h,w]	**vminub** **vminuh** **vminuw**	vD,vA,vB
Vector Minimum Signed Integer [b,h,w]	**vminsb** **vminsh** **vminsw**	vD,vA,vB

3.4.5.2 AltiVec integer compare instructions

The vector integer compare instructions algebraically or logically compare the contents of the elements in vector register **v**A with the contents of the elements in **v**B. Each compare result vector is comprised of TRUE (0xFF, 0xFFFF, 0xFFFFFFFF) or FALSE (0x00, 0x0000, 0x00000000) elements of the size specified by the compare source operand element (byte, halfword, or word). The result vector can be directed to any vector register and can be manipulated with any of the instructions as normal data (for example, combining condition results). Vector compares provide equal-to and greater-than predicates. Others are synthesized from these by logically combining or inverting result vectors.

This table lists the AltiVec integer compare instructions.

Table 3-26. AltiVec integer compare instructions

Name	Mnemonic	Syntax
Vector Compare Greater Than Unsigned Integer [b,h,w]	**vcmpgtub**[.] **vcmpgtuh**[.] **vcmpgtuw**[.]	vD,vA,vB
Vector Compare Greater Than Signed Integer [b,h,w]	**vcmpgtsb**[.] **vcmpgtsh**[.] **vcmpgtsw**[.]	vD,vA,vB
Vector Compare Equal To Unsigned Integer [b,h,w]	**vcmpequb**[.] **vcmpequh**[.] **vcmpequw**[.]	vD,vA,vB

3.4.5.3 AltiVec integer logical instructions

The AltiVec integer logical instructions shown in the following table perform bit-parallel operations on the operands.

Table 3-27. AltiVec integer logical instructions

Name	Mnemonic	Syntax
Vector Logical AND	**vand**	vD,vA,vB
Vector Logical OR	**vor**	vD,vA,vB
Vector Logical XOR	**vxor**	vD,vA,vB

Table 3-27. AltiVec integer logical instructions (continued)

Name	Mnemonic	Syntax
Vector Logical AND with Complement	**vandc**	vD,vA,vB
Vector Logical NOR	**vnor**	vD,vA,vB

3.4.5.4 AltiVec integer rotate and shift instructions

This table lists the AltiVec integer rotate instructions.

Table 3-28. AltiVec integer rotate instructions

Name	Mnemonic	Syntax
Vector Rotate Left Integer [b,h,w]	**vrlb** **vrlh** **vrlw**	vD,vA,vB

This table lists the AltiVec integer shift instructions.

Table 3-29. AltiVec integer shift instructions

Name	Mnemonic	Syntax
Vector Shift Left Integer [b,h,w]	**vslb** **vslh** **vslw**	vD,vA,vB
Vector Shift Right Integer [b,h,w]	**vsrb** **vsrh** **vsrw**	vD,vA,vB
Vector Shift Right Algebraic Integer [b,h,w]	**vsrab** **vsrah** **vsraw**	vD,vA,vB

3.4.5.5 AltiVec floating-point instructions

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

- Arithmetic
- Rounding and conversion
- Compare
- Estimate

The AltiVec floating-point data format complies with the ANSI/IEEE-754 standard. A quantity in this format represents one of the following:

- A signed normalized number
- A signed denormalized number
- A signed zero
- A signed infinity

- A quiet not a number (QNaN)
- A signaling NaN (SNaN)

Operations perform to a Java/IEEE/C9X-compliant subset of the IEEE standard. AltiVec does not report IEEE exceptions, but rather produces default results as specified by the Java/IEEE/C9X standard. For further details on exceptions, see *AltiVec Technology Programming Environments Manual for Power ISA Processors*.

3.4.5.5.1 AltiVec floating-point behavior for special case data

This section describes the implementation-specific features of AltiVec with respect to floating-point data types as implemented on the e6500 core. The descriptions in this section cover both Java and non-Java modes and include the following special case data:

- Denorm data for all instructions
- NaNs, denorms, and zeros for compare, min, and max operations
- Zero and Nan data for round to float integral operations

The following list describes the e6500 core behavior in various special cases:

- The core defaults to Java mode (VSCR[NJ] = 0).
- The core handles NaN operands the same way regardless of Java or non-Java mode. If any operand is a NaN, the e6500 core returns a NaN result.
- If the proper result can be determined to be a NaN, the core ignores any denorm inputs and returns the NaN result.
- The core handles most denorms in Java mode by taking an Altivec assist interrupt, but, for some instructions, the e6500 core can produce the exact result without taking the interrupt.
- VFPU detects underflows and production of denormalized numbers on vector float results before rounding, not after.
- The **vrefp** instruction returns the exact answer for operand of power of two. For example: vrefp(+2.0) = +0.50.
- The **vrefp** instruction does not overflow. Reciprocal of smallest normalized number: mantissa = 1.0, unbiased exponent = -126. Result is: 1.0 x 2^ 126, so overflow is not possible.
- The **vrefp** instruction can underflow before rounding. Reciprocal of largest number: mantissa 1.11---111, unbiased exponent +127. Result: unnormalized mantissa has at least one leading zero, while the exponent before normalization is -127. Therefore, the intermediate result before rounding is a denormalized number.
- The **vrsqrtefp** instruction does not round the least significant bit of the mantissa.
- VFPU executes **mfvscr** and **mtvscr**.
- **vctuxs**: When the input operand falls into this range: -1.0 < vB * 2^UIMM < 0.0, VFPU produces result of 0x0000_0000 but does not write 1 to VSCR[SAT]. The supporting argument is if the intermediate value of vB * 2^UIMM is a negative fraction, then the integer approximation of that negative fraction is representable as an unsigned integer with a value of 0x0000_0000. Therefore, the result is not considered saturated and VSCR[SAT] should not be set.

Table 3-30, Table 3-31, Table 3-32, Table 3-33, and Table 3-34 detail the implementation-specific behaviors for AltiVec floating-point operation. The term "trap" means that an AltiVec assist interrupt is taken and software must retrieve and emulate the instruction to provide correct behavior.

Table 3-30. AltiVec denorm handling

Instruction	Input Denorm detected		Output Denorm detected	
	Java	Non-Java	Java	Non-Java
vaddfp, **vsubfp**, **vmaddfp**, **vnmsubfp**	Trap (unless result is NaN)[1]	Input treated as correctly signed zero	Trap	Result squashed to correctly signed zero
vrefp	Trap	Denorm squashed to zero, returning ± infinity	Trap	Result squashed to zero
vrsqrtefp	Trap	Denorm squashed to zero, returning ± infinity	Never produces a denorm	Never produces a denorm
vlogefp	Trap	Denorm squashed to zero, returning -infinity	Never produces a denorm	Never produces a denorm
vexptefp	Result is +1.0	Input squashed to zero, output result is +1.0	Trap	Result squashed to zero
vcfux, **vcfsx**	never sees denorms			
vctsxs, **vctuxs**	Trap[1]	Output result is 0x0	Never produces a denorm	Never produces a denorm

[1] If the instruction has a denorm operand but produces a NaN result, the e6500 core returns the NaN result. For example, (0 * infinity) + denorm returns a NaN result and does not cause an AltiVec assist interrupt in Java mode.

Table 3-31. AltiVec floating-point compare, min, and max in non-Java mode

VA	VB	vminfp	vmaxfp	vcmpgtfp	vcmpgefp	vcmpeqfp	vcmpbfp	
							LE	GE
NaN_A	-	qNaN_A	qNaN_A	False	False	False	0	0
-	NaN_B	qNaN_B	qNaN_B	False	False	False	0	0
+Den_A	-B	-B	+Zero	True	True	False	0	0
-Den_A	-B	-B	-Zero	True	True	False	0	0
+Den_A	+B	+Zero	+B	False	False	False	1	1
-Den_A	+B	-Zero	+B	False	False	False	1	1
-A	+Den_B	-A	+Zero	False	False	False	1	0
-A	-Den_B	-A	-Zero	False	False	False	1	0
+A	+Den_B	+Zero	+A	True	True	False	0	1
+A	-Den_B	-Zero	+A	True	True	False	0	1
+Den_A/+Zero	+Den_B/+Zero	+Zero	+Zero	False	True	True	1	1
+Den_A/+Zero	-Den_B/-Zero	-Zero	+Zero	False	True	True	1	1

Table 3-31. AltiVec floating-point compare, min, and max in non-Java mode

VA	VB	vminfp	vmaxfp	vcmpgtfp	vcmpgefp	vcmpeqfp	vcmpbfp	
							LE	GE
-Den_A/-Zero	+Den_B/+Zero	-Zero	+Zero	False	True	True	1	1
-Den_A/-Zero	-Den_B/-Zero	-Zero	-Zero	False	True	True	1	1

Table 3-32. AltiVec floating-point compare, min, and max in Java mode

VA	VB	vminfp	vmaxfp	vcmpgtfp	vcmpgefp	vcmpeqfp	vcmpbfp	
							LE	GE
NaN_A	-	qNaN_A	qNaN_A	False	False	False	0	0
-	NaN_B	qNaN_B	qNaN_B	False	False	False	0	0
+Den_A	-B	-B	+Den_A	True	True	False	0	0
-Den_A	-B	-B	-Den_A	True	True	False	0	0
+Den_A	+B	+Den_A	+B	False	False	False	1	1
-Den_A	+B	-Den_A	+B	False	False	False	1	1
-A	+Den_B	-A	+Den_B	False	False	False	1	0
-A	-Den_B	-A	-Den_B	False	False	False	1	0
+A	+Den_B	+Den_B	+A	True	True	False	0	1
+A	-Den_B	-Den_B	+A	True	True	False	0	1
+Den_A	+-Zero	+-Zero	+Den_A	True	True	False	0	1
-Den_A	+-Zero	-Den_A	+-Zero	False	False	False	1	0
+-Zero	+Den_B	+-Zero	+Den_B	False	False	False	1	1
+-Zero	-Den_B	-Den_B	+-Zero	True	True	False	0	0
-Den_A	+Den_B	-Den_A	+Den_B	False	False	False	1	See Note [1]
+Den_A	-Den_B	-Den_B	+Den_A	True	True	False	0	See Note[1]
-Den_A	-Den_B	See Note[1]						0
+Den_A	+Den_B							1

[1] Result depends on input operands.

Table 3-33. AltiVec round to integer instructions in non-Java mode

VB sign	VB exponent	vrfin	vrfiz	vrfip	vrfim
neg	127 > exp > 24	VB	VB	VB	VB
neg	23 > exp > 0	Round towards nearest	Truncate fraction	Round towards +Inf	Round towards -Inf
neg	exp = -1	Round to nearest	-zero	-zero	-1.0
neg	-2 > exp > -126	-zero	-zero	-zero	-1.0
neg	input is denorm	-zero	-zero	-zero	-zero
neg	input is zero	-zero	-zero	-zero	-zero
pos	input is zero	+zero	+zero	+zero	+zero
pos	input is denorm	+zero	+zero	+zero	+zero
pos	-126 < exp < -2	+zero	+zero	+1.0	+zero
pos	exp = -1	Round towards nearest	+zero	+1.0	+zero
pos	0 < exp < 23	Round towards nearest	Truncate fraction	Round towards +Inf	Round towards -Inf
pos	24 < exp < 126	VB	VB	VB	VB

Table 3-34. AltiVec round to integer instructions in Java mode

VB sign	VB exponent	vrfin	vrfiz	vrfip	vrfim
neg	127 > exp > 24	VB	VB	VB	VB
neg	23 > exp > 0	Round towards nearest	Truncate fraction	Round towards +Inf	Round towards -Inf
neg	exp = -1	Round to nearest	-zero	-zero	-1.0
neg	-2 > exp > -126	-zero	-zero	-zero	-1.0
neg	input is denorm	Trap	Trap	Trap	Trap
neg	input is zero	-zero	-zero	-zero	-zero
pos	input is zero	+zero	+zero	+zero	+zero
pos	input is denorm	Trap	Trap	Trap	Trap
pos	-126 < exp < -2	+zero	+zero	+1.0	+zero
pos	exp = -1	Round towards nearest	+zero	+1.0	+zero
pos	0 < exp < 23	Round to nearest	Truncate fraction	Round to +Inf	Round to -Inf
pos	24 < exp < 126	VB	VB	VB	VB

The round-to-integer instructions never produce denorms.

3.4.5.5.2 Floating-point division and square root

AltiVec instructions do not have division or square root instructions. AltiVec implements Vector Reciprocal Estimate Floating-Point (**vrefp**) and Vector Reciprocal-Square-Root Estimate Floating-Point (**vrsqrtefp**) instructions along with a Vector Negative Multiply-Subtract Floating-Point (**vnmsubfp**) instruction assisting in the Newton-Raphson refinement of the estimates. To accomplish division, simply multiply by the reciprocal estimate of the dividend ($x/y = x * 1/y$) and square root by multiplying the original number by the reciprocal of the square root estimate ($\sqrt{x} = x * 1/\sqrt{x}$). In this way, AltiVec provides inexpensive divides and square-roots that are fully pipelined, sub-operation scheduled, and faster even than many hardware dividers. Software methods are available to further refine these to correct IEEE results. See *AltiVec Technology Programming Environments Manual for Power ISA Processors* for a more complete description of floating-point division and square root computation.

3.4.5.5.3 AltiVec floating-point arithmetic instructions

This table lists the AltiVec floating-point arithmetic instructions.

Table 3-35. AltiVec floating-point arithmetic instructions

Name	Mnemonic	Syntax
Vector Add Floating-Point	**vaddfp**	vD,vA,vB
Vector Subtract Floating-Point	**vsubfp**	vD,vA,vB
Vector Maximum Floating-Point	**vmaxfp**	vD,vA,vB
Vector Minimum Floating-Point	**vminfp**	vD,vA,vB

3.4.5.5.4 AltiVec floating-point multiply-add instructions

This table lists the AltiVec floating-point multiply-add instructions.

Table 3-36. AltiVec floating-point multiply-add instructions

Name	Mnemonic	Syntax
Vector Multiply-Add Floating-Point	**vmaddfp**	vD,vA,vC,vB
Vector Negative Multiply-Subtract Floating-Point	**vnmsubfp**	vD,vA,vC,vB

3.4.5.5.5 Floating-point rounding and conversion instructions

All AltiVec floating-point arithmetic instructions use the IEEE default rounding mode, round-to-nearest. AltiVec does not provide the IEEE directed rounding modes.

AltiVec provides separate instructions for converting floating-point numbers to integral floating-point values for all IEEE rounding modes as follows:

- Round-to-nearest (**vrfin**) (round)
- Round-toward-zero (**vrfiz**) (truncate)
- Round-toward-minus-infinity (**vrfim**) (floor)
- Round-toward-positive-infinity (**vrfip**) (ceiling)

Floating-point conversions to integers (**vctuxs**, **vctsxs**) use round-toward-zero (truncate).

This table lists the floating-point rounding and conversion instructions.

Table 3-37. AltiVec floating-point rounding and conversion instructions

Name	Mnemonic	Syntax
Vector Round to Floating-Point Integer Nearest	**vrfin**	vD,vB
Vector Round to Floating-Point Integer Toward Zero	**vrfiz**	vD,vB
Vector Round to Floating-Point Integer Toward Positive Infinity	**vrfip**	vD,vB
Vector Round to Floating-Point Integer Toward Minus Infinity	**vrfim**	vD,vB
Vector Convert from Unsigned Fixed-Point Word	**vcfux**	vD,vB, **UIMM**
Vector Convert from Signed Fixed-Point Word	**vcfsx**	vD,vB, **UIMM**
Vector Convert to Unsigned Fixed-Point Word Saturate	**vctuxs**	vD,vB, **UIMM**
Vector Convert to Signed Fixed-Point Word Saturate	**vctsxs**	vD,vB, **UIMM**

3.4.5.5.6 AltiVec floating-point compare instructions

All AltiVec floating-point compare instructions (**vcmpeqfp**, **vcmpgtfp**, **vcmpgefp**, and **vcmpbfp**) return FALSE if either operand is a NaN. Not equal-to, not greater-than, not greater-than-or-equal-to, and not-in-bounds NaNs compare to everything, including themselves.

The AltiVec floating-point compare instructions compare the elements in two vector registers word-by-word, interpreting the elements as single-precision numbers. With the exception of the Vector Compare Bounds Floating-Point (**vcmpbfp**) instruction, they set the target vector register, and CR[6] if Rc = 1, in the same manner as do the vector integer compare instructions.

The Vector Compare Bounds Floating-Point (**vcmpbfp**) instruction sets the target vector register, and CR[6] if Rc = 1, to indicate whether the elements in **v**A are within the bounds specified by the corresponding element in **v**B, as explained in the instruction description. A single-precision value x is said to be within the bounds specified by a single-precision value y if (–y ≤ x ≤ y).

This table lists the AltiVec floating-point compare instructions.

Table 3-38. AltiVec floating-point compare instructions

Name	Mnemonic	Syntax
Vector Compare Greater Than Floating-Point [Record]	**vcmpgtfp**[.]	vD,vA,vB
Vector Compare Equal to Floating-Point [Record]	**vcmpeqfp**[.]	vD,vA,vB
Vector Compare Greater Than or Equal to Floating-Point [Record]	**vcmpgefp**[.]	vD,vA,vB
Vector Compare Bounds Floating-Point [Record]	**vcmpbfp**[.]	vD,vA,vB

3.4.5.5.7 AltiVec floating-point estimate instructions

This table lists the AltiVec floating-point estimate instructions.

Table 3-39. AltiVec floating-point estimate instructions

Name	Mnemonic	Syntax
Vector Reciprocal Estimate Floating-Point	**vrefp**	**v**D,**v**B
Vector Reciprocal Square Root Estimate Floating-Point	**vrsqrtefp**	**v**D,**v**B
Vector Log2 Estimate Floating-Point	**vlogefp**	**v**D,**v**B
Vector 2 Raised to the Exponent Estimate Floating-Point	**vexptefp**	**v**D,**v**B

3.4.5.6 AltiVec compatibility instructions

The data stream control instructions present in some implementations of earlier PowerPC processors to assist in reducing latency by stream prefetching are provided as a means to correctly run AltiVec code written for such processors. On the e6500 core, these instructions execute as no-ops.

This table lists the AltiVec compatibility instructions.

Table 3-40. AltiVec compatibility instructions

Name	Mnemonic	Syntax
Data Stream Stop[1]	**dss**	STRM
Data Stream Stop All[1]	**dssall**	
Data Stream Touch[1]	**dst**	STRM,**v**D,**v**A,**v**B
Data Stream Touch for Store[1]	**dstst**	STRM,**v**D,**v**A,**v**B
Data Stream Touch for Store Transient[1][1]	**dststt**	STRM,**v**D,**v**A,**v**B
Data Stream Touch Transient	**dstt**	STRM,**v**D,**v**A,**v**B

[1] The data stream control instructions are provided to permit AltiVec code from earlier PowerPC processors to correctly execute. These instructions perform no action and are treated as no-ops.

3.4.5.7 AltiVec permutation and formatting instructions

Vector pack, unpack, merge, splat, permute, and select can be used to accelerate various vector math and vector formatting. Details of the various instructions are provided in the following sections.

3.4.5.7.1 AltiVec pack instructions

Halfword vector pack instructions (**vpkuhum**, **vpkuhus**, **vpkshus**, **vpkshss**) truncate the 16 halfwords from two concatenated source operands producing a single result of 16 bytes (quad word) using either modulo (2^8), 8-bit signed-saturation, or 8-bit unsigned-saturation to perform the truncation. Similarly, word vector pack instructions (**vpkuwum**, **vpkuwus**, **vpkswus**, and **vpksws**) truncate the eight words from two concatenated source operands producing a single result of eight halfwords using modulo (2^16), 16-bit signed-saturation, or 16-bit unsigned-saturation to perform the truncation.

One special form of Vector Pack Pixel (**vpkpx**) instruction packs eight 32-bit (8/8/8/8) pixels from two concatenated source operands into a single result of eight 16-bit (1/5/5/5) αRGB pixels. The least significant bit of the first 8-bit element becomes the 1-bit α field, and each of the three 8-bit R, G, and B fields are reduced to 5 bits by ignoring the 3 least significant bits.

This table describes the AltiVec pack instructions.

Table 3-41. AltiVec pack instructions

Name	Mnemonic	Syntax
Vector Pack Unsigned Integer [**h**,**w**] Unsigned Modulo	**vpkuhum** **vpkuwum**	vD,vA,vB
Vector Pack Unsigned Integer [**h**,**w**] Unsigned Saturate	**vpkuhus** **vpkuwus**	vD,vA,vB
Vector Pack Signed Integer [**h**,**w**] Unsigned Saturate	**vpkshus** **vpkswus**	vD,vA,vB
Vector Pack Signed Integer [**h**,**w**] Signed Saturate	**vpkshss** **vpkswss**	vD,vA,vB
Vector Pack Pixel	**vpkpx**	vD,vA,vB

3.4.5.7.2 AltiVec unpack instructions

Byte vector unpack instructions unpack the eight low bytes (or eight high bytes) of one source operand into eight halfwords using sign extension to fill the most significant bits. Halfword vector unpack instructions unpack the four low halfwords (or four high halfwords) of one source operand into four words using sign extension to fill the most significant bits.

A special-purpose form of vector unpack is provided, the Vector Unpack Low Pixel (**vupklpx**) and the Vector Unpack High Pixel (**vupkhpx**) instructions for 1/5/5/5 αRGB pixels. The 1/5/5/5 pixel vector unpacks the four low 1/5/5/5 pixels (or four 1/5/5/5 high pixels) into four 32-bit (8/8/8/8) pixels. The 1-bit α element in each pixel is sign extended to 8 bits, and the 5-bit R, G, and B elements are each zero extended to 8 bits.

This table describes the AltiVec unpack instructions.

Table 3-42. AltiVec unpack instructions

Name	Mnemonic	Syntax
Vector Unpack High Signed Integer [b,h]	**vupkhsb** **vupkhsh**	vD,vB
Vector Unpack High Pixel	**vupkhpx**	vD,vB
Vector Unpack Low Signed Integer [**b**,**h**]	**vupklsb** **vupklsh**	vD,vB
Vector Unpack Low Pixel	**vupklpx**	vD,vB

3.4.5.7.3 AltiVec merge instructions

Byte vector merge instructions interleave the eight low bytes (or eight high bytes) from two source operands producing a result of 16 bytes. Similarly, halfword vector merge instructions interleave the four low halfwords (or four high halfwords) of two source operands to produce a result of eight halfwords. Word vector merge instructions interleave the two low words (or two high words) from two source operands producing a result of four words. The vector merge instruction has many uses; most notably among them is a way to efficiently transpose SIMD vectors.

This table describes the merge instructions.

Table 3-43. AltiVec merge instructions

Name	Mnemonic	Syntax
Vector Merge High Integer [b,h,w]	**vmrghb** **vmrghh** **vmrghw**	v**D**,v**A**,v**B**
Vector Merge Low Integer [b,h,w]	**vmrglb** **vmrglh** **vmrglw**	v**D**,v**A**,v**B**

3.4.5.7.4 AltiVec splat instructions

This table describes the AltiVec splat instructions.

Table 3-44. AltiVec splat instructions

Name	Mnemonic	Syntax
Vector Splat Integer [b,h,w]	**vspltb** **vsplth** **vspltw**	v**D**,v**B**,**UIMM**
Vector Splat Immediate Signed Integer [b,h,w]	**vspltisb** **vspltish** **vspltisw**	v**D**,**SIMM**

3.4.5.7.5 AltiVec Permute instruction

The Permute instruction allows any byte in any two source vector registers to be directed to any byte in the destination vector. The fields in a third source operand specify from which field in the source operands the corresponding destination field is taken.

This table describes the vector permute instruction.

Table 3-45. AltiVec Permute instruction

Name	Mnemonic	Syntax
Vector Permute	**vperm**	v**D**,v**A**,v**B**,v**C**

3.4.5.7.6 AltiVec Select instruction

The **vsel** instruction selects one field from one or the other of two source operands under control of its mask operand. Use of the TRUE/FALSE compare result vector with select in this manner produces a two-instruction equivalent of conditional execution on a per-field basis.

This table describes the **vsel** instruction.

Table 3-46. AltiVec Select instruction

Name	Mnemonic	Syntax
Vector Select	**vsel**	**v**D,**v**A,**v**B,**v**C

3.4.5.7.7 AltiVec shift instructions

The AltiVec shift instructions shift the contents of a vector register or of a pair of vector registers left or right by a specified number of bytes (**vslo**, **vsro**, **vsldoi**) or bits (**vsl**, **vsr**). Depending on the instruction, this shift count is specified either by low-order bits of a vector register or by an immediate field in the instruction. In the former case, the low-order seven bits of the shift count register give the shift count in bits ($0 \le \text{count} \le 127$).

This table describes the AltiVec shift instructions.

Table 3-47. AltiVec shift instructions

Name	Mnemonic	Syntax
Vector Shift Left	**vsl**	**v**D,**v**A,**v**B
Vector Shift Right	**vsr**	**v**D,**v**A,**v**B
Vector Shift Left Double by Octet Immediate	**vsldoi**	**v**D,**v**A,**v**B,SH
Vector Shift Left by Octet	**vslo**	**v**D,**v**A,**v**B
Vector Shift Right by Octet	**vsro**	**v**D,**v**A,**v**B

3.4.5.7.8 AltiVec status and control register instructions

This table summarizes the instructions for reading from or writing to the Vector Status and Control (VSCR) register.

Table 3-48. Move to/from the AltiVec status and control register instructions

Name	Mnemonic	Syntax
Move to Vector Status and Control Register	**mtvscr**	**v**B
Move from Vector Status and Control Register	**mfvscr**	**v**D

3.4.5.7.9 GPR to AltiVec move instructions

This table summarizes the instructions for moving data from GPRs to a vector register.

Table 3-49. Move to vector register from GPR instructions

Name	Mnemonic	Syntax
Move to Vector from Integer Double Word and Splat	**mvidsplt**	**v**D,**r**A,**r**B
Move to Vector from Integer Word and Splat	**mviwsplt**	**v**D,**r**A,**r**B

3.4.6 Branch and flow control instructions

Some branch instructions can redirect instruction execution conditionally based on the bit values in CR. Information about branch instruction address calculation is provided in *EREF*.

3.4.6.1 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 3-51 as having the value *t*, is used by some implementations for branch prediction; this is not used on the e6500 core.

NOTE

The e6500 core 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 execution unit."

This table provides the BO bit descriptions.

Table 3-50. 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 e6500 core does not use static branch prediction and ignores this bit.

This table provides the BO operand encodings.

Table 3-51. 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.

Table 3-51. BO operand encodings (continued)

BO	Description
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 provide a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance. e6500 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.

3.4.6.2 Branch instructions

The following table lists branch instructions. Appendix A, “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 e6500 core does not use the BO operand for static branch prediction.

Table 3-52. 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

3.4.6.3 Integer Select (isel) instruction

Integer Select (**isel**), shown in the following table, is a conditional register move instruction that helps eliminate branches. Programming guidelines for **isel** are given in *EREF*.

Table 3-53. Integer Select instruction

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

3.4.6.4 Condition register logical instructions

The following table shows the condition register logical instructions. Both of these instructions and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Table 3-54. 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.

3.4.6.5 Trap instructions

Trap instructions, shown in the following table, 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 A, "Simplified Mnemonics," for more information.

Table 3-55. Trap instructions

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

3.4.6.6 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. For additional details, see Section 3.4.12.1, "System linkage and MSR access instructions."

This table lists the system linkage instruction.

Table 3-56. 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.

3.4.6.7 Hypervisor privilege instruction

The hypervisor facility defines the Generate Embedded Hypervisor Privilege Exception instruction (**ehpriv**), which generates a hypervisor privilege exception. See Section 4.9.21, "Hypervisor privilege interrupt—IVOR41." **ehpriv** is fully described in *EREF*. Note that the OC field is not interpreted by hardware but is for the use of the hypervisor to provide specific emulation.

This table shows the hypervisor privilege instruction.

Table 3-57. Hypervisor privilege instruction

Name	Mnemonic	Syntax
Hypervisor Privilege	**ehpriv**	OC

3.4.7 Processor control instructions

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

3.4.7.1 Move to/from Condition Register instructions

This table summarizes the instructions for reading from or writing to the CR.

Table 3-58. 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 e6500 core.
Move to Condition Register from XER	**mcrxr**	**cr**D	—
Move from Condition Register	**mfcr**	**r**D	—

Table 3-58. Move to/from Condition Register instructions (continued)

Name	Mnemonic	Syntax	Implementation Note
Move from One Condition Register Field	**mfocrf**	**r**D,FXM	See *EREF* for a full description of this instruction.
Move to One Condition Register Field	**mtocrf**	FXM,**r**S	See *EREF* for a full description of this instruction.

This table lists the **mtspr** and **mfspr** instructions.

Table 3-59. 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.

3.4.7.2 Wait for Interrupt instruction

wait stops synchronous thread activity until an asynchronous interrupt or a debug instruction complete exception occurs (or, optionally, when the thread's reservation is not valid). In a core, the wait condition of all threads also terminates when a cache stash is received by the core.

On the e6500 core, **wait** also causes power consumption to be reduced when the processor is waiting. Power reduction is stepped over time; although, specifying WH = 1 causes immediate power reduction. Specifying WH = 1 should only be used if it is known that the wait will be a longer period of time.

Power reduction states caused by the **wait** instruction are further described in Section 8.3, "Core power management states." **wait** also causes any prefetched instructions to be discarded, and thread instruction fetching ceases until the wait condition terminates.

Table 3-60. Wait for interrupt instruction

Name	Mnemonic	Syntax
Wait for Interrupt	**wait**	WC,WH

3.4.8 Performance monitor instructions (user level)

The performance monitor provides read-only, application-level access to some performance monitor resources. This table lists the performance monitor instructions.

Table 3-61. Performance monitor instructions

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

The user-level PMRs listed in Table 2-16 are accessed with **mfpmr**. Attempting to write user-level PMRs in either mode causes an illegal instruction exception.

3.4.9 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 *EREF* for additional information about these instructions and about related aspects of memory synchronization. See Table 3-62 for a summary.

This table describes the memory synchronization instructions.

Table 3-62. Memory synchronization Instructions

Name	Mnemonic	Syntax	Implementation Notes
Instruction Synchronize	**isync**	—	**isync** is refetch serializing. The e6500 core waits for previous instructions (including interrupts they generate) to complete before **isync** executes. This purges all instructions from the thread 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.
Load (Byte, Halfword, Word, Doubleword) and Reserve Indexed	**lbarx** **lharx** **lwarx** **ldarx**	**r**D,**r**A,**r**B	Load and reserve instructions (**lbarx**, **lharx**, **lwarx**, **ldarx**) when paired with store conditional instructions (**stbcx.**, **sthcx.**, **stwcx.**, **stdcx.**) 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, the e6500 core 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 e6500 core makes reservations on behalf of aligned 64-byte sections of address space (coherence granule). While the e6500 core 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 the e6500 core 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 load and reserve or store conditional instructions 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 *EREF*.

Table 3-62. Memory synchronization Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Memory Barrier	**mbar**	MO	**mbar** provides a memory barrier. The behavior of **mbar** depends on the value of the 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** instruction provides a storage ordering function for all memory access instructions executed by the processor executing **mbar**. Executing **mbar** ensures that all data storage accesses caused by instructions preceding the **mbar** have completed before any data storage accesses caused by any instructions after the **mbar**. This order is seen by all mechanisms. MO = 1—**mbar** functions identically to **eieio**. For more information, see Section 3.4.9.1, “mbar (MO = 1).”

Table 3-62. Memory synchronization Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Memory Synchronize	**sync** (**msync**)	L,E	**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 several types of memory barriers specified by the L and E fields: • L = 0 (“heavyweight sync”). The memory barrier is throughout the memory hierarchy. In the e6500 core, **sync 0,0** waits for proceeding data memory accesses to become visible to the entire memory hierarchy; then, it is broadcast on the CoreNet interface. **sync 0,0** completes only after its address tenure. Subsequent instructions can execute out of order but complete only after the **sync 0,0** completes. The simplified mnemonics **hwsync**, **sync**, and **msync** are equivalent to **sync 0,0**. • L = 1 (“lightweight sync”). The memory barrier provides an ordering function for the storage accesses caused by load, store, and **dcbz** type instructions executed by the processor executing the **sync** instruction and for which the specified storage locations are 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** type instruction and b_j is an access caused by a load instruction. The **sync 1,0** 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 **sync1,0**. • E (elemental sync). When the E operand is specified and is not 0, the bits of the E field (E[0] - E[3]), if set to 1, provide a memory barrier for the storage accesses caused by load, store, and **dcbz** type instructions executed by the processor executing the **sync** instruction and for which the specified storage locations are neither write through required nor caching inhibited. The applicable pairs are all pairs a_i,b_j of such accesses which are defined by how bits in the E field are set as follows: E[0]—load with load. a_i is an access caused by a load instruction, and b_j is an access caused by a load instruction. E[1]—load with store. a_i is an access caused by a load instruction, and b_j is an access caused by a store or **dcbz** type instruction. E[2]—store with load. a_i is an access caused by a store or **dcbz** type instruction, and b_j is an access caused by a load instruction. E[3]—store with store. a_i is an access caused by a store or **dcbz** type instruction, and b_j is an access caused by a store or **dcbz** type instruction. All four bits of the E operand can be specified simultaneously. For example, E = 0b1101 is equivalent to **lwsync**. The simplified mnemonic for elemental sync is **esync** E. Omitting the E operand for **sync** assumes a value of 0 for E. Memory accesses performed by a hardware page table translation are treated as loads with respect to the **sync 0,0** (**hwsync**) memory barrier. In particular, **hwsync** provides an ordering function for all preceding stores to a page table caused by store instructions and the implicit loads that may occur during a page table translation after the **hwsync** instruction completes. Executing a **hwsync** instruction ensures that all such stores are performed with respect to the thread executing the **hwsync** instruction, before any implicit accesses to the affected PTEs (targets of pervious stores) due to a page table translation are performed with respect to that thread. **sync** latency depends on the processor state when it is dispatched and on various system-level conditions. Frequent use of **sync 0,0** degrades performance and **esync** should be used where possible. In multiprocessing code that performs locking operations to lock shared data structures: • **sync**—ensures 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. **esync 0b0001** is preferable in many cases. • Unlike a context-synchronizing operation, **sync** does not discard prefetched instructions.

Table 3-62. Memory synchronization Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Store (Byte, Halfword, Word, Doubleword) Conditional Indexed	**stbcx.** **sthcx.** **stwcx.** **stdcx.**	rS,rA,rB	See **lbarx**, **lharx**, **lwarx**, **ldarx** (listed in this table) for a description of how load and reserve and store conditional instructions are used in pairs. For **stbcx.**, **sthcx.**, **stwcx.**, **stdcx.**, the e6500 core 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.

The section, “Lock Acquisition and Import Barriers,” in *EREF* describes how the **sync** and **mbar** instructions can be used to control memory access ordering when memory is shared between programs.

3.4.9.1 mbar (MO = 1)

As defined by the architecture, **mbar** (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. The two sets follow:

- Caching-inhibited, guarded loads and stores to memory, and write-through-required stores to memory. **mbar** (MO = 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** have completed with respect to main memory before any such accesses caused by instructions following **mbar** 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** type instructions 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** (MO = 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** complete before any applicable stores caused by instructions after it.

Memory accesses caused by **dcbz** or **dcba** type instructions are ordered like a store.

Except for **dcbz** and **dcba** type instructions, **mbar** (MO = 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** does not affect the order of accesses in one set with respect to accesses in the other.

mbar (MO = 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** (MO = 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** (MO = 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.

Because the threads perform store operations in order to memory that is designated as both caching-inhibited and guarded, **mbar** (MO = 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 *EREF* describes how **sync** and **mbar** control memory access ordering when programs share memory.

3.4.10 Reservations

The ability to emulate an atomic operation using load with reservation and store conditional instructions is based on the conditional behavior of **stbcx.**, **sthcx.**, **stwcx.**, **stdcx.**, the reservation set by **lbarx**, **lharx**, **lwarx**, **ldarx**, 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 *EREF*. On the e6500 core, a reservation by a thread will be lost for any of the following reasons:

- The thread executes a store condition instruction.
- 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). If the modification is done using a decorated load, decorated store, or decorated notify instruction, it is undefined whether the reservation is lost.
- 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).
- Some other processor executes.

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.

Software should not perform decorated storage operations to the same reservation granule that is a target of load and reserve instructions, as doing so does not guarantee that reservations are cleared appropriately.

3.4.11 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 3.4.12.4, “Supervisor-level memory control instructions,” for information about supervisor-level cache and translation lookaside buffer management instructions. Cache-locking instructions are described in Section 3.4.11.2, “Cache locking instructions.”

3.4.11.1 User-level cache instructions

The instructions listed in Table 3-63 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 e6500 core’s caches.

3.4.11.1.1 CT field values

The e6500 core 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.

  Only CT = 0 or CT = 2 may be used with a **dcblq.** or an **icblq.** instruction. All other CT values used with these instructions set CR0 to 0b000 || XER[SO].
- 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 3.4.11.2, “Cache locking instructions,” describes cache-locking instructions.

This table describes the user-level cache instructions.

Table 3-63. User-level cache instructions

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Allocate	**dcba**	rA,rB	If L1CSR0[DCBZ32] = 0, **dcba** operates on all bytes in the cache line (cache-line operation). If L1CSR0[DCBZ32] = 1, **dcba** operates on 32 bytes (32-byte operation). The **dcba** instruction 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 no-op 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 no-op, executing the **dcba** can result in IAC debug events, but does not cause DAC debug events.
Data Cache Block Allocate by Line	**dcbal**	rA,rB	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**	rA,rB	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. For the e6500 core, if **dcbf** is performed to memory that is not caching inhibited, memory coherent, and not write through required (WIMGE = 0b001xx), it is treated as a store with respect to memory barriers established by **lwsync** and **esync**.
Data Cache Block Set to Zero	**dcbz**	rA,rB	If L1CSR0[DCBZ32] = 0, **dcbz** 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** will take an alignment exception if any of the following occur: • The page is marked write-through. • The page is marked caching-inhibited.
Data Cache Block Set to Zero by Line	**dcbzl**	rA,rB	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].

Table 3-63. User-level cache instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Store	**dcbst**	**r**A,**r**B	**dcbst** is implemented identically to **dcbf**.
Data Cache Block Touch	**dcbt**	TH,**r**A,**r**B [1]	When **dcbt** executes, the e6500 core checks for protection violations (as for a load instruction). **dcbt** is treated as a no-op in the following cases on the e6500 core: • The access causes 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 no-op, 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 e6500 core checks for protection violations (as for a load instruction). **icbt** is treated as a no-op in the following cases on the e6500 core: • The access causes 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 no-op, executing the **icbt** can result in IAC debug events, but does not cause DAC debug events. **Note:** The primary instruction cache (CT=0) on the e6500 core does not perform **icbt** instructions and they are treated as a no-op.
Make it So	**miso**	—	**miso** is a hint to the processor that performance will be improved if all stores previously executed are performed as soon as possible. On the e6500 core, this causes all the store gather buffers to be sent to the L2 cache, which is the point of coherency. **miso** can improve multiprocessor performance if other processors are waiting to see a store performed.

[1] TH was formerly defined as CT.

3.4.11.2 Cache locking instructions

Table 3-64 describes the implementation of the cache locking instructions, which are fully described in *EREF*.

The **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, **icblc**, **dcblq.**, and **icblq.** cache-locking instructions require hypervisor state privilege to execute when MSRP[UCLEP] = 1. Execution of these instructions in the guest-supervisor state when MSRP[UCLEP] = 1 causes a hypervisor privilege exception. User mode execution of these instructions is unaffected and is controlled by MSR[UCLE].

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

Unless otherwise stated, the behavior applies to all instructions.

Table 3-64. 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.
Data Cache Block Lock Query	**dcblq.**	CT,rA,rB	If the line is locked in the specified cache, CR0[EQ] is set to 1; otherwise, it is set to 0. **dcblq.** can only be used with CT = 0 and CT = 2, other values of CT cause the instruction to set CR0 to b000 || XER[SO].
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.
Instruction Cache Block Lock Query	**icblq.**	CT,rA,rB	If the line is locked in the specified cache, CR0[EQ] is set to 1; otherwise, it is set to 0. **icblq.** can only be used with CT = 0 and CT = 2, other values of CT cause the instruction to set CR0 to b000 || XER[SO].

Full descriptions of these instructions are in the “Instruction Set” chapter of *EREF*. Note the following behaviors for the e6500 core:

- 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. An unable-to-lock condition occurs when:
  - — The targeted cache is not enabled.
  - — The target address is marked Caching Inhibited (WIMGE = 0bx1xx).
  - — 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.
  - — If an overlocking condition occurs in the L2 cache (CT=2), the line is not loaded or locked.
- Setting L1CSR0[CLFC] flash invalidates all primary data cache lock bits and setting L1CSR1[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 locked lines.

  Because L1 cache locking is persistent, setting L1CSR0[CFI] or L1CSR1[ICFI] does not clear the locks in the respective caches when 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.

Cache locking clear instructions (**dcblc** and **icblc**) are no-oped 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
dcblq.
(check CR0[EQ] to determine if the line is locked)
```

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

```
icbtls
icblq.
(check CR0[EQ] to determine if the line is locked)
```

- "CR0 = 0" means the instruction is completed without accessing the cache and the CR0 field is set to 0.

3.4.12 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 e6500. Instructions described here have an associated privilege and actions, as described in the following table.

Table 3-65. 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

3.4.12.1 System linkage and MSR access instructions

Table 3-66 describes system linkage instructions as they are implemented on the e6500 core. The user-level **sc** (LEV = 0) instruction lets a user program call on the system to perform a service and causes the thread 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 thread 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 thread maps the instruction to **rfgi**, ensuring 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 will emulate the taking of these interrupts in the 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.

This table lists the supervisor-level system linkage instructions.

Table 3-66. System linkage instructions—supervisor-level

Name	Mnemonic	Syntax	Implementation Notes
Return from Interrupt	**rfi**	—	These instructions are context-synchronizing, which, for the e6500 core, means they work their way to the final execute stage, update architected registers, and redirect instruction flow. In the 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** because 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 3-67. 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 the 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 3-59) 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 *EREF* for more information on context synchronization requirements when altering certain SPRs.

3.4.12.2 Thread management instructions

The thread management instructions provide read-write access to thread management resources that allow the e6500 threads to be controlled. Thread management instructions are hypervisor privileged.

This table lists the thread management instructions.

Table 3-68. Thread management instructions

Name	Mnemonic	Syntax
Move from Thread Management Register	**mftmr**	**r**D,TMRN
Move to Thread Management Register	**mttmr**	TMRN,**r**S

The TMRs listed in Section 2.15.2, "Thread management registers (TMRs)" are accessed with **mftmr** and **mttmr** instructions.

INIA*n* and IMSR*n* TMRs can only be written with **mttmr** when its associated thread '*n*' is disabled. Writes to INIA or IMSR of an enabled thread are ignored.

Executing **mftmr** or **mttmr** in the guest-supervisor state to a valid TMR causes an embedded hypervisor privilege interrupt.

3.4.12.3 External PID load and 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 manipulate a process' virtual memory using the context and credentials of the process.

The external PID instructions are implemented as described in *EREF*. Any implementation-specific behaviors for external PID instructions is the same as the non-external PID analogous instruction for the e6500 core (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 3-69. External PID load and 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 Vector by External PID Indexed	**lvepx**	**v**D,**r**A,**r**B	**lvx**
Load Vector by External PID Indexed LRU	**lvepxl**	**v**D,**r**A,**r**B	**lvxl**
Load Word by External PID Indexed	**lwepx**	**r**D,**r**A,**r**B	**lwzx**

Table 3-69. External PID load and store instructions (continued)

Instruction	Mnemonic	Syntax	Non External PID Analogous Instruction
Load Doubleword by External PID Indexed	**ldepx**	rD,rA,rB	**ldx**
Store Byte by External PID Indexed	**stbepx**	rS,rA,rB	**stbx**
Store Floating-Point Doubleword by External PID Indexed	**stfdepx**	frS,rA,rB	**stfdx**
Store Halfword by External PID Indexed	**sthepx**	rS,rA,rB	**sthx**
Store Vector by External PID Indexed	**stvepx**	vS,rA,rB	**stvx**
Store Vector by External PID Indexed LRU	**stvepxl**	vS,rA,rB	**stvxl**
Store Word by External PID Indexed	**stwepx**	rS,rA,rB	**stwx**
Store Doubleword by External PID Indexed	**stdepx**	rS,rA,rB	**stdx**
Data Cache Block Flush by External PID Indexed	**dcbfep**	rA,rB	**dcbf**
Data Cache Block Store by External PID Indexed	**dcbstep**	rA,rB	**dcbst**
Data Cache Block Touch by External PID Indexed	**dcbtep**	TH,rA,rB	**dcbt**
Data Cache Block Touch for Store by External PID Indexed	**dcbtstep**	TH,rA,rB	**dcbtst**
Data Cache Block Zero by External PID Indexed	**dcbzep**	rA,rB	**dcbz**
Data Cache Block Zero Long by External PID Indexed	**dcbzlep**	rA,rB	**dcbzl**
Instruction Cache Block Invalidate by External PID Indexed	**icbiep**	rA,rB	**icbi**

3.4.12.4 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 3.4.11, "Memory control instructions," describes user-level memory control instructions.

3.4.12.4.1 Supervisor-level cache instruction

The following table lists the supervisor-level cache management instructions except for cache management instructions, which are part of the External PID instructions.

Table 3-70. Supervisor-level cache management instruction

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Invalidate	**dcbi**	rA,rB	**dcbi** executes as defined in *EREF* but has implementation-dependent behaviors. When the address to be invalidated is marked Memory Coherence Required (WIMGE = 0bx01xx), a **dcbf** is performed that 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 e6500 core, **dcbi** cannot generate a cache-locking exception**.** For the e6500 core, if **dcbi** is performed to memory that is not caching inhibited, memory coherent, and not write through required (WIMGE = 0b001xx), it is treated as a store with respect to memory barriers established by **lwsync** and **esync**.

dcbi is guest supervisor privileged.

See Section 3.4.11.1, “User-level cache instructions,” for cache instructions that provide user-level programs the ability to manage on-chip caches.

3.4.12.4.2 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*. The e6500 core implements MMU architecture version 2.

This table summarizes the operation of the TLB instructions in the e6500 core.

Table 3-71. 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 that are protected by the IPROT attribute (entry$_{\text{IPROT}}$=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**). **Note:** **tlbilx** will invalidate the TLB entries of all threads on the e6500 core but will not be synchronized with respect to the data accesses and instruction fetches in the threads on which **tlbilx** is not executed. If the invalidation must be synchronized in the other threads, then software must arrange to execute **tlbilx** on all threads.
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 or LRAT 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 ATSEL, TLBSEL, ESEL, and EPN fields of MAS0 and MAS2. The contents extracted from the MMU are placed in MAS0–MAS3, MAS7, and MAS8. 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, MAS0[V] is set and the found entry is read into MAS0–MAS3, MAS7, and MAS8. If the entry is not found, MAS0[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 e6500 core, take an illegal instruction exception if **r**A != 0.

Table 3-71. TLB management instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
TLB Synchronize	**tlbsync**	—	Causes a TLBSYNC transaction on the 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 the e6500 core 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 or LRAT entry in the MMU. If a TLB entry is written by the guest supervisor and a matching LRAT entry is found, the RPN fields of MAS3 and MAS7 are translated through the LRAT and the resulting translation is written into the RPN field of the TLB entry. The entry written is specified by the ATSEL, TLBSEL, ESEL (ESEL is not used if HES = 1 and TLBSEL = 0, and hardware selects which way to write) and EPN fields of MAS0 and MAS2. See Section 6.3, "Translation lookaside buffers (TLBs)." **tlbwe** is hypervisor privileged if EPCR[DGTMI] = 1. TLB entries may be written to TLB0 by the guest supervisor if a matching LRAT entry is found.

Implementation notes:

- If an attempt is made to write a TLB1 entry and MAS1[TSIZE] specifies an invalid size (that is, 0), the TSIZE for the entry is set to 2 (4 KB).
- If an attempt is made to write an LRAT entry and MAS1[TSIZE] specifies an invalid size (that is, 0, 1, or 31), the LSIZE for the entry is set to 2 (4 KB).
- Reads (**tlbre**) from TLB0 ignore the value of MAS0[HES] and always perform reads as if MAS0[HES] = 0.
- Reads (**tlbre** or **tlbsx**) from TLB1 that read indirect, valid entries (TLB[V] = 1 and TLB[IND] = 1) should always return MAS3[SPSIZE] = 0b00010 and write TLB[UND] = TLB[SR].
- Writes (**tlbwe**) to TLB1 that write indirect, valid entries (TLB[V] = 1 and TLB[IND] = 1) always write TLB[SR] = MAS3[UND].
- Writes (**tlbwe**) that write TLB entries when executing in 32-bit mode (MSR[CM] = 0) write 0 to the upper 32 bits of the EPN field of the TLB entry.
- TLB0 does not store the IND attribute and it is ignored on writes to TLB0.
- 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 sub-routines.
- Executing **tlbsx** with **r**A != 0 causes an illegal instruction exception on the e6500 core. Software should always use **tlbsx** with **r**A = 0.

3.4.12.5 Message Clear and Message Send instructions

The e6500 core can generate messages to other processors and devices in the system. Messages are generated by using the Message Send (**msgsnd**) instruction. When a thread 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 e6500 core accepts the following message types, which generate corresponding interrupts:

- Processor doorbell
- Processor doorbell critical
- Guest processor doorbell
- Guest processor doorbell critical
- Guest processor doorbell machine check

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** instructions are implemented as described in the architecture and in *EREF*.

See Section 4.9.20.1, "Doorbell interrupt definitions," for more information about doorbell interrupt types.

This table lists the Message Clear and Message Send instructions.

Table 3-72. 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.

3.4.12.6 Performance monitor instructions (supervisor level)

Software communicates with the performance monitor through performance monitor registers (PMRs) with the instructions listed in the following table.

Table 3-73. 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. 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 2.16, "Performance monitor registers (PMRs)."

3.4.13 Recommended simplified mnemonics

The description of each instruction includes the mnemonic and a formatted list of operands. Compliant assemblers support the mnemonics and operands listed. Simplified mnemonics and symbols are provided for frequently used instructions. See Appendix A, "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.

3.4.14 Context synchronization

Context synchronization is achieved by post- and pre-synchronizing instructions. An instruction is pre-synchronized by completing all instructions before dispatching the pre-synchronized 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 *EREF*. **dnh** can be used to halt the thread when an external debugger is attached and has enabled halting by setting EDBCR0[DNH_EN]. When the thread is halted, the DUI field is passed directly to the debugger as information describing the reason for the halt. The DCTL has uses defined for triggering functions in the integrated device. See Section 9.9.16.2, “Debugger Notify Halt (dnh) instruction.” If an external debugger is not attached or has not enabled halting, **dnh** takes an illegal instruction exception.

The **dni** instruction can be used to signal a debug interrupt when MSR[DE] = 1.

This table lists the **dnh** debug instructions.

Table 3-74. dnh debug instruction

Name	Mnemonic	Syntax	Implementation Details
Debugger Notify Halt	**dnh**	DUI,DCTL	—
Debugger Notify Interrupt	**dni**	DUI,DCTL	—

3.6 Instruction listing

This table lists the instructions implemented on the e6500 core.

Table 3-75. e6500 core instruction set

Mnemonic	Syntax	Classification	Cross-Reference
add	rD,rA,rB	Integer	Table 3-5
add.	rD,rA,rB	Integer	Table 3-5
addc	rD,rA,rB	Integer	Table 3-5
addc.	rD,rA,rB	Integer	Table 3-5
addco	rD,rA,rB	Integer	Table 3-5
addco.	rD,rA,rB	Integer	Table 3-5
adde	rD,rA,rB	Integer	Table 3-5
adde.	rD,rA,rB	Integer	Table 3-5
addeo	rD,rA,rB	Integer	Table 3-5

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
addeo.	rD,rA,rB	Integer	Table 3-5
addi	rD,rA,SIMM	Integer	Table 3-5
addic	rD,rA,SIMM	Integer	Table 3-5
addic.	rD,rA,SIMM	Integer	Table 3-5
addis	rD,rA,SIMM	Integer	Table 3-5
addme	rD,rA	Integer	Table 3-5
addme.	rD,rA	Integer	Table 3-5
addmeo	rD,rA	Integer	Table 3-5
addmeo.	rD,rA	Integer	Table 3-5
addo	rD,rA,rB	Integer	Table 3-5
addo.	rD,rA,rB	Integer	Table 3-5
addze	rD,rA	Integer	Table 3-5
addze.	rD,rA	Integer	Table 3-5
addzeo	rD,rA	Integer	Table 3-5
addzeo.	rD,rA	Integer	Table 3-5
and	rA,rS,rB	Integer logical	Table 3-7
and.	rA,rS,rB	Integer logical	Table 3-7
andc	rA,rS,rB	Integer logical	Table 3-7
andc.	rA,rS,rB	Integer logical	Table 3-7
andi.	rA,rS,UIMM	Integer logical	Table 3-7
andis.	rA,rS,UIMM	Integer logical	Table 3-7
b	LI	Branch	Table 3-52
ba	LI	Branch	Table 3-52
bc	BO,BI,BD	Branch	Table 3-52
bca	BO,BI,BD	Branch	Table 3-52
bcctr	BO,BI	Branch	Table 3-52
bcctrl	BO,BI	Branch	Table 3-52
bcl	BO,BI,BD	Branch	Table 3-52
bcla	BO,BI,BD	Branch	Table 3-52
bclr	BO,BI	Branch	Table 3-52
bclrl	BO,BI	Branch	Table 3-52
bl	LI	Branch	Table 3-52
bla	LI	Branch	Table 3-52

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
bpermd	**r**A,**r**S,**r**B	Integer logical	Table 3-7
cmp	**crf**D,L,**r**A,**r**B	Compare	Table 3-6
cmpb	**r**A,**r**S,**r**B	Integer logical	Table 3-7
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
cntlzd	**r**A,**r**S	Integer logical	Table 3-7
cntlzd.	**r**A,**r**S	Integer logical	Table 3-7
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 3-63
dcbal	**r**A,**r**B	Extended cache line/cache control	Table 3-63
dcbf	**r**A,**r**B	Cache control	Table 3-63
dcbfep	**r**A,**r**B	External PID load/store	Table 3-69
dcbi	**r**A,**r**B	Cache control	Table 3-63
dcblc	CT,**r**A,**r**B	Cache locking	Table 3-64
dcblq.	CT,**r**A,**r**B	Cache locking	Table 3-64
dcbst	**r**A,**r**B	Cache control	Table 3-63
dcbstep	**r**A,**r**B	External PID load/store	Table 3-69
dcbt	TH,**r**A,**r**B	Cache control	Table 3-63
dcbtep	TH,**r**A,**r**B	External PID load/store	Table 3-69
dcbtls	CT,**r**A,**r**B	Cache locking	Table 3-64
dcbtst	CT,**r**A,**r**B	Cache control	Table 3-63
dcbtstep	TH,**r**A,**r**B	External PID load/store	Table 3-69
dcbtstls	CT,**r**A,**r**B	Cache locking	Table 3-64

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
dcbz	rA,rB	Cache control	Table 3-63
dcbzep	rA,rB	External PID load/store	Table 3-69
dcbzl	rA,rB	Extended cache line/cache control	Table 3-63
dcbzlep	rA,rB	External PID load/store	Table 3-69
divd	rD,rA,rB	Integer	Table 3-5
divd.	rD,rA,rB	Integer	Table 3-5
divdo	rD,rA,rB	Integer	Table 3-5
divdo.	rD,rA,rB	Integer	Table 3-5
divdu	rD,rA,rB	Integer	Table 3-5
divdu.	rD,rA,rB	Integer	Table 3-5
divduo	rD,rA,rB	Integer	Table 3-5
divduo.	rD,rA,rB	Integer	Table 3-5
divw	rD,rA,rB	Integer	Table 3-5
divw.	rD,rA,rB	Integer	Table 3-5
divwo	rD,rA,rB	Integer	Table 3-5
divwo.	rD,rA,rB	Integer	Table 3-5
divwu	rD,rA,rB	Integer	Table 3-5
divwu.	rD,rA,rB	Integer	Table 3-5
divwuo	rD,rA,rB	Integer	Table 3-5
divwuo.	rD,rA,rB	Integer	Table 3-5
dnh	DUI,DCTL	Debug	Table 3-74
dni	DUI,DCTL	Debug	Table 3-74
dsn	rA,rB	Decorated load/store	Table 3-18
dss	STRM	AltiVec compatibility	Table 3-40
dssall		AltiVec compatibility	Table 3-40
dst	**STRM**,vD,vA,vB	AltiVec compatibility	Table 3-40
dstst	**STRM**,vD,vA,vB	AltiVec compatibility	Table 3-40
dststt	**STRM**,vD,vA,vB	AltiVec compatibility	Table 3-40
dstt	**STRM**,vD,vA,vB	AltiVec compatibility	Table 3-40
ehpriv	OC	Hypervisor	Table 3-57."
eqv	rA,rS,rB	Integer logical	Table 3-7
eqv.	rA,rS,rB	Integer logical	Table 3-7
extsb	rA,rS	Integer logical	Table 3-7

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
extsb.	rA,rS	Integer logical	Table 3-7
extsh	rA,rS	Integer logical	Table 3-7
extsh.	rA,rS	Integer logical	Table 3-7
extsw	rA,rS	Integer logical	Table 3-7
extsw.	rA,rS	Integer logical	Table 3-7
fabs	frD,frB	Floating-point	Table 3-19
fabs.	frD,frB	Floating-point	Table 3-19
fadd	frD,frA,frB	Floating-point	Table 3-19
fadd.	frD,frA,frB	Floating-point	Table 3-19
fadds	frD,frA,frB	Floating-point	Table 3-19
fadds.	frD,frA,frB	Floating-point	Table 3-19
fcfid	frD,frB	Floating-point	Table 3-19
fcfid.	frD,frB	Floating-point	Table 3-19
fcmpo	crfD,frA,frB	Floating-point	Table 3-19
fcmpu	crfD,frA,frB	Floating-point	Table 3-19
fctid	frD,frB	Floating-point	Table 3-19
fctid.	frD,frB	Floating-point	Table 3-19
fctidz	frD,frB	Floating-point	Table 3-19
fctidz.	frD,frB	Floating-point	Table 3-19
fctiw	frD,frB	Floating-point	Table 3-19
fctiw.	frD,frB	Floating-point	Table 3-19
fctiwz	frD,frB	Floating-point	Table 3-19
fctiwz.	frD,frB	Floating-point	Table 3-19
fdiv	frD,frA,frB	Floating-point	Table 3-19
fdiv.	frD,frA,frB	Floating-point	Table 3-19
fdivs	frD,frA,frB	Floating-point	Table 3-19
fdivs.	frD,frA,frB	Floating-point	Table 3-19
fmadd	frD,frA,frC,frB	Floating-point	Table 3-20
fmadd.	frD,frA,frC,frB	Floating-point	Table 3-20
fmadds	frD,frA,frC,frB	Floating-point	Table 3-20
fmadds.	frD,frA,frC,frB	Floating-point	Table 3-20
fmr	frD,frB	Floating-point	Table 3-24
fmr.	frD,frB	Floating-point	Table 3-24

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
fmsub	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-19
fmsub.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-19
fmsubs	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-19
fmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-19
fmul	**fr**D,**fr**A,**fr**C	Floating-point	Table 3-19
fmul.	**fr**D,**fr**A,**fr**C	Floating-point	Table 3-19
fmuls	**fr**D,**fr**A,**fr**C	Floating-point	Table 3-19
fmuls.	**fr**D,**fr**A,**fr**C	Floating-point	Table 3-19
fnabs	**fr**D,**fr**B	Floating-point	Table 3-19
fnabs.	**fr**D,**fr**B	Floating-point	Table 3-19
fneg	**fr**D,**fr**B	Floating-point	Table 3-19
fneg.	**fr**D,**fr**B	Floating-point	Table 3-19
fnmadd	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-20
fnmadd.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-20
fnmadds	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-20
fnmadds.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-20
fnmsub	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-20
fnmsub.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-20
fnmsubs	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-20
fnmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-20
fres	**fr**D,**fr**B	Floating-point	Table 3-20
fres.	**fr**D,**fr**B	Floating-point	Table 3-20
frsp	**fr**D,**fr**B	Floating-point	Table 3-20
frsp.	**fr**D,**fr**B	Floating-point	Table 3-20
frsqrte	**fr**D,**fr**B	Floating-point	Table 3-20
frsqrte.	**fr**D,**fr**B	Floating-point	Table 3-20
fsel	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-20
fsel.	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	Table 3-20
fsub	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-20
fsub.	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-20
fsubs	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-20
fsubs.	**fr**D,**fr**A,**fr**B	Floating-point	Table 3-20
icbi	**fr**A,**fr**B	Cache control	Table 3-63

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
icbiep	rA,rB	External PID load/store	Table 3-69
icblc	CT,rA,rB	Cache locking	Table 3-64
icblq.	CT,rA,rB	Cache locking	Table 3-64
icbt	CT,rA,rB	Cache control	Table 3-63
icbtls	CT,rA,rB	Cache locking	Table 3-64
isel	rD,rA,rB,crbC	Integer select	Table 3-53
isync	—	Synchronization	Table 3-62
lbarx	rD,rA,rB	Synchronization	Table 3-62
lbdx	rD,rA,rB	Decorated load/store	Table 3-18
lbepx	rD,rA,rB	External PID load/store	Table 3-69
lbz	rD,d(rA)	Integer load	Table 3-10
lbzu	rD,d(rA)	Integer load	Table 3-10
lbzux	rD,rA,rB	Integer load	Table 3-10
lbzx	rD,rA,rB	Integer load	Table 3-10
ld	rD,d(rA)	Integer load	Table 3-10
ldarx	rD,rA,rB	Synchronization	Table 3-62
ldbrx	rD,rA,rB	Integer load/store w/byte reverse	Table 3-12
lddx	rD,rA,rB	Decorated load/store	Table 3-18
ldepx	rD,rA,rB	External PID load/store	Table 3-69
ldu	rD,d(rA)	Integer load	Table 3-10
ldux	rD,rA,rB	Integer load	Table 3-10
ldx	rD,rA,rB	Integer load	Table 3-10
lfd	frD,d(rA)	Floating-point load/store	Table 3-14
lfddx	frD,rA,rB	Decorated load/store	Table 3-18
lfdepx	frD,rA,rB	External PID load/store	Table 3-69
lfdu	frD,d(rA)	Floating-point load/store	Table 3-14
lfdux	frD,rA,rB	Floating-point load/store	Table 3-14
lfdx	frD,rA,rB	Floating-point load/store	Table 3-14
lfs	frD,d(rA)	Floating-point load/store	Table 3-14
lfsu	frD,d(rA)	Floating-point load/store	Table 3-14
lfsux	frD,rA,rB	Floating-point load/store	Table 3-14
lfsx	frD,rA,rB	Floating-point load/store	Table 3-14
lha	rD,d(rA)	Integer load	Table 3-10

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
lharx	rD,rA,rB	Synchronization	Table 3-62
lhau	rD,d(rA)	Integer load	Table 3-10
lhaux	rD,rA,rB	Integer load	Table 3-10
lhax	rD,rA,rB	Integer load	Table 3-10
lhbrx	rD,rA,rB	Integer load/store w/byte reverse	Table 3-12
lhdx	rD,rA,rB	Decorated load/store	Table 3-18
lhepx	rD,rA,rB	External PID load/store	Table 3-69
lhz	rD,d(rA)	Integer load	Table 3-10
lhzu	rD,d(rA)	Integer load	Table 3-10
lhzux	rD,rA,rB	Integer load	Table 3-10
lhzx	rD,rA,rB	Integer load	Table 3-10
lmw	rD,d(rA)	Integer load	Table 3-10
lvebx	vD,rA,rB	AltiVec load	Table 3-16
lvehx	vD,rA,rB	AltiVec load	Table 3-16
lvepx	vD,rA,rB	External PID load/store	Table 3-69
lvepxl	vD,rA,rB	External PID load/store	Table 3-69
lvewx	vD,rA,rB	AltiVec load	Table 3-16
lvexbx	vD,rA,rB	AltiVec load	Table 3-16
lvexhx	vD,rA,rB	AltiVec load	Table 3-16
lvexwx	vD,rA,rB	AltiVec load	Table 3-16
lvtlx	vD,rA,rB	AltiVec load	Table 3-16
lvtlxl	vD,rA,rB	AltiVec load	Table 3-16
lvtrx	vD,rA,rB	AltiVec load	Table 3-16
lvtrxl	vD,rA,rB	AltiVec load	Table 3-16
lvsl	vD,rA,rB	AltiVec load	Table 3-16
lvsm	vD,rA,rB	AltiVec load	Table 3-16
lvsr	vD,rA,rB	AltiVec load	Table 3-16
lvswx	vD,rA,rB	AltiVec load	Table 3-16
lvswxl	vD,rA,rB	AltiVec load	Table 3-16
lvx	vD,rA,rB	AltiVec load	Table 3-16
lvxl	vD,rA,rB	AltiVec load	Table 3-16
lwa	rD,d(rA)	Integer load	Table 3-10
lwarx	rD,rA,rB	Synchronization	Table 3-62

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
lwaux	**r**D,**r**A,**r**B	Integer load	Table 3-10
lwax	**r**D,**r**A,**r**B	Integer load	Table 3-10
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 3-18
lwepx	**r**D,**r**A,**r**B	External PID load/store	Table 3-69
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 3-62
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 3-58
mfcr	**r**D	Condition register logical	Table 3-58
mffs	**fr**D	FPSCR	Table 3-23
mffs.	**fr**D	FPSCR	Table 3-23
mfmsr	**r**D	MSR	Table 3-67
mfocrf	**r**D,C**R**M	CR logical	Table 3-58
mfpmr	**r**D,PM**R**N	Move from PMR	Table 3-73
mfspr	**r**D,SP**R**	SPR	Table 3-59
mftb	**r**D,SP**R**	Move from time base	Table 3-59
mftmr	**r**D,TMRN	Thread Management	Table 3-68
mfvscr	**v**D	AltiVec status and control register	Table 3-48
miso	—	Cache control	Table 3-63
msgclr	**r**B	Doorbell	Table 3-72
msgsnd	**r**B	Doorbell	Table 3-72
mtcrf	C**R**M,**r**S	Condition register logical	Table 3-58
mtfsb0	**crb**D_FP	FPSCR	Table 3-23
mtfsb0.	**crb**D_FP	FPSCR	Table 3-23
mtfsb1	**crb**D_FP	FPSCR	Table 3-23
mtfsb1.	**crb**D_FP	FPSCR	Table 3-23
mtfsf	FM,fB	FPSCR	Table 3-23
mtfsf.	FM,fB	FPSCR	Table 3-23

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
mtfsfi	**crf**D_FP,FP_IMM	FPSCR	Table 3-23
mtfsfi.	**crf**D_FP,FP_IMM	FPSCR	Table 3-23
mtmsr	**r**S	MSR	Table 3-67
mtocrf	C**R**M,**r**S	CR logical	Table 3-58
mtpmr	PM**R**N,**r**S	Move to PMR	Table 3-61
mtspr	SP**R**,**r**S	SPR	Table 3-59
mttmr	TMRN,**r**S	Thread Management	Table 3-68
mtvscr	**v**B	AltiVec status and control register	Table 3-48
mulhd	**r**D,**r**A,**r**B	Integer	Table 3-5
mulhd.	**r**D,**r**A,**r**B	Integer	Table 3-5
mulhdu	**r**D,**r**A,**r**B	Integer	Table 3-5
mulhdu.	**r**D,**r**A,**r**B	Integer	Table 3-5
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
mulld	**r**D,**r**A,**r**B	Integer	Table 3-5
mulld.	**r**D,**r**A,**r**B	Integer	Table 3-5
mulldo	**r**D,**r**A,**r**B	Integer	Table 3-5
mulldo.	**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
mvidsplt	**v**D,**r**A,**r**B	GPR to AltiVec move	Table 3-49
mviwsplt	**v**D,**r**A,**r**B	GPR to AltiVec move	Table 3-49
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
nego	**r**D,**r**A	Integer	Table 3-5
nego.	**r**D,**r**A	Integer	Table 3-5

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
nor	rA,rS,rB	Integer logical	Table 3-7
nor.	rA,rS,rB	Integer logical	Table 3-7
or	rA,rS,rB	Integer logical	Table 3-7
or.	rA,rS,rB	Integer logical	Table 3-7
orc	rA,rS,rB	Integer logical	Table 3-7
orc.	rA,rS,rB	Integer logical	Table 3-7
ori	rA,rS,UIMM	Integer logical	Table 3-7
oris	rA,rS,UIMM	Integer logical	Table 3-7
popcntb	rA,rS	Integer logical	Table 3-7
popcntd	rA,rS	Integer logical	Table 3-7
popcntw	rA,rS	Integer logical	Table 3-7
prtyd	rA,rS	Integer logical	Table 3-7
prtyw	rA,rS	Integer logical	Table 3-7
rfci	—	System Linkage	Table 3-66
rfdi	—	System Linkage	Table 3-66
rfgi	—	System Linkage	Table 3-66
rfi	—	System Linkage	Table 3-66
rfmci	—	System Linkage	Table 3-66
rldcl	rA,rS,rB,MB	Integer rotate	Table 3-8
rldcl.	rA,rS,rB,MB	Integer rotate	Table 3-8
rldcr	rA,rS,rB,ME	Integer rotate	Table 3-8
rldcr.	rA,rS,rB,ME	Integer rotate	Table 3-8
rldic	rA,rS,SH,MB	Integer rotate	Table 3-8
rldic.	rA,rS,SH,MB	Integer rotate	Table 3-8
rldicl	rA,rS,SH,MB	Integer rotate	Table 3-8
rldicl.	rA,rS,SH,MB	Integer rotate	Table 3-8
rldicr	rA,rS,SH,ME	Integer rotate	Table 3-8
rldicr.	rA,rS,SH,ME	Integer rotate	Table 3-8
rldimi	rA,rS,SH,MB	Integer rotate	Table 3-8
rldimi.	rA,rS,SH,MB	Integer rotate	Table 3-8
rlwimi	rA,rS,SH,MB,ME	Integer rotate	Table 3-8
rlwimi.	rA,rS,SH,MB,ME	Integer rotate	Table 3-8
rlwinm	rA,rS,SH,MB,ME	Integer rotate	Table 3-8

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
rlwinm.	rA,rS,SH,MB,ME	Integer rotate	Table 3-8
rlwnm	rA,rS,rB,MB,ME	Integer rotate	Table 3-8
rlwnm.	rA,rS,rB,MB,ME	Integer rotate	Table 3-8
sc	LEV	System call	Table 3-8
sld	rA,rS,rB	Integer shift	Table 3-9
sld.	rA,rS,rB	Integer shift	Table 3-9
slw	rA,rS,rB	Integer shift	Table 3-9
slw.	rA,rS,rB	Integer shift	Table 3-9
srad	rA,rS,rB	Integer shift	Table 3-9
srad.	rA,rS,rB	Integer shift	Table 3-9
sradi	rA,rS,SH	Integer shift	Table 3-9
sradi.	rA,rS,SH	Integer shift	Table 3-9
sraw	rA,rS,rB	Integer shift	Table 3-9
sraw.	rA,rS,rB	Integer shift	Table 3-9
srawi	rA,rS,SH	Integer shift	Table 3-9
srawi.	rA,rS,SH	Integer shift	Table 3-9
srd	rA,rS,rB	Integer shift	Table 3-9
srd.	rA,rS,rB	Integer shift	Table 3-9
srw	rA,rS,rB	Integer shift	Table 3-9
srw.	rA,rS,rB	Integer shift	Table 3-9
stb	rS,d(rA)	Integer store	Table 3-11
stbcx.	rS,rA,rB	Synchronization	Table 3-62
stbdx	rS,rA,rB	Decorated load/store	Table 3-18
stbepx	rS,rA,rB	External PID load/store	Table 3-69
stbu	rS,d(rA)	Integer store	Table 3-11
stbux	rS,rA,rB	Integer store	Table 3-11
stbx	rS,rA,rB	Integer store	Table 3-11
std	rS,d(rA)	Integer store	Table 3-11
stdbrx	rS,rA,rB	Integer load/store w/byte reverse	Table 3-12
stdcx.	rS,rA,rB	Synchronization	Table 3-62
stddx	rS,rA,rB	Decorated load/store	Table 3-18
stdepx	rS,rA,rB	External PID load/store	Table 3-69
stdu	rS,d(rA)	Integer store	Table 3-11

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
stdux	rS,rA,rB	Integer store	Table 3-11
stdx	rS,rA,rB	Integer store	Table 3-11
stfd	frS,d(rA)	Floating-point store	Table 3-15
stfddx	frS,rA,rB	Decorated load/store	Table 3-18
stfdepx	frS,rA,rB	External PID load/store	Table 3-69
stfdu	frS,d(rA)	Floating-point store	Table 3-15
stfdux	frS,rA,rB	Floating-point store	Table 3-15
stfdx	frS,rA,rB	Floating-point store	Table 3-15
stfiwx	frS,rA,rB	Floating-point store	Table 3-15
stfs	frS,d(rA)	Floating-point store	Table 3-15
stfsu	frS,d(rA)	Floating-point store	Table 3-15
stfsux	frS,rA,rB	Floating-point store	Table 3-15
stfsx	frS,rA,rB	Floating-point store	Table 3-15
sth	rS,d(rA)	Integer store	Table 3-11
sthbrx	rS,rA,rB	Integer load/store w/byte reverse	Table 3-12
sthcx.	rS,rA,rB	Synchronization	Table 3-62
sthdx	rS,rA,rB	Decorated load/store	Table 3-18
sthepx	rS,rA,rB	External PID load/store	Table 3-69
sthu	rS,d(rA)	Integer store	Table 3-11
sthux	rS,rA,rB	Integer store	Table 3-11
sthx	rS,rA,rB	Integer store	Table 3-11
stmw	rS,d(rA)	Integer store	Table 3-11
stvebx	vS,rA,rB	AltiVec store	Table 3-17
stvehx	vS,rA,rB	AltiVec store	Table 3-17
stvepx	vS,rA,rB	External PID load/store	Table 3-69
stvepxl	vS,rA,rB	External PID load/store	Table 3-69
stvewx	vS,rA,rB	AltiVec store	Table 3-17
stvexbx	vS,rA,rB	AltiVec store	Table 3-17
stvexhx	vS,rA,rB	AltiVec store	Table 3-17
stvexwx	vS,rA,rB	AltiVec store	Table 3-17
stvflx	vS,rA,rB	AltiVec store	Table 3-17
stvflxl	vS,rA,rB	AltiVec store	Table 3-17
stvfrx	vS,rA,rB	AltiVec store	Table 3-17

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
stvfrxl	vS,rA,rB	AltiVec store	Table 3-17
stvswx	vS,rA,rB	AltiVec store	Table 3-17
stvswxl	vS,rA,rB	AltiVec store	Table 3-17
stvx	vS,rA,rB	AltiVec store	Table 3-17
stvxl	vS,rA,rB	AltiVec store	Table 3-17
stw	rS,d(rA)	Integer store	Table 3-11
stwbrx	rS,rA,rB	Integer load/store w/byte reverse	Table 3-12
stwcx.	rS,rA,rB	Synchronization	Table 3-62
stwdx	rS,rA,rB	Decorated load/store	Table 3-18
stwepx	rS,rA,rB	External PID load/store	Table 3-69
stwu	rS,d(rA)	Integer store	Table 3-11
stwux	rS,rA,rB	Integer store	Table 3-11
stwx	rS,rA,rB	Integer store	Table 3-11
subf	rD,rA,rB	Integer	Table 3-5
subf.	rD,rA,rB	Integer	Table 3-5
subfc	rD,rA,rB	Integer	Table 3-5
subfc.	rD,rA,rB	Integer	Table 3-5
subfco	rD,rA,rB	Integer	Table 3-5
subfco.	rD,rA,rB	Integer	Table 3-5
subfe	rD,rA,rB	Integer	Table 3-5
subfe.	rD,rA,rB	Integer	Table 3-5
subfeo	rD,rA,rB	Integer	Table 3-5
subfeo.	rD,rA,rB	Integer	Table 3-5
subfic	rD,rA,SIMM	Integer	Table 3-5
subfme	rD,rA	Integer	Table 3-5
subfme.	rD,rA	Integer	Table 3-5
subfmeo	rD,rA	Integer	Table 3-5
subfmeo.	rD,rA	Integer	Table 3-5
subfo	rD,rA,rB	Integer	Table 3-5
subfo.	rD,rA,rB	Integer	Table 3-5
subfze	rD,rA	Integer	Table 3-5
subfze.	rD,rA	Integer	Table 3-5
subfzeo	rD,rA	Integer	Table 3-5

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
subfzeo.	rD,rA	Integer	Table 3-5
sync (msync)	L,E	Synchronization	Table 3-62
td	TO,rA,rB	Trap	Table 3-55
tdi	TO,rA,SIMM	Trap	Table 3-55
tlbilx	T,rA,rB	TLB management	Table 3-71
tlbivax	rA,rB	TLB management	Table 3-71
tlbre	—	TLB management	Table 3-71
tlbsx	rA,rB	TLB management	Table 3-71
tlbsync	—	TLB management	Table 3-71
tlbwe	—	TLB management	Table 3-71
tw	TO,rA,rB	Trap	Table 3-55
twi	TO,rA,SIMM	Trap	Table 3-55
vabsdub	vD,vA,vB	AltiVec integer	Table 3-25
vabsduh	vD,vA,vB	AltiVec integer	Table 3-25
vabsduw	vD,vA,vB	AltiVec integer	Table 3-25
vaddcuw	vD,vA,vB	AltiVec integer	Table 3-25
vaddfp	vD,vA,vB	AltiVec floating-point arithmetic	Table 3-35
vaddsbs	vD,vA,vB	AltiVec integer	Table 3-25
vaddshs	vD,vA,vB	AltiVec integer	Table 3-25
vaddsws	vD,vA,vB	AltiVec integer	Table 3-25
vaddubm	vD,vA,vB	AltiVec integer	Table 3-25
vaddubs	vD,vA,vB	AltiVec integer	Table 3-25
vadduhm	vD,vA,vB	AltiVec integer	Table 3-25
vadduhs	vD,vA,vB	AltiVec integer	Table 3-25
vadduwm	vD,vA,vB	AltiVec integer	Table 3-25
vadduws	vD,vA,vB	AltiVec integer	Table 3-25
vand	vD,vA,vB	AltiVec logical	Table 3-27
vandc	vD,vA,vB	AltiVec logical	Table 3-27
vavgsb	vD,vA,vB	AltiVec integer	Table 3-25
vavgsh	vD,vA,vB	AltiVec integer	Table 3-25
vavgsw	vD,vA,vB	AltiVec integer	Table 3-25
vavgub	vD,vA,vB	AltiVec integer	Table 3-25

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
vavguh	vD,vA,vB	AltiVec integer	Table 3-25
vavguw	vD,vA,vB	AltiVec integer	Table 3-25
vcfsx	vD,vB, **UIMM**	AltiVec floating-point rounding and conversion	Table 3-37
vcfux	vD,vB, **UIMM**	AltiVec floating-point rounding and conversion	Table 3-37
vcmpbfp	vD,vA,vB	AltiVec floating-point compare	Table 3-38
vcmpbfp.	vD,vA,vB	AltiVec floating-point compare	Table 3-38
vcmpeqfp	vD,vA,vB	AltiVec floating-point compare	Table 3-38
vcmpeqfp.	vD,vA,vB	AltiVec floating-point compare	Table 3-38
vcmpequb	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpequb.	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpequh	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpequh.	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpequw	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpequw.	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpgefp	vD,vA,vB	AltiVec floating-point compare	Table 3-38
vcmpgefp.	vD,vA,vB	AltiVec floating-point compare	Table 3-38
vcmpgtfp	vD,vA,vB	AltiVec floating-point compare	Table 3-38
vcmpgtfp.	vD,vA,vB	AltiVec floating-point compare	Table 3-38
vcmpgtsb	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpgtsb.	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpgtsh.	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpgtsw	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpgtsw.	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpgtub	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpgtub.	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpgtuh	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpgtuh.	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpgtuw	vD,vA,vB	AltiVec integer compare	Table 3-26
vcmpgtuw.	vD,vA,vB	AltiVec integer compare	Table 3-26
vctsxs	vD,vB, **UIMM**	AltiVec floating-point rounding and conversion	Table 3-37

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
vctuxs	vD,vB, **UIMM**	AltiVec floating-point rounding and conversion	Table 3-37
vexptefp	vD,vB	AltiVec floating-point estimate	Table 3-39
vlogefp	vD,vB	AltiVec floating-point estimate	Table 3-39
vmaddfp	vD,vA,vC,vB	AltiVec floating-point mulptipy-add	Table 3-36
vmaxfp	vD,vA,vB	AltiVec floating-point arithmetic	Table 3-35
vmaxsb	vD,vA,vB	AltiVec integer	Table 3-25
vmaxsh	vD,vA,vB	AltiVec integer	Table 3-25
vmaxsw	vD,vA,vB	AltiVec integer	Table 3-25
vmaxub	vD,vA,vB	AltiVec integer	Table 3-25
vmaxuh	vD,vA,vB	AltiVec integer	Table 3-25
vmaxuw	vD,vA,vB	AltiVec integer	Table 3-25
vmhaddshs	vD,vA,vB,vC	AltiVec integer	Table 3-25
vmhraddshs	vD,vA,vB,vC	AltiVec integer	Table 3-25
vminfp	vD,vA,vB	AltiVec floating-point arithmetic	Table 3-35
vminsb	vD,vA,vB	AltiVec integer	Table 3-25
vminsh	vD,vA,vB	AltiVec integer	Table 3-25
vminsw	vD,vA,vB	AltiVec integer	Table 3-25
vminub	vD,vA,vB	AltiVec integer	Table 3-25
vminuh	vD,vA,vB	AltiVec integer	Table 3-25
vminuw	vD,vA,vB	AltiVec integer	Table 3-25
vmladduhm	vD,vA,vB,vC	AltiVec integer	Table 3-25
vmrghb	vD,vA,vB	AltiVec merge	Table 3-43
vmrghh	vD,vA,vB	AltiVec merge	Table 3-43
vmrghw	vD,vA,vB	AltiVec merge	Table 3-43
vmrglb	vD,vA,vB	AltiVec merge	Table 3-43
vmrglh	vD,vA,vB	AltiVec merge	Table 3-43
vmrglw	vD,vA,vB	AltiVec merge	Table 3-43
vmsummbm	vD,vA,vB,vC	AltiVec integer	Table 3-25
vmsumshm	vD,vA,vB,vC	AltiVec integer	Table 3-25
vmsumshs	vD,vA,vB,vC	AltiVec integer	Table 3-25
vmsumubm	vD,vA,vB,vC	AltiVec integer	Table 3-25
vmsumuhm	vD,vA,vB,vC	AltiVec integer	Table 3-25

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
vmsumuhs	vD,vA,vB,vC	AltiVec integer	Table 3-25
vmulesb	vD,vA,vB	AltiVec integer	Table 3-25
vmulesh	vD,vA,vB	AltiVec integer	Table 3-25
vmuleub	vD,vA,vB	AltiVec integer	Table 3-25
vmuleuh	vD,vA,vB	AltiVec integer	Table 3-25
vmulosb	vD,vA,vB	AltiVec integer	Table 3-25
vmulosh	vD,vA,vB	AltiVec integer	Table 3-25
vmuloub	vD,vA,vB	AltiVec integer	Table 3-25
vmulouh	vD,vA,vB	AltiVec integer	Table 3-25
vnmsubfp	vD,vA,vC,vB	AltiVec floating-point multiply-add	Table 3-36
vnor	vD,vA,vB	AltiVec logical	Table 3-27
vor	vD,vA,vB	AltiVec logical	Table 3-27
vperm	vD,vA,vB,vC	AltiVec permute	Table 3-45
vpkpx	vD,vA,vB	AltiVec pack	Table 3-41
vpkshss	vD,vA,vB	AltiVec pack	Table 3-41
vpkshus	vD,vA,vB	AltiVec pack	Table 3-41
vpkswss	vD,vA,vB	AltiVec pack	Table 3-41
vpkswus	vD,vA,vB	AltiVec pack	Table 3-41
vpkuhum	vD,vA,vB	AltiVec pack	Table 3-41
vpkuhus	vD,vA,vB	AltiVec pack	Table 3-41
vpkuwum	vD,vA,vB	AltiVec pack	Table 3-41
vpkuwus	vD,vA,vB	AltiVec pack	Table 3-41
vrefp	vD,vB	AltiVec floating-point estimate	Table 3-39
vrfim	vD,vB	AltiVec floating-point rounding and conversion	Table 3-37
vrfin	vD,vB	AltiVec floating-point rounding and conversion	Table 3-37
vrfip	vD,vB	AltiVec floating-point rounding and conversion	Table 3-37
vrfiz	vD,vB	AltiVec floating-point rounding and conversion	Table 3-37
vrlb	vD,vA,vB	AltiVec shift and rotate	Table 3-28
vrlh	vD,vA,vB	AltiVec shift and rotate	Table 3-28
vrlw	vD,vA,vB	AltiVec shift and rotate	Table 3-28

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
vrsqrtefp	vD,vB	AltiVec floating-point estimate	Table 3-39
vsel	vD,vA,vB,vC	AltiVec select	Table 3-46
vsl	vD,vA,vB	AltiVec shift	Table 3-47
vslb	vD,vA,vB	AltiVec shift and rotate	Table 3-29
vsldoi	vD,vA,vB,SH	AltiVec shift	Table 3-47
vslh	vD,vA,vB	AltiVec shift and rotate	Table 3-29
vslo	vD,vA,vB	AltiVec shift	Table 3-47
vslw	vD,vA,vB	AltiVec shift and rotate	Table 3-29
vspltb	vD,vB,**UIMM**	AltiVec splat	Table 3-44
vsplth	vD,vB,**UIMM**	AltiVec splat	Table 3-44
vspltisb	vD,**SIMM**	AltiVec splat	Table 3-44
vspltish	vD,**SIMM**	AltiVec splat	Table 3-44
vspltisw	vD,**SIMM**	AltiVec splat	Table 3-44
vspltw	vD,vB,**UIMM**	AltiVec splat	Table 3-44
vsr	vD,vA,vB	AltiVec shift	Table 3-47
vsrab	vD,vA,vB	AltiVec shift and rotate	Table 3-29
vsrah	vD,vA,vB	AltiVec shift and rotate	Table 3-29
vsraw	vD,vA,vB	AltiVec shift and rotate	Table 3-29
vsrb	vD,vA,vB	AltiVec shift and rotate	Table 3-29
vsrh	vD,vA,vB	AltiVec shift and rotate	Table 3-29
vsro	vD,vA,vB	AltiVec shift	Table 3-47
vsrw	vD,vA,vB	AltiVec shift and rotate	Table 3-29
vsubcuw	vD,vA,vB	AltiVec integer	Table 3-25
vsubfp	vD,vA,vB	AltiVec floating-point arithmetic	Table 3-35
vsubsbs	vD,vA,vB	AltiVec integer	Table 3-25
vsubshs	vD,vA,vB	AltiVec integer	Table 3-25
vsubsws	vD,vA,vB	AltiVec integer	Table 3-25
vsububm	vD,vA,vB	AltiVec integer	Table 3-25
vsububs	vD,vA,vB	AltiVec integer	Table 3-25
vsubuhm	vD,vA,vB	AltiVec integer	Table 3-25
vsubuhs	vD,vA,vB	AltiVec integer	Table 3-25
vsubuwm	vD,vA,vB	AltiVec integer	Table 3-25
vsubuws	vD,vA,vB	AltiVec integer	Table 3-25

Table 3-75. e6500 core instruction set (continued)

Mnemonic	Syntax	Classification	Cross-Reference
vsum2sws	vD,vA,vB	AltiVec integer	Table 3-25
vsum4sbs	vD,vA,vB	AltiVec integer	Table 3-25
vsum4shs	vD,vA,vB	AltiVec integer	Table 3-25
vsum4ubs	vD,vA,vB	AltiVec integer	Table 3-25
vsumsws	vD,vA,vB	AltiVec integer	Table 3-25
vupkhpx	vD,vB	AltiVec unpack	Table 3-42
vupkhsb	vD,vB	AltiVec unpack	Table 3-42
vupkhsb	vD,vB	AltiVec unpack	Table 3-42
vupklpx	vD,vB	AltiVec unpack	Table 3-42
vupklsb	vD,vB	AltiVec unpack	Table 3-42
vupklsh	vD,vB	AltiVec unpack	Table 3-42
vxor	vD,vA,vB	AltiVec logical	Table 3-27
wait	WC,WH	Wait	Table 3-62
wrtee	rS	MSR	Table 3-67
wrteei	E	MSR	Table 3-67
xor	rA,rS,rB	Integer logical	Table 3-7
xor.	rA,rS,rB	Integer logical	Table 3-7
xori	rA,rS,UIMM	Integer logical	Table 3-7
xoris	rA,rS,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 e6500 core. It identifies and describes the portions of the interrupt model that are defined by *EREF* and those that are specific to the e6500 core.

4.1 Overview

Although the e6500 core has two simultaneously executing threads, most interrupts only interrupt one thread. If both threads are enabled and one thread is interrupted, the other thread continues to execute normally because it is logically a separate processor.

A note on terminology:

The terms ‘interrupt’ and ‘exception’ are used as follows:

- 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.
- 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 causes 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, described as follows:

- Standard interrupts
- Critical interrupts
- Debug interrupts
- Machine check 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 the state when they are taken, and they use the **rfi** instruction to restore that state. Asynchronous, non-critical interrupts can be masked by the external interrupt enable bit, MSR[EE], when not in a 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 the state when they are taken, and they use the **rfgi** instruction to restore that state. Guest interrupts are listed in Table 2-11.

Section 2.2.2.1, “Register mapping in the 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.

Critical interrupts (critical input and watchdog timer interrupts) 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 input and watchdog timer critical interrupts can be masked by the critical enable bit, MSR[CE].

Debug interrupts 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].

Machine check interrupts 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].

The e6500 core 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 e6500 core details, see Section 4.9.3, “Machine check interrupt—IVOR1.”

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 to change the state of the SRR0/SRR1 registers before software can save them, which destroys the return path to the original interrupt. See Section 4.6.1, “Interrupt ordering and masking,” for additional details.

All interrupts except the machine check interrupt are context synchronizing, as defined in the instruction model chapter of *EREF*. A machine check interrupt acts like a context-synchronizing operation with respect to subsequent instructions.

4.2 e6500 implementation of interrupt architecture

This section describes the architecture-defined interrupt model as implemented on the e6500 core. Specific details are also provided throughout this chapter. Each thread of the e6500 core implements all the interrupts defined by the embedded category and implements the following interrupts that are defined by—but not required by—optional parts of the embedded architecture:

- In general, each e6500 thread 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).

  Each thread of the e6500 core implements three types of machine check interrupts: asynchronous, error report, and NMI. Some asynchronous machine check events are logged directly into the MCSR of both threads. If such an event is logged in the MCSR and MSR[ME] = 1 or MSR[GS] = 1, a machine check interrupt is taken by a thread. But, if some of the MCSR asynchronous bits are set and MSR[ME] = 0 and MSR[GS] = 0, the asynchronous machine check exception is pending. If these bits are still set when MSR[ME] or MSR[GS] is changed to 1, the asynchronous machine check interrupt occurs. The e6500 core does not take a checkstop, as is the case with some previous e500 cores.

  In addition, each thread of the e6500 core 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 do not occur for instructions that have not completed and deallocated. For example, the thread 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).
- Each thread of the e6500 core implements interrupt signals from the integrated device for external input critical input, machine check, and NMI. There is one set of these signals for each thread from the interrupt controller on the integrated device. See the integrated device reference manual for more information.
- Each thread of the e6500 core implements debug interrupts as described by the embedded.enhanced debug category, which provides a separate set of debug save/restore registers (DSRR0 and DSRR1) per thread.
- Each thread of the e6500 core implements the performance monitor interrupt from the embedded.performance monitor category.
- Each thread of the e6500 core implements the enabled floating-point exception (program interrupt) and the floating-point unavailable interrupt from the floating-point category.
- Each thread of the e6500 core implements the AltiVec available exception and the AltiVec unavailable interrupt from the vector category.
- Each thread of the e6500 core implements the AltiVec assist exception and the AltiVec assist interrupt from the vector category.
- Each thread of the e6500 core implements the following interrupts defined by the embedded.processor control category:
  — Processor doorbell
  — Processor doorbell critical
- Each thread of the e6500 core 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 e6500 core 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 an e6500 thread are directed to either the guest state or the hypervisor state. The state to which an interrupt is directed determines which SPRs are used to form the vector address, which save/restore register are used to capture the thread state at the time of the interrupt, and which ESR is used to post the 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 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 thread 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 [not a virtualization fault])
  - Instruction storage (EPCR[ISIGS] = 1 and (TLB[IND] = 0 or TLB[VF] = 0))
  - Performance monitor (EPCR[PMGS] = 1)

  Note that a data storage interrupt caused by a virtualization fault exception is always taken in the hypervisor state.

An interrupt is never directed to the guest state when the processor executes in the hypervisor state.

For more specific information about how interrupts are directed, see *EREF*.

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, the 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 the state was successfully saved. If MCSRR1[RI] is set, it is safe to return when processing is complete.

4.5 Interrupt registers

The following table summarizes registers used for interrupt handling. *EREF* 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.

Section 4.3, “Directed interrupts” describes, in detail, whether the interrupt is directed to hypervisor or guest-supervisor software.

Table 4-1. Interrupt registers

Register	Description
Save/Restore 0 (SRR0, CSRR0, DSRR0, GSRR0, MCSRR0) registers	On an interrupt, *x*SRR0 holds the EA at which thread 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 1 registers (SRR1, CSRR1, DSRR1, GSRR1, MCSRR1) registers	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.
(Guest)Data Exception Address (DEAR/GDEAR) registers	Contains the address referenced by a load, store, or cache management instruction that caused an alignment, data TLB miss, LRAT error, or data storage interrupt. When executing in the guest state (MSR[GS] = 1), accesses to DEAR are mapped to GDEAR upon executing **mtspr** or **mfspr**. DEAR and GDEAR are described in Section 2.9.2, “(Guest) Data Exception Address (DEAR/GDEAR) registers.”
(Guest)Exception Proxy (EPR/GEPR) registers	Defined by the external interrupt proxy facility, which is described in Section 4.9.6.1, “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 2.9.6, “(Guest) External Proxy (EPR/GEPR) registers.”

Table 4-1. Interrupt registers (continued)

Register	Description
(Guest)Interrupt Vector Prefix (IVPR/GIVPR) registers	(G)IVPR[0–47] provides the high-order 48 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. The IVPR is shared by the threads (processors) of an e6500 because it is expected that only one hypervisor will be present. In contrast, there is a GIVPR private to each thread since it is intended that threads can host different guests.
(Guest)Exception Syndrome (ESR/GESR) registers	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 2.9.7, "(Guest) Exception Syndrome (ESR/GESR) registers."
(Guest)Interrupt Vector Offset (IVORs/GIVORs) registers	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 e6500 core. Supported (G)IVORs and the associated interrupts are listed in Table 4-2. The IVOR is shared by the threads (processors) of an e6500 because it is expected that only one hypervisor will be present. In contrast, there is a GIVOR private to each thread since it is intended that threads can host different guests.
Machine-check address registers (MCAR/MCARU/ MCARUA)	On a machine check interrupt, MCAR/MCARU/MCARUA is updated with the address of the data associated with the machine check if applicable. See Section 2.9.9, "Machine-check address registers (MCAR/MCARU/MCARUA)."
Machine Check Syndrome (MCSR) registers	On a machine check interrupt, MCSR is updated with a syndrome to indicate exceptions, listed in Table 2-14 and fully described in the *EREF*. Section 2.9.10, "Machine Check Syndrome (MCSR) register," describes MCSR bit fields as they are defined for the e6500.

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 must look beyond ESR[BO] (to 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-type program interrupt)
- An attempt by an application program to execute a privileged instruction or to access a privileged SPR (privileged instruction exception-type program interrupt)

- An attempt to access a non-existent SPR (illegal-operation program exception on all accesses to undefined SPRs, regardless of MSR[GS,PR])
- An attempt to execute an instruction in guest-supervisor state that accesses hypervisor-only resources or is hypervisor privileged (embedded hypervisor privilege)
- An attempt to write a TLB entry in guest-supervisor state and a matching LRAT entry does not exist (LRAT error interrupt)
- An attempt to access a location that is either unavailable (instruction or data TLB error interrupt) or not permitted (instruction or data storage interrupt)
- An attempt to access a location with an effective address alignment not supported by the implementation (alignment interrupt)
- Execution of a System Call (**sc**) instruction (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 interrupt type)
- Execution of an unimplemented, defined instruction (illegal instruction exception)

Invocation of an interrupt is precise. When the interrupt is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts because the invocation of the interrupt required to complete execution has not occurred.

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 *EREF*.

4.7 Interrupt classification

All interrupts except machine check are grouped by two 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 the 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 asynchronous or synchronous. 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 cause the exception. *EREF* describes asynchronous and synchronous interrupts.

4.8 Interrupt processing

Each interrupt has a vector, that is, the address of the initial instruction that is executed when an interrupt occurs. When an interrupt is taken by a thread, 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 (MSR) register."
5. Instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt type ([G]IVPR[0–47] || (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 MSR to be restored from *x*SRR1 and instruction execution to resume at the address contained in *x*SRR0.

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 load and reserve instruction in the old process with a store conditional instruction 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

In the e6500 core, an asynchronous machine check interrupt is posted to both threads if the source of the machine check is common to both threads and the exception cannot be attributed to a particular thread.

This table summarizes each interrupt type, exceptions that may cause each 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	4-13
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,SPV], [EPID], [PT]	—	SP	SRRs	4-20
		Page table fault				SP		
		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 and¶ EPCR[DSIGS] = 1	[ST], [FP,SPV], [EPID], [PT]	—	SP	GSRRs	4-20
		Page table fault				SP		
		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	[PT]	—	SP	SRRs	4-23
		Page table fault				SP		
		Instruction virtualization fault	TLB[VF]=1 and TLB[IND]=1			SP		
GIVOR3	Instruction storage (ISI)	Access	MSR[GS] = 1 and EPCR[ISIGS] = 1	[PT]	—	SP	GSRRs	4-23
		Page table fault				SP		
IVOR4	External input [3]		EPCR[EXTGS] = 0	—	MSR[EE] or MSR[GS]	A	SRRs	4-25

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
GIVOR4	External input [3]		EPCR[EXTGS] = 1	—	MSR[EE] and MSR[GS]	A	GSRRs	4-25
IVOR5	Alignment		—	[ST],[FP,SPV], [EPID]	—	SP	SRRs	4-27
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 op		PUO[4]	—	SP		
IVOR7	Floating-point unavailable		—	—	—	SP	SRRs	4-29
IVOR8	System call		MSR[GS] = 0	—	—	SP*	SRRs	4-30
GIVOR8	System call		MSR[GS] = 1	—	—	SP*	GSRRs	4-30
IVOR10	Decrementer		—	—	(MSR[EE] or MSR[GS]) and TCR[DIE]	A	SRRs	4-31
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,SPV], [EPID]	—	SP	SRRs	4-33
GIVOR13	Data TLB error	Data TLB miss	MSR[GS] = 1 and EPCR[DTLBGS] = 1	[ST],[FP,SPV], [EPID]	—	SP	GSRRs	4-33
IVOR14	Instruction TLB error	Instruction TLB miss	MSR[GS] = 0 or EPCR[ITLBGS] = 0	—	—	SP	SRRs	4-34
GIVOR14	Instruction TLB error	Instruction TLB miss	MSR[GS] = 1 and EPCR[ITLBGS] = 1	—	—	SP	GSRRs	4-34

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 4.9.16.1, "Suppressing debug events in hypervisor mode."	SP[5]	DSRRs	4-35
		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		
IVOR32	AltiVec unavailable		—	SPV	—	SP	SRRs	4-36
IVOR33	AltiVec assist		—	SPV	—	SP	SRRs	4-37
IVOR35	Performance monitor		EPCR[PMGS] = 0	—	MSR[EE] or MSR[GS]	A	SRRs	4-37
GIVOR35	Performance monitor		EPCR[PMGS] = 1	—	MSR[EE] and MSR[GS]	A	GSRRs	4-37
IVOR36	Processor doorbell		—	—	MSR[EE] or MSR[GS]	A	SRRs	4-39
IVOR37	Processor doorbell critical		—	—	MSR[CE] or MSR[GS]	A	CSRRs	4-39
IVOR38	Guest processor doorbell		—	—	MSR[EE] and MSR[GS]	A	GSRRs	4-40
IVOR39	Guest processor doorbell critical		—	—	MSR[CE] and MSR[GS]	A	CSRRs	4-40
	Guest processor doorbell machine check		—	—	MSR[ME] and MSR[GS]	A	CSRRs	4-41
IVOR40	Hypervisor system call		—	—	—	SP*	SRRs	4-30

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
IVOR41	Hypervisor privilege		—	—	—	SP	SRRs	4-42
IVOR42	LRAT error		—	[ST], [FP,SPV], [EPID], [PT], [DATA]	—	SP	SRRs	4-45

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 4.9.6.1, "External proxy," describes how the e6500 core interacts with a programmable interrupt controller (PIC) defined by the integrated device.

4 PUO does not occur on the e6500 core.

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 (non-multiple) load or store aligned on an operand-sized boundary.

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. Inclusion of the addressing registers **r**A and possibly **r**B in the range to be loaded is a programming error, and the rules for partial execution do not protect these registers against overwriting.

Access control is not violated in any case.

As previously stated, elementary, aligned, and 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 loads/stores, plus the following:
  — Alignment
  — Data storage (if the access crosses a page boundary and protections on the two 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*. Each thread has a signal pin.

As defined by the architecture, CSRR0, CSRR1, and MSR are updated as shown in the following 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. • MSR[CM] is set to EPCR[ICM]. • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || 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 re-enabling 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 the same interrupt registers. The three different interrupts are as follows:

- Asynchronous machine check exceptions—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 in Section 4.9.3.4, "Asynchronous machine check exceptions." Each thread has a signal pin.
- Synchronous error report exceptions—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—non-maskable, non-recoverable interrupts that are signaled from the SoC (typically described in the integrated device reference manual as the *nmi* signal) as described by Section 4.9.3.2, "NMI exceptions." Each thread has a signal pin.

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/MCARUA), according to Table 4-4.
- The Machine-Check Syndrome (MCSR) register is used to log information about the error condition. The MCSR is described in Section 2.9.10, "Machine Check Syndrome (MCSR) register."
- At the end of the machine check interrupt software handler, a Return from Machine Check Interrupt (**rfmci**) instruction 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. Machine check exceptions 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. 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 MCSR by setting an error status bit corresponding to the error that was detected. If the error bit set in 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 the error causing the asynchronous machine check interrupt is not associated with a specific thread, it is posted to both threads.

- If an instruction in a thread 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, then the instruction takes an error report machine check interrupt on that thread to prevent the erroneous data from propagating.
- It is possible that a single error within a thread can set both an asynchronous machine check error condition in the MCSR and associate an error report with the instruction that consumed data associated with the error. The asynchronous error bit will always be set. 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 reads 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 causes the error report bits to be set and the processor 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 thread is the consumer of some data for which the error was detected by some agent other than the thread. For example, an error in DRAM may occur and, if the thread executed a load instruction that accessed that DRAM where the error occurred, the load instruction takes 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 an e6500 thread. MCSR[NMI] 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 that the taking of an asynchronous machine check interrupt always occurs when any of the asynchronous machine check error bits are 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]. 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.

When a machine check interrupt is taken, registers are updated as shown in the following table.

Table 4-4. Machine check interrupt settings

Register	Setting
MCSRR0	The thread 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	• MSR[CM] is set to EPCR[ICM]. • RI is cleared. • All other MSR bits are cleared.
MCAR (MCARU)	MCAR is updated with the address of the data associated with the machine check. See Section 2.9.9, "Machine-check address registers (MCAR/MCARU/MCARUA)."
MCSR	Set according to the machine check condition. See Table 2-14.

Instruction execution resumes at address IVPR[0–47] || IVOR1[48–59] || 0b0000.

NOTES

For implementations on which a machine check interrupt is caused by referring to an invalid physical address, executing **dcbz**, **dcbzl**, **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, an interrupt can occur 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**.

The e6500 machine check exception sources are specified in the following 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 of a thread. 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 or tag parity error due to a load or store	L1CSR0[DCPE] and L1CSR0[DCE]
L2 MMU multi-way hit	HID0[EN_L2MMU_MHD]
LRAT multi-way hit. An LRAT multi-way hit can occur due to a page table translation while in guest mode or from a **tlbwe** in guest mode and multiple entries hit during LRAT translation.	HID0[EN_L2MMU_MHD]

Table 4-5. Machine check exception sources (continued)

Source	Additional Enable Bits[1]
Non-maskable interrupt	None
Self-test error. In the e6500 core, this condition is set due to a power-on reset self-test failure. If the self-test failure is related to a resource that is shared by both threads in the core (for example, an L2MMU array), the exception is posted to thread 0 because only that thread is enabled. If power-on reset self-test is enabled, then software should read MCSR to determine success or failure of the self-test. If software does not clear STE before enabling machine checks (by setting MSR[ME] or MSR[GS]), a machine check interrupt occurs due to the setting of MSR[ME] or MSR[GS].	None

[1] "Additional Enable Bits" indicates any other state that, if not enabled, inhibits the recognition of this particular error condition.

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 an e6500 thread. 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 because 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 in 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 to the associated thread, and the thread's MCSRR0 holds the address of the load instruction with which the parity error is associated. For more details about 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 thread's MCSR indicates the operation that incurred the error as follows:

- 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.

- 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. Because 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 that are considered load instructions with respect to permission checking and debug events are reported as store error reports (MCSR[ST] is set). See Section 2.9.10, "Machine Check Syndrome (MCSR) register," for which instructions set MCSR[LD] or MCSR[ST].
- **tlbwe** instruction error report. An error occurred performing logical-to-real address translation in the LRAT while attempting to execute the **tlbwe** instruction corresponding to the address contained in MCSRR0. Note that no synchronous machine check error report bits are set as the result of this error; however, the asynchronous bit MCSR[LRAT_MHIT] always appears as set if the error report occurs.

Table 4-6 describes which error sources generate which error report status bits in MCSR.

Note that there is no MCSR error status bit for CoreNet data errors that are forwarded to a thread as an RLnk error. If an RLnk 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 thread does not generate an asynchronous machine check. The device that detects the error is expected to report it. For example, assume that a thread attempts to perform a load from a PCI device that encounters an error. The PCI device signals a "PCI Master Abort" and signals the error to the programmable interrupt controller (PIC).

The thread's memory transaction should be completed with a data error so that the thread is not hung awaiting the transaction. Eventually, the PIC should interrupt the thread. (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 gracefully from errors.

In a multicore system, the PIC is likely to steer all PCI error interrupts to one thread. For the PCI Master Abort example, assume that thread B performs a load that gets a PCI Master Abort, and the PIC steers the PCI's error signal to thread (processor) A's machine check input signal. Here, the error report in thread B prevents the propagation of bad data; thread A gets the task of attempting a graceful recovery. Some interprocessor communication is likely necessary.

Table 4-6. 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, TLB0 parity error, or LRAT multi-way hit		
	CoreNet bad data / RLnk error		

Table 4-6. Synchronous machine check error reports (continued)

Synchronous Machine Check Source	Error Type	MCSR Update[1]	Precise[2]
Load (or touch) instruction	Data cache data array parity error	LD, [LDG][4]	Yes
	Data cache tag parity error		
	Data cache tag multi-way hit		
	L2MMU multi-way hit, TLB0 parity error, or LRAT multi-way hit		
	CoreNet Bad Data / RLnk error (on load data or read of PTE during page table translation)		
Store or cache operation instruction	L2MMU multi-way hit, TLB0 parity error, or LRAT multi-way hit	ST	Yes
	CoreNet Bad Data / RLnk error (on read of PTE during page table translation)		
	Data cache tag multi-way hit		
tlbwe instruction	LRAT multi-way hit	—	Yes

[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 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 will point to the oldest instruction from that fetch group.

[4] LDG is set if the load was a guarded load (WIMGE = xxx1x).

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 4-7. 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		None
	NMI pin	N/A	NMI		None
Self-test	Self test error.	N/A	STE		None
Instruction cache	Data array parity error	Instruction fetch	MAV	ICPERR	EA
	Tag array parity error				RA
Data cache	Data array parity error	Load Snoop [4]	MAV	DCPERR	RA
	Tag array parity error	Load, store, touch, or cache operation Snoop [4]			
LRAT	Multi-way hit	**tlbwe** or page table translation	MAV	LRAT_MHIT	LA[4]
L2 MMU	Multi-way hit	**tlbsx**, **tlbre**, instruction fetch, load, touch, store, cache op (all types)	MAV	L2MMU_MHIT	EA[5]
	TLB0 parity	**tlbsx**, **tlbre**, instruction fetch, load, touch, store, cache op (all types), **tlbivax** snoop	MAV	TLBPERROR	EA[5]

[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, logical, or effective address (RA, LA, or EA) or no address associated with the error.

[3] The machine check input pin is used by the SoC to indicate all types of machine check type errors that are detected by the SoC. Software must query error logging information within the SoC to determine the specific error condition and source.

[4] LA is the logical address (guest space RA) to be translated into a real (physical) address.

[5] The lower 12 bits of the EA may be 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:

- The exception is not a virtualization fault (TLB[VF] = 0).
- The exception occurs in 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 4-8. Data storage interrupt

IVOR	Interrupt	Exception	Directing State at Exception	(G)ESR[1]	Save/Restore Registers
IVOR2	Data storage (DSI)	Virtualization fault	TLB[VF] = 1 (from direct or indirect matching TLB entry)	[ST], [FP,SPV], [EPID], [PT]	SRRs
		Read access	MSR[GS] = 0 or EPCR[DSIGS] = 0	[FP,SPV], [EPID], [PT]	
		Write access		ST, [FP,SPV], [EPID], [PT]	
		Page table fault		[ST], [FP,SPV], [EPID], PT	
		Load reserve or store conditional to write-through required location (W = 1)		[ST]	
		Cache locking		[DLK,ILK],[ST]	
		Byte ordering		[ST],[FP,SPV],BO, [EPID]	
GIVOR2	Guest data storage (DSI)	Read access	MSR[GS] = 1 and EPCR[DSIGS] = 1	[FP,SPV], [EPID], [PT]	GSRRs
		Write access		ST, [FP,SPV], [EPID], [PT]	
		Page table fault		[ST], [FP,SPV],[EPID], PT	
		Load reserve or store conditional to write-through required location (W = 1)		[ST]	

1 In general, when an interrupt affects a (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 e6500-specific behavior.

Table 4-9. Data storage interrupt exception conditions

Exception	Cause
Virtualization fault	Loads and stores translated by TLB entries with TLB[VF] = 1 or an indirect entry during a page table translation always take a data storage interrupt directed to the hypervisor state.
Page table fault	A page table translation occurs on a load, store, or cache management instruction, and the resulting PTE is not valid (PTE[V] = 0).

Table 4-9. Data storage interrupt exception conditions (continued)

Exception	Cause
Read access control exception	Occurs when one 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). • During page table translation, the following conditions exist (if these conditions exist, (G)ESR[PT] is set if no TLB entry was created from the page table translation): • MSR[PR] = 1 (user mode) • A load or load-class cache management instruction caused the page table translation • The resulting PTE is valid (PTE[V] = 1) • PTE[BAP_4] & PTE[R] = 0 (no read permission) • During page table translation, the following conditions exist (if these conditions exist, (G)ESR[PT] is set if no TLB entry was created from the page table translation): • MSR[PR] = 0 (supervisor mode) • A load or load-class cache management instruction caused the page table translation • The resulting PTE is valid (PTE[V] = 1) • PTE[BAP_5] & PTE[R] = 0 (no read permission)
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). • During page table translation, the following conditions exist (if these conditions exist, (G)ESR[PT] is set if no TLB entry was created from the page table translation): • MSR[PR] = 1 (user mode) • A store or store-class cache management instruction caused the page table translation • The resulting PTE is valid (PTE[V] = 1) • PTE[BAP_2] & PTE[R] & PTE[C] = 0 (no write permission) • During page table translation, the following conditions exist (if these conditions exist, (G)ESR[PT] is set if no TLB entry was created from the page table translation): • MSR[PR] = 0 (supervisor mode) • A store or store-class cache management instruction caused the page table translation • The resulting PTE is valid (PTE[V] = 1) • PTE[BAP_3] & PTE[R] & PTE[C] = 0 (no write permission)
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 e6500 core, 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**, **icblq.**, **dcbtls**, **dcbtstls**, **dcblq.**, or **dcblc** when (MSR[PR] = 1) and (MSR[UCLE] = 0). ESR is set as follows: • For **icbtls**, **icblq.**, and **icblc**, ESR[ILK] is set. • For **dcbtls**, **dcbtstls**, **dcblq.**, or **dcblc**, ESR[DLK] is set. The architecture refers to this as a cache-locking exception.
Storage synchronization exception	Occurs when a **lbarx**, **lharx**, **lwarx**, **ldarx**, **stbcx.**, **sthcx.**, **stwcx.**, or **stdcx.** instruction attempts to access a location marked write-through required. See "Atomic Update Primitives Using **lwarx** and **stwcx.**," in the "Instruction Model" chapter of *EREF*.

Regardless of the EA, **icbt**, **dcbt**, **dcbtst**, **dcba** and **dcbal** instructions cannot cause a data storage interrupt. Because many DSI exceptions are based on the EA, store conditional instructions may result in DSI exceptions regardless of whether the store was performed or not. See "Atomic Update Primitives Using **lwarx** and **stwcx.**," in the "Instruction Model" chapter of *EREF*.

NOTE

icbi, **icbt**, **icblc**, **icblq.**, and **icbtls** are treated as loads from the addressed byte with respect to translation and protection. All 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 thread suppresses execution of the instruction that caused it. Registers associated with the thread are updated as described in the following table:

Table 4-10. 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**, **dcblq.**, or **dcblc** is executed in user mode and MSR[UCLE] = 0 ILK Set when a DSI occurs because **icbtls**, **icblq.**,or **icblc** is executed in user mode and MSR[UCLE] = 0 BO Set if the instruction caused a byte-ordering exception [PT] Set during a page table translation if a read or write access control exception occurred and no TLB entry was created, or if a page table fault exception or a virtualization fault exception occurred (A page table fault occurs if the associated PTE[V] bit is 0.) 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. • CM is set to EPCR[ICM] if the interrupt is directed to hypervisor state. • CM is set to EPCR[GICM] if the interrupt is directed to guest state. • RI is not cleared. • All other 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[0–47] || (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 exception occurs in the guest state (MSR[GS] = 1).

- The interrupt is programmed to be directed to the guest state (EPCR[ISIGS] = 1) and an instruction virtualization fault does not occur.

If all the above conditions are met, the ISI is directed to the guest-supervisor state.

This table describes exception conditions.

Table 4-11. Instruction storage interrupt exception conditions

Exception	Cause
Execute access control exception	Occurs when one of the following conditions exists: • In user mode (MSR[PR] = 1), an instruction fetch attempts to access a memory location that is not user-mode execute enabled (page access control bit UX = 0). • In supervisor mode (MSR[PR] = 0), an instruction fetch attempts to access a memory location that is not supervisor-mode execute enabled (page access control bit SX = 0). • During page table translation, the following conditions exist (if these conditions exist (G)ESR[PT] is set) if no TLB entry was created from the page table translation): • MSR[PR] = 1 (user mode) • an instruction fetch caused the page table translation • the resulting PTE is valid (PTE[V] = 1) • PTE[BAP$_0$] & PTE[R] = 0 (no execute permission) • During page table translation, the following conditions exist (if these conditions exist (G)ESR[PT] is set) if no TLB entry was created from the page table translation): • MSR[PR] = 0 (supervisor mode) • a load or load-class cache management instruction caused the page table translation • the resulting PTE is valid (PTE[V] = 1) • PTE[BAP$_1$] & PTE[R] = 0 (no execute permission)
Page table fault	A page table translation occurs on an instruction fetch and the resulting PTE is not valid (PTE[V] = 0).
Instruction virtualization fault	A page table translation occurs on an instruction fetch and the matching indirect entry has TLB[VF]=1.

When an ISI occurs, the thread suppresses execution of the instruction causing the interrupt.

Registers associated with the thread are updated as shown in the following table.

Table 4-12. 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. • CM is set to EPCR[ICM] if the interrupt is directed to hypervisor state. • CM is set to EPCR[GICM] if the interrupt is directed to guest state. • RI is not cleared. • All other MSR bits are cleared.
(G)ESR	BO Set if the interrupt-causing instruction encountered a byte-ordering exception; otherwise, it is cleared. PT Set during a page table translation if an execute access control exception occurred and no TLB entry was created, if a page table fault exception occurred (if the associated PTE[V] = 0), or if an instruction virtualization fault occurred because the indirectly entry has TLB[VF] = 1. All other defined ESR bits are cleared.

Instruction execution resumes at address (G)IVPR[0–47] || (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 the external input interrupt is enabled. There is a signal pin for each thread. The interrupt is directed to the hypervisor unless the following conditions exist:

- The exception occurs in 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 MSR[EE], MSR[GS], and EPCR[EXTGS] 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.

In an integrated device, external interrupts are typically signaled to a specific thread in 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 designated thread of the core as quickly as possible.

The e6500 core provides two methods of receiving external input interrupts, which are controlled through a register field in the PIC:

- In one method, the legacy method, a thread of the core takes an external input interrupt when the *int* signal from the PIC is asserted and the external input interrupt is enabled in the thread. 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. 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 a thread in 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 asserts a vector, the interrupt occurs and the thread 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 4.9.6.1, "External proxy."

Registers of the interrupted thread are updated as shown in the following table.

Table 4-13. 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

Table 4-13. External Input Interrupt Register Settings (continued)

Register	Setting
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. • CM is set to EPCR[ICM] if the interrupt is directed to hypervisor state. • CM is set to EPCR[GICM] if the interrupt is directed to guest state. • RI is not cleared. • All other 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[0–47] || (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 re-enabling MSR[EE].

4.9.6.1 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), of the interrupted thread, as described in Section 2.9.6, “(Guest) External Proxy (EPR/GEPR) registers.”

Instead of just signaling *int*, the PIC also provides the specific vector for the interrupt. When the interrupt is enabled and the PIC asserts a vector, the interrupt occurs and the interrupted thread of the core communicates to the PIC that the interrupt has been taken and provides the vector from the PIC in the (G)EPR register of the thread, 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 thread, and the interrupt that is put *in-service* is determined by when the thread takes the interrupt based on the appropriate enabling conditions. From a system software perspective, the thread does not acknowledge the interrupt until the external input interrupt is taken.

Software in the external input interrupt handler then reads (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; 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 requires that the following sequence occurs:

```
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 e6500 core, these exceptions are as follows:

- The following accesses are not word aligned:
  - Floating-point loads and stores (including **lfddx** and **stfddx**)
  - Load multiple or store multiple instruction (**lmw** and **stmw**).
- A load and reserve or store conditional instruction that is not aligned to the data size of the instruction:
  - **lharx** or **sthcx.** which is not halfword aligned,
  - **lwarx** or **stwcx.** which is not word aligned,
  - **ldarx** or **stdcx.** which is not doubleword aligned.

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** or **dcbzl** is attempted to a page marked write-through or cache-inhibited.
- A **stvflx, stvflxl, stvfrx,** or **stvfrxl** of more than 8 bytes is attempted to a page marked write-through or cache-inhibited.

For other accesses, the e6500 core performs misaligned accesses in hardware within a single cycle if the misaligned operand lies within a double-word boundary. Accesses that cross a double-word 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 non-pipelined. 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 thread suppresses execution of the instruction causing the alignment interrupt. Registers of the interrupted thread are updated as shown in the following table.

Table 4-14. 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. • CM is set to EPCR[ICM]. • RI is not cleared. • All other 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 of the thread resumes at address IVPR[0–47] || 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 4-15. 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 FPSCR.	FP
Illegal instruction	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 the e6500 core, 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 • A **mtspr** or **mfspr** that specifies a privileged SPR • A **mtpmr** or **mfpmr** that specifies a privileged PMR	PPR

Table 4-15. Program interrupt exception conditions (continued)

Exception	Cause	ESR Bits Set
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), and (MSR[GS] | ~EPCR[DUVD])), then a debug interrupt is taken instead of the program interrupt.	PTR
Unimplemented operation	The e6500 core does not take unimplemented operation exceptions. All defined but unimplemented instructions take an illegal instruction exception.	—

Registers are updated as shown in the following table.

Table 4-16. 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 • CM is set to EPCR[ICM] • RI is not cleared • All other MSR bits are cleared.
ESR	FP Set if an enabled floating-point exception-type program interrupt; otherwise, it is cleared. PIL Set if an illegal instruction exception-type program interrupt; otherwise, it is cleared. PPR Set if a privileged instruction exception-type program interrupt; otherwise, it is cleared. PTR Set if a trap exception-type program interrupt; otherwise, it is cleared. All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || 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 the following table.

Table 4-17. 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. • CM is set to EPCR[ICM]. • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || 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 4-19.

The system call interrupt is directed to the hypervisor if executed in the hypervisor state (MSR[GS] = 0) and is directed to the guest supervisor if executed in the 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 Table 4-18.

This table describes which (G)IVOR is taken based on the setting of MSR[GS] and the value of the LEV operand.

Table 4-18. System call/hypervisor system call interrupt selection

LEV	MSR[GS]	Interrupt
> 1	—	Undefined
1	—	IVOR40
0	0	IVOR8
	1	GIVOR8

Table 4-19. 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. • MSR[CM] is set to EPCR[ICM] if the interrupt is directed to hypervisor state. • MSR[CM] is set to EPCR[GICM] if the interrupt is directed to guest state. • RI is not cleared. • All other MSR bits are cleared.

For a system call interrupt, instruction execution resumes at address (G)IVPR[0–47] || (G)IVOR8[48–59] || 0b0000.

For a hypervisor system call interrupt, instruction execution resumes at address IVPR[0–47] || IVOR40[48–59] || 0b0000.

Hypervisor system call interrupts are provided as a 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])). MSR[EE] also enables external input, processor doorbell, guest processor doorbell, and fixed-interval timer interrupts.

This table shows register updates.

Table 4-20. 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. • CM is set to EPCR[ICM]. • RI is not cleared. • All other MSR bits are cleared.
TSR	DIS is set.

Instruction execution resumes at address IVPR[0–47] || 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. Writing a 1 causes the bit to be cleared; writing 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 *EREF* describes the architecture definition of the fixed-interval timer.

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].

NOTE

MSR[EE] also enables external input, processor doorbell, guest processor doorbell, and decrementer interrupts.

Registers are updated as shown in the following table.

Table 4-21. 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. • CM is set to EPCR[ICM] • RI is not cleared. • All other MSR bits are cleared.
TSR	FIS is set.

Instruction execution resumes at address IVPR[0–47] || IVOR11[48–59] || 0b0000.

NOTE

To avoid redundant fixed-interval timer interrupts, before re-enabling 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 *EREF* describes the architecture definition of the watchdog timer.

NOTE

MSR[CE] also enables the critical input interrupt.

Registers are updated as shown in the following table.

Table 4-22. 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. • CM is set to EPCR[ICM]. • RI is not cleared. • All other MSR bits are cleared.
TSR	WIS is set.

Instruction execution resumes at address IVPR[0–47] || IVOR12[48–59] || 0b0000.

NOTE

To avoid redundant watchdog timer interrupts, before re-enabling 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 4-23 is presented to the interrupt mechanism. The interrupt is directed to the hypervisor unless the following conditions exist:

- The exception occurs in 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 4-23. 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, and the resulting page table translation fails because the associated indirect entry does not match any valid TLB entry. (That means there is no associated indirect TLB entry in the TLB.)

When the interrupt occurs, the thread suppresses execution of the excepting instruction. Registers are updated as shown in the following table.

Table 4-24. 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. • CM is set to EPCR[ICM] if the interrupt is directed to the hypervisor state. • CM is set to EPCR[GICM] if the interrupt is directed to the guest state. • RI is not cleared. • All other 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.

Table 4-24. Data TLB error interrupt register settings (continued)

Register	Setting
(G)ESR	[ST] Set if the instruction causing the interrupt is a store, **dcbi**, **dcbz**, or **dcbzl**; otherwise, it is 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. [SPV] Set if the instruction causing the interrupt is an AltiVec load or store.
MAS*n*	If EPCR[DMIUH] = 1 and an instruction TLB error, data TLB error, instruction storage, or data storage interrupt is directed to the hypervisor, MAS registers are not changed. See Table 6-11

Instruction execution resumes at address (G)IVPR[0–47] || (G)IVOR13[48–59] || 0b0000.

Implementation notes:

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, the e6500 core 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 4-25 is presented to the interrupt mechanism. The interrupt is directed to the hypervisor unless the following conditions exist:

- The exception occurs in 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 4-25. 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 and the resulting page table translation fails because the associated indirect entry does not match any valid TLB entry. (That means there is no associated indirect TLB entry in the TLB.)

When an instruction TLB error interrupt occurs, the processor suppresses execution of the instruction causing the exception.

Registers are updated as shown in the following table.

Table 4-26. Instruction 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

Table 4-26. Instruction TLB error interrupt register settings (continued)

Register	Setting
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. • CM is set to EPCR[ICM] if the interrupt is directed to hypervisor state. • CM is set to EPCR[GICM] if the interrupt is directed to guest state. • RI is not cleared. • All other MSR bits are cleared.
MAS*n*	If EPCR[DMIUH] = 1 and an instruction TLB error, data TLB error, instruction storage, or data storage interrupt is directed to the hypervisor, MAS registers are not changed. See Table 6-11.

Instruction execution resumes at address (G)IVPR[0–47] || (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.

The e6500 core does not support imprecise debug events and DBSR bits are not set while MSR[DE] = 0.

The “Debug Support,” chapter of *EREF* describes such architectural aspects of the debug interrupt.

Registers are updated as shown in the following table.

Table 4-27. 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

Table 4-27. Debug interrupt register settings (continued)

Register	Description
MSR	• ME, is unchanged. • CM is set to EPCR[ICM] • RI is not cleared. • All other MSR bits are cleared.
DBSR	Set to indicate type of debug event. See Section 2.14.9, "Debug Status (DBSR/DBSRWR) register"

Note that on the e6500 core, 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 e6500 core 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[0–47] || IVOR15[48–59] || 0b0000.

4.9.16.1 Suppressing debug events in hypervisor mode

Synchronous debug events can be suppressed when executing in the hypervisor state. This prevents debug events from being recorded (and subsequent debug interrupts from occurring) when executing in the 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.

4.9.17 AltiVec unavailable interrupt—IVOR32

An AltiVec unavailable interrupt occurs when no higher priority interrupt exists, an attempt is made to execute an AltiVec instruction (including AltiVec load, store, and move instructions), and the AltiVec available bit in the MSR is disabled (MSR[SPV] = 0). SRR0, SRR1, and MSR are updated as shown in the following table.

Table 4-28. AltiVec Unavailable Interrupt Register Settings

Register	Description
SRR0	Set to the EA of the instruction causing the AltiVec unavailable interrupt
SRR1	Set to the MSR contents at the time of the interrupt
MSR	• ME, CE, and DE are unchanged. • CM is set to EPCR[ICM]. • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || IVOR32[48–59] || 0b0000.

When MSR[SPV] = 0 and an AltiVec unavailable interrupt occurs, software should determine the state of the AltiVec device in CDCSR0 and bring the device up to a ready state if it wishes to set SPV and re-execute the instruction that caused the interrupt.

4.9.18 AltiVec assist interrupt—IVOR33

The AltiVec assist interrupt occurs when no higher priority exception exists and an AltiVec assist exception is presented to the interrupt mechanism due to a denormalized floating-point number used as an operand to an AltiVec floating-point instruction requiring software assist. The instruction handler is required to emulate the interrupt causing instruction to provide correct results with the denormalized input. SRR0, SRR1, and MSR are updated as shown in the following table.

Table 4-29. AltiVec assist interrupt register settings

Register	Description
SRR0	Set to the EA of the instruction causing the AltiVec assist interrupt
SRR1	Set to the MSR contents at the time of the interrupt
MSR	• ME, CE, and DE are unchanged. • CM is set to EPCR[ICM]. • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || IVOR33[48–59] || 0b0000.

4.9.19 Performance monitor interrupt—IVOR35/GIVOR35

A performance monitor interrupt is implemented as defined in *EREF*. Conditions that can be programmed to trigger an interrupt on an e6500 thread are described in Section 9.12, "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 for a given thread, PMGC0[PMIE] must be set. A PMC*n* register overflow condition occurs with the following settings:

- PMLCa*n*[CE] = 1—For the given counter the overflow condition is enabled.
- PMC*n*[OV] = 1—The given counter indicates an overflow.

Performance monitor counters can be frozen on a triggering-enabled condition or event if PMGC0[FCECE] = 1.

Performance monitor interrupts are directed to the guest-supervisor state if EPCR[PMGS] = 1; otherwise, they are directed to the hypervisor state. When the performance monitor interrupt is directed to the guest-supervisor state, it is masked from being taken if MSR[GS] = 0 or MSR[EE] = 0. When the performance monitor interrupt is directed to the hypervisor state, it is masked from being taken if MSR[GS] = 0 and MSR[EE] = 0.

Although the interrupt condition could occur when the performance monitor interrupt is masked, the interrupt cannot be taken until the masking condition is changed. If a counter overflows while PMGC0[FCECE] = 0, PMLCa*n*[CE] = 1, and the interrupt is masked, the counter can wrap around to all zeros again without the interrupt being taken.

The registers of a thread (processor) are updated as shown in the following table.

Table 4-30. Performance monitor interrupt register settings

Register	Setting
(G)SRR0	Set to the EA 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. • CM is set to EPCR[ICM] if the interrupt is directed to the hypervisor state. • CM is set to EPCR[GICM] if the interrupt is directed to the guest state. • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address (G)IVPR[0–47] || (G)IVOR35[48–59] || 0b0000.

4.9.20 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 or thread 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 thread. *EREF* defines how threads send messages and the actions that threads take upon receipt of a message. Actions taken by devices other than processors are not defined.

msgsnd and **msgclr** instructions are provided for sending messages to threads and clearing received and accepted messages. These instructions are hypervisor privileged. See Section 3.4.12.5, "Message Clear and Message Send instructions."

The e6500 threads filter, accept, and handle the message types defined in Table 4-31. 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 4-31. Message types

Value	Description
0	Doorbell interrupt (DBELL). Causes a processor doorbell exception on a thread that receives and accepts the message.
1	Doorbell critical interrupt (DBELL_CRIT). Causes a processor doorbell critical exception on a thread that receives and accepts the message.
2	Guest processor doorbell interrupt (G_DBELL). Causes a guest processor doorbell exception on a thread that receives and accepts the message.

Table 4-31. Message types (continued)

Value	Description
3	Guest processor doorbell critical interrupt (G_DBELL_CRIT). Causes a guest processor doorbell critical exception on a thread 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 thread that receives and accepts the message.

No other message type is accepted on the e6500 core.

4.9.20.1 Doorbell interrupt definitions

The architecture defines the following doorbell interrupts, which are implemented on the e6500 core:

- 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)

4.9.20.1.1 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 thread.

Registers are updated as shown in the following table.

Table 4-32. 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. • CM is set to EPCR[ICM] • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || IVOR36[48–59] || 0b0000.

4.9.20.1.2 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 the following table.

Table 4-33. 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. • CM is set to EPCR[ICM] • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || IVOR37[48–59] || 0b0000.

4.9.20.1.3 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 thread.

Registers are updated as shown in the following table.

Table 4-34. Guest processor doorbell interrupt register settings

Register	Setting
GSRR0	Set to the EA of the next instruction to be executed
GSRR1	Set to the MSR contents at the time of the interrupt
MSR	• ME, CE, and DE are unchanged. • CM is set to EPCR[ICM] • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || IVOR38[48–59] || 0b0000.

NOTE

Although 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 the 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.

4.9.20.1.4 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 thread.

Thread registers are updated as shown in the following table.

Table 4-35. 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. • CM is set to EPCR[ICM]. • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || 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.

4.9.20.1.5 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 thread (processor).

Thread (processor) registers are updated as shown in the following table.

Table 4-36. 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. • CM is set to EPCR[ICM]. • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || 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.

4.9.21 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 thread. See Section 3.4.6.7, "Hypervisor privilege instruction."

Thread registers are updated as shown in the following table.

Table 4-37. 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. • CM is set to EPCR[ICM]. • RI is not cleared. • All other MSR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || 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.

This table lists the resources that cause a hypervisor privilege exception when accessed in the guest-supervisor state.

Table 4-38. Hypervisor privilege exceptions from the 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	—

Table 4-38. Hypervisor privilege exceptions from the guest-supervisor state (continued)

Resource	Hypervisor Privilege on Read	Hypervisor Privilege on Write	Hypervisor Privilege on Execute	Notes
tlbwe	—	—	Yes or No	Hypervisor privilege occurs only when EPCR[DGTMI] = 1 or an attempt to write TLB1.
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	—	Illegal instruction occurs on attempted read.
DEAR	No	No	—	Guest supervisor state access to DEAR maps to GDEAR.
DEC	Yes	Yes	—	—
DECAR	—	Yes	—	Illegal instruction occurs on attempted read.
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.
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	—	—
LPER	Yes	Yes	—	—
LPERU	Yes	Yes	—	—
LPIDR	Yes	Yes	—	—
LRATCFG	Yes	—	—	Illegal instruction occurs on attempted write.
LRATPS	Yes	—	—	Illegal instruction occurs on attempted write.
MAS5	Yes	Yes	—	—

Table 4-38. Hypervisor privilege exceptions from the guest-supervisor state (continued)

Resource	Hypervisor Privilege on Read	Hypervisor Privilege on Write	Hypervisor Privilege on Execute	Notes
MAS8	Yes	Yes	—	—
MCAR	Yes	Yes	—	—
MCARU MCARUA	Yes	Yes	—	—
MCSR	Yes	Yes	—	—
MCSRR*n*	Yes	Yes	—	—
MMUCFG	Yes	—	—	Illegal instruction occurs on attempted write.
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.
PWRMGTCR0	Yes	Yes	—	—
SCCSRBAR	Yes	—	—	Illegal instruction occurs on attempted write
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	—	—	Illegal instruction occurs on attempted write.
TBL(W)	—	Yes	—	Illegal instruction occurs on attempted read.
TBU(R)	No	—	—	Illegal instruction occurs on attempted write.
TBU(W)	—	Yes	—	Illegal instruction occurs on attempted read.
TCR	Yes	Yes	—	—
TLB0CFG	Yes	—	—	Illegal instruction occurs on attempted write.
TLB1CFG	Yes	—	—	Illegal instruction occurs on attempted write.
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/no[2]	—	—
PMLCA*n*	Yes/no[2]	Yes/no[2]	—	—

Table 4-38. Hypervisor privilege exceptions from the guest-supervisor state (continued)

Resource	Hypervisor Privilege on Read	Hypervisor Privilege on Write	Hypervisor Privilege on Execute	Notes
PMLCB*n*	Yes/no[2]	Yes/no[2]	—	—
PMGC0	Yes/no[2]	Yes/no[2]	—	—

[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 in supervisor mode and are no-oped in user mode.

4.9.22 LRAT error interrupt—IVOR42

An LRAT error exception occurs when one of the following occurs:

- The processor executes a **tlbwe** instruction in the guest supervisor state, EPCR[DGTMI] = 0, MAS0[TLBSEL] = 0, and there is no matching translation in the LRAT.
- The processor executes a page table translation in the guest supervisor state and there is no matching translation in the LRAT corresponding to the PTE[ARPN] field.

LRAT error interrupts occur because the logical-to-real address translation cannot be performed in the LRAT.

Thread registers are updated as shown in the following table.

Table 4-39. LRAT error 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. • CM is set to EPCR[ICM]. • RI is not cleared. • All other MSR bits are cleared.
DEAR	If the LRAT error interrupt occurred for a page table translation, 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. If the LRAT error interrupt occurred as a result of a **tlbwe** instruction, then DEAR is undefined.
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. [SPV] Set if the instruction causing the interrupt is an AltiVec load or store. [EPID] Set if the instruction causing the interrupt is an external PID instruction and the translation of the operand address causes the interrupt. [DATA] Set if the instruction causing the interrupt is a load, store, or cache management instruction and the translation of the operand address causes the interrupt. [PT] Set if the interrupt is the result of a page table translation. All other defined ESR bits are cleared.
LPER	Set to the values of the lower 28 bits of PTE[ARPN], PTE[WIMGE], and PTE[PS] if the LRAT error interrupt was a result of a page table translation. If the interrupt was a result of a **tlbwe** instruction, LPER is set to 0.

Instruction execution resumes at address IVPR[0–47] || IVOR42[48–59] || 0b0000.

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 an 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 4-40. Operations to avoid before the save/restore registers are saved to memory

Operation	Reason
Re-enabling MSR[EE] , MSR[CE], MSR[DE], or MSR[ME] in interrupt handlers	This prevents any asynchronous interrupts, as well as any debug interrupts (in the case of MSR[DE]), including synchronous and asynchronous types.
Branching (or sequential execution) to addresses not mapped by the TLB, mapped without SX set, or causing large address or instruction address overflow exceptions	This prevents instruction storage, instruction TLB error, and instruction address overflow interrupts.
Load, store, or cache management instructions to addresses not mapped or without permissions	This prevents data storage and data TLB error interrupts.
Execution of System Call (**sc**), trap (**tw**, **twi**, **td**, **tdi**), or **ehpriv** instructions	This prevents system call and trap exception-type program interrupts. Note that **ehpriv** instructions can be executed in the guest-supervisor state.
Re-enabling of MSR[PR]	Prevents privileged instruction exception-type program interrupts. Alternatively, software could re-enable 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 of the associated thread are required to report any synchronous imprecise interrupt-generating exceptions. 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 only one of the synchronous interrupt types exists at any given time. The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

This section does not define the permitted setting of multiple exceptions for which the corresponding interrupt types are disabled. The generation of exceptions for which the corresponding interrupt types are disabled has no effect on the generation of other exceptions for which the corresponding interrupt types are enabled. Conversely, if a particular exception for which the corresponding interrupt type is enabled is shown in the following sections to be of a higher priority than another exception, it 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 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 4-41 and Table 4-42 describes the relative priority of each exception type. Exception priority is listed from highest to lowest, and the lower the numerical relative priorities shown imply 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 4-41. Asynchronous exception priorities

Relative Priority	Exception	Interrupt Level[1]	Interrupt Nature	Pre- or Post-Completion[2]	Comments
0	Machine Check	Machine Check	Async	N/A	Asynchronous exceptions may come from the processor or from an external source.
1	Guest Processor Doorbell Machine Check	Critical	Async	N/A	—
2	Debug - UDE	Debug	Async	N/A	Debug-UDE is often used for an externally generated high priority attention signal.
	Debug - Interrupt Taken	Debug	Async	N/A	Debug interrupt taken after original interrupt has changed NIA and MSR.
	Debug - Critical Interrupt Taken	Debug	Async	N/A	Debug interrupt taken after original critical interrupt has changed NIA and MSR.
3	Critical Input	Critical	Async	N/A	—
4	Watchdog	Critical	Async	N/A	—
5	Processor Doorbell Critical	Critical	Async	N/A	—
6	Guest Processor Doorbell Critical	Critical	Async	N/A	—
7	External Input	Base	Async	N/A	—
14	Program - Delayed Floating Point Enabled	Base	Async	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	Async	N/A	—
23	Decrementer	Base	Async	N/A	—
24	Processor Doorbell	Base	Async	N/A	—
25	Guest Processor Doorbell	Base	Async	N/A	—
26	Performance Monitor	Base	Async	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 4-42. Synchronous exception priorities

Relative Priority	Exception	Interrupt Level[1]	Interrupt Nature	Pre or Post Completion[2]	Comments
0	Error Report	Machine Check	Sync	Pre	—
8	Debug - Instruction Address Compare	Debug	Sync	Pre	—
9	ITLB	Base	Sync	Pre	—
9.5	ISI	Base	Sync	Pre	—
10	LRAT error on instruction fetch	Base	Sync	Pre	—
11	Program - Privileged Instruction	Base	Sync	Pre	—
	Embedded Hypervisor Privilege	Base	Sync	Pre	—
12	FP Unavailable	Base	Sync	Pre	—
	AltiVec Unavailable	Base	Sync	Pre	—
13	Debug - Trap	Debug	Sync	Pre	—
14	Program - Illegal Instruction	Base	Sync	Pre	—
	Program - Unimplemented Operation	Base	Sync	Pre	—
	Program - Trap	Base	Sync	Pre	—
	Program - Floating Point Enabled	Base	Sync	Pre	—
15	DTLB	Base	Sync	Pre	—
15.5	DSI	Base	Sync	Pre	A DSI Virtualization Fault always takes priority over all other causes of DSI.
16	Alignment	Base	Sync	Pre	—
17	LRAT error on data access or **tlbwe**	Base	Sync	Pre	—
18	System Call	Base	Sync	Post	System Call Interrupt has SRR0 pointing to instruction after sc (that is, post completion).
	Embedded Hypervisor System Call	Base	Sync	Post	Embedded Hypervisor System Call Interrupt has SRR0 pointing to instruction after sc (that is, post completion).
	AltiVec Assist	Base	Sync	Post	—

Table 4-42. Synchronous exception priorities

Relative Priority	Exception	Interrupt Level[1]	Interrupt Nature	Pre or Post Completion[2]	Comments
19	Debug - Return from Interrupt	Debug	Sync	Pre	—
	Debug - Return from Critical Interrupt	Debug	Sync	Pre	—
	Debug - Branch Taken	Debug	Sync	Pre	—
20	Debug - Data Address Compare	Debug	Sync	Pre	—
21	Debug - Instruction Complete	Debug	Sync	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.

Chapter 5
Core Caches and Memory Subsystem

This chapter describes the caches and cache structures that are local to the e6500 core, as well as the e6500's memory subsystem (MSS), which encompasses the L1 caches, the dual Load/Store Units (LSU), the Instruction Unit (also called the Fetch Unit), the cluster L2 cache, and the cluster CoreNet interface (commonly called a *Bus Interface Unit* or BIU).

The e6500 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.

The dual LSU, one per thread, manage 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.

Clusters of cores share a 2048 KB, four-bank, 16-way set associative shared L2 cache. In addition, there is also support for a platform cache implemented by the integrated device.

The BIU is the interface from the cluster 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. The size of a cache block is determined by the implementation and, on the e6500 core, a cache block or line is 16 words.

5.1 Overview

This section lists features of the dual LSU, the Instruction Unit (also called the Fetch Unit), the L1 cache, the cluster shared L2 cache, and cluster CoreNet interface.

The dual LSU has the following features:

- Dual Load and Store pipelines, one per thread, accessing a shared Level 1 Cache

- Accesses to system memory are performed (critical data first). For data accesses, the LSU receives the critical data 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. 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 cluster BIU and into the memory subsystem.
- L1 load miss queueing. On a load miss, an entry in the load miss queue is allocated and then a bus transaction is queued to read the line. Load hits and load misses continue to be processed until there are more than eight outstanding load misses and L1 stashes.
- Store gathering. When a caching-allowed store misses in the data cache, the store data is written to a cache line–wide store gather buffer. Individual store requests from the store queue are gathered if they are to the same cache line and meet the conditions for store gathering. Store gather buffer entries are sent to the L2 cache if the store gather buffer entry is complete or if other conditions for advancing a store gather buffer entry are met.
- Data reload buffering contains all cache line reloads to the L1 cache.

The L1 cache implementation has the following features:

- Separate 32 KB instruction and data caches (Harvard architecture)
- Eight-way set-associative, non-blocking caches
- Physically addressed cache directories. The physical (real) address tag is stored in the cache directory.
- Two-cycle access time provides three-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.
- Both L1 caches support parity generation and checking (enabled through L1CSR0 and L1CSR1 bits), as follows:
  — Instruction cache: one parity bit per word of instruction, one bit of parity per tag
  — Data cache: one parity bit per byte of data, one bit of parity per tag

  See Section 5.4.4, “L1 cache parity.”
- Both caches also support parity error injection, which provides a way to test error recovery software by intentionally injecting parity errors into the instruction and data caches. See Section 5.4.5, “L1 cache parity error injection.”
- The data cache and instruction cache do not contain modified data. See Section 5.5.1, “Data cache coherency model.”
- The L1 instruction cache supports automatic error correction by invalidation when an access detects an 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 an access detects an error. 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.
- 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."
- The L1 data cache uses a First-In-First-Out replacement algorithm. See Section 5.7.2.1, "FIFO replacement."
- The L1 instruction cache uses a Pseudo-Least-Recently-Used (PLRU) replacement algorithm. See Section 5.7.2.2, "PLRU replacement."
- Support for individual line locking with persistent locks. See Section 5.6.4, "Instruction and data cache line locking and unlocking."
- Support for cache stashing to the L1 data cache from other devices in the integrated device
- Both instruction and data cache lines are reloaded in a single-cycle, 64-byte write from a reload buffer as described in Section 5.3.1, "Dual Load/Store Unit (LSU)." Cache reloads write all 64 bytes at once and, therefore, do not occur until all data has been buffered from the CoreNet interface.

The cluster shared L2 cache has the following features:

- Dynamic Harvard architecture, merged instruction and data cache.
- 2048 KB array divided into four banks of 512 KB, organized as 512 sixteen-way sets of 64-byte cache lines
- 40-bit physical address
- Modified, exclusive, shared, invalid, incoherent, locked, and stale states
- Support for modified, exclusive, and shared intervention from the L2 cache
- Support for cache stashing to the L2 cache from other devices in the integrated device
- 16-way set associativity with Streaming Pseudo Least Recently Used with Aging (SPLRU with Aging) replacement. Additional support for Pseudo Least Recently Used (PLRU), Streaming Pseudo Least Recently Used (SPLRU), and First-In-First-Out (FIFO) replacement.
- Supports way partitioned cache operation. See Section 5.8.4.5, "L2 cache partitioning."
- 64-byte (16-word) cache-line, coherency-granule size.
- Support for individual line locking with persistent locks. See Section 5.8.4.4, "L2 cache line locking and unlocking."
- Inclusive for data lines and generally inclusive for instruction lines
- Reloaded whenever the L1 instruction cache makes a request, 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 have a copy of the line.
- Pipelined data array access with two-cycle repeat rate
- ECC protection for data, tag and status arrays
- ABIST support

The cluster BIU is the interface manager to the CoreNet interface and the rest of the system. The cluster BIU sends and receives transactions from the CoreNet interface and routes them to the appropriate core that requires them.

The cluster 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 e6500 core.

5.2.1 Cache identifiers

Instructions having a cache target (CT) 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 3.4.11.1.1, “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 cluster shared 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, because 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 initiates 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.

5.3 Core memory subsystem block diagram

The instruction and data caches are integrated with the Dual LSU, the instruction unit, and the eLink core/cluster interface is shown in the following table.

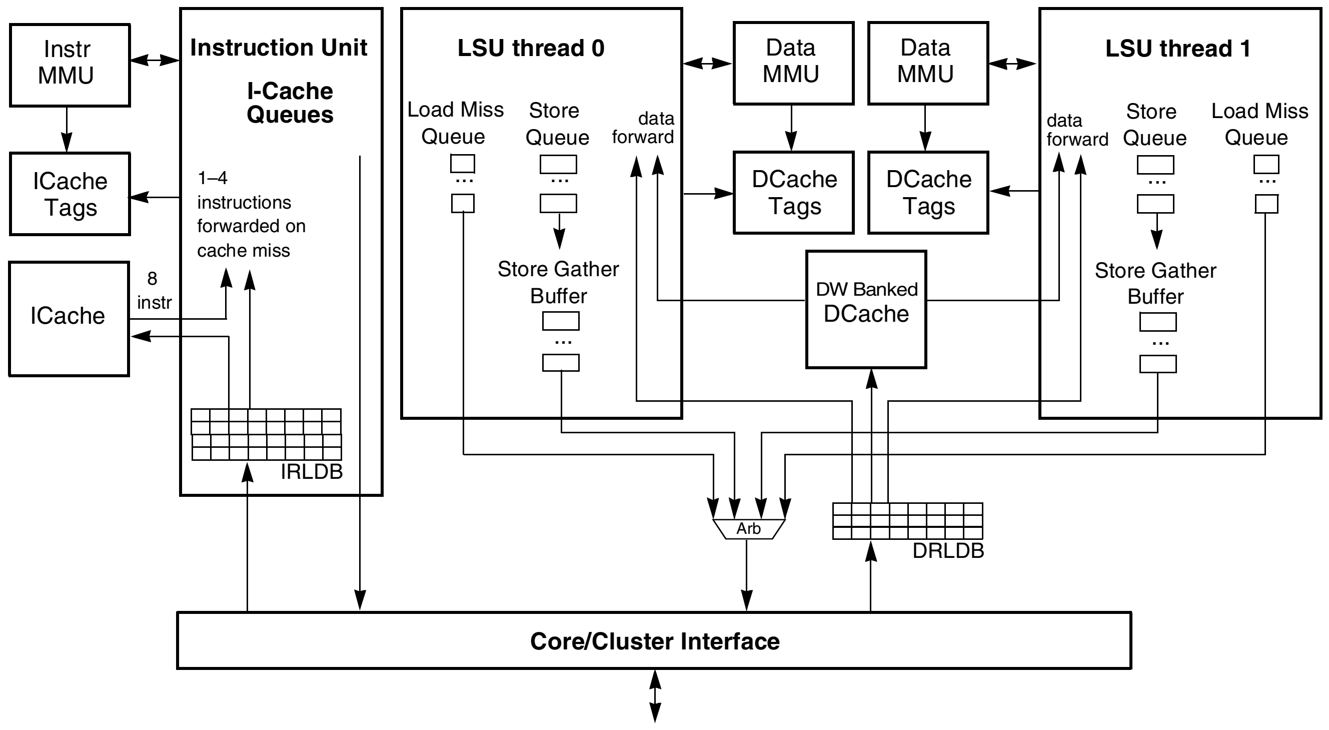


Figure 5-1. Core memory subsystem block diagram

The following sections briefly describe the Dual LSU, the instruction unit, and the core/cluster interface.

5.3.1 Dual Load/Store Unit (LSU)

The Dual LSU pipelines, one per thread, access a shared Level 1 Cache. Each thread-specific LSU executes integer, floating-point, and AltiVec load and store instructions for their respective threads and manages transactions between the caches and the register files (GPRs and FPRs). Each thread-specific LSU 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, doubleword, or quad-word basis. The data cache is provided with a 64-byte interface (the width of a cache block).

This section provides an overview of how each thread-specific LSU coordinates traffic in their 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 10.4.4, “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 thread-specific 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. Caching-inhibited loads that are not guarded are also executed 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 L1 Load Miss Queue (LMQ)

On a load miss, the thread-specific LSU allocates an LMQ entry and then queues a core/cluster interface transaction to read the line.

5.3.1.3 Store Queue

Stores cannot execute speculatively and remain in the thread-specific LSU Store Queue, shown in Figure 5-1, until completion logic indicates that the store is to be committed. The Store Queue arbitrates for access to the L1 data cache. When arbitration is successful, the data cache is written and the store is removed from the queue. If a store is caching-inhibited, the operation moves through the Store Queue onto the rest of the memory subsystem. Stores do not execute speculatively, but they can be performed out of order with respect to other loads and stores if they are not marked as guarded.

5.3.1.4 Store Gather Buffer

The thread-specific LSU Store Gather Buffer gathers individual store requests from the Store Queue into a single core/cluster interface request. This gather function is extremely important to improve store performance and power consumption of the L1 and L2 cache memory subsystem.

5.3.1.5 Data Reload Data Buffer (DRLDB)

The Data Reload Data Buffer (DRLDB) is a three-entry buffer that holds data used to reload the L1 data cache after a load miss. This structure is associated with the L1 data cache and is not replicated per thread.

5.3.2 Instruction Unit

The Instruction Unit (also called the Fetch Unit) interfaces with the L1 instruction cache and the core/cluster interface. As with the data caches, instructions that miss in the instruction cache are buffered as they are fetched into the four-entry Instruction Reload Data Buffer (IRLDB). After an entire line is available, it is written into the instruction cache.

5.3.3 Core/Cluster interface

The Core/Cluster interface is the interface between the core and the cluster shared L2.

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 organization

This figure shows the organization of the L1 data cache.

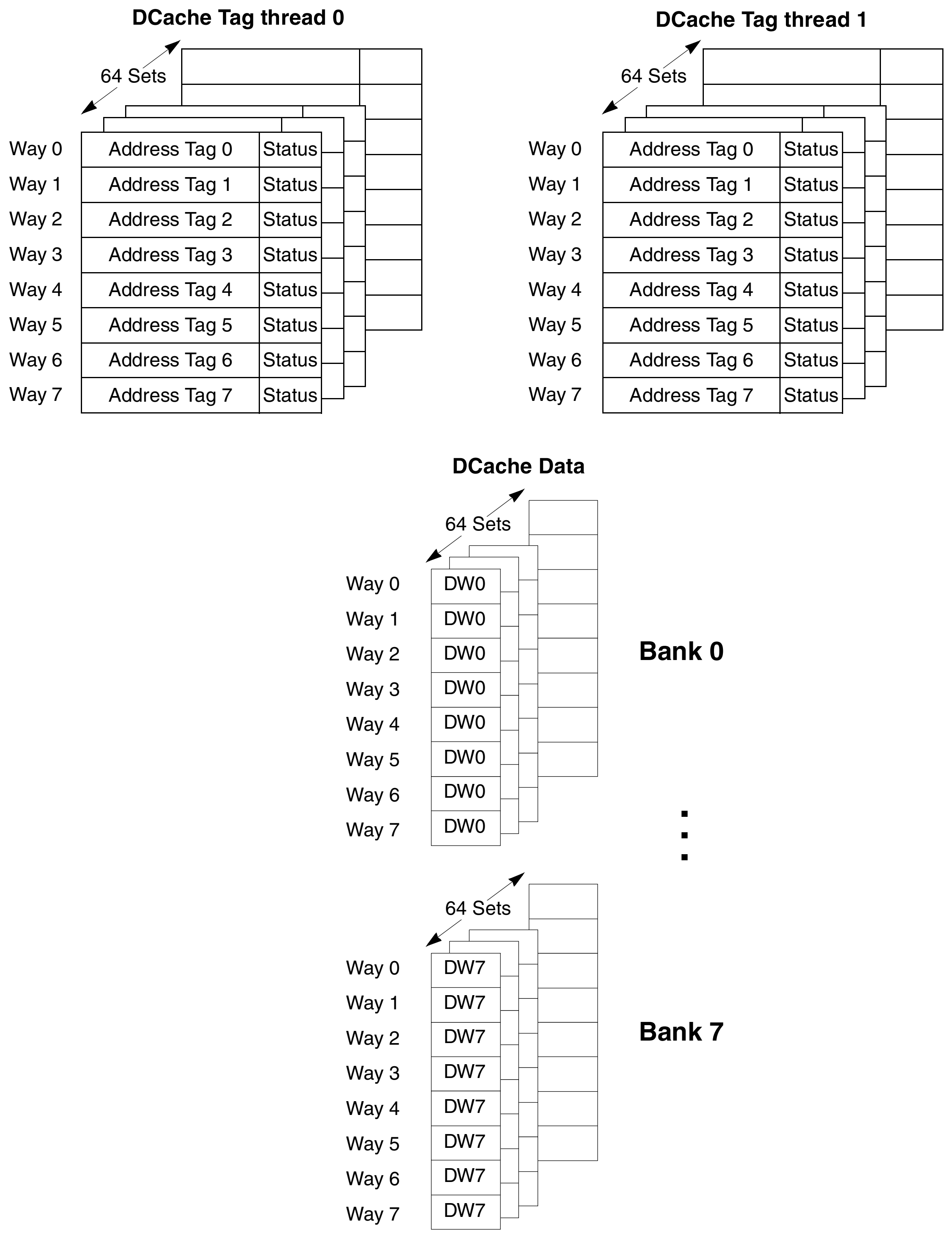


Figure 5-2. L1 data cache organization

The data cache has separate tag arrays per thread and a shared data array. The data array is banked by doubleword. The data array can be accessed simultaneously by both threads when the threads target different doublewords. If both threads target the same doubleword, a winner is determined through round-robin arbitration. Note that accesses requiring more than one doubleword, such as AltiVec instructions or full cache-line instructions such as **dcbzl,** create additional thread collision scenarios.

Each tag entry contains an address tag, status information, and lock information. Each data entry contains 64 bytes of data. Also, although it is not shown in Figure 5-2, the data cache has one parity bit/byte (eight parity bits/doubleword) and one parity bit/tag.

Each cache block is loaded from a 16-word boundary (that is, physical addresses bits 34–39 are zero). Cache blocks are also aligned on page boundaries. The tags consist of physical address bits PA[0:27]. Address translation occurs in parallel with set selection. Physical address bits PA[28:33] provide the index to select a cache set. Physical address bits PA[34:36] select the doubleword bank. Physical address bits PA[37:39] locate a byte within the selected doubleword.

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 data 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).

5.4.2 Write-through cache

The L1 data cache is a write-through cache and does not contain modified data.

If data or tags are corrupted in the L1 cache, the line can be invalidated and repopulated with valid data from the rest of the memory hierarchy.

5.4.3 L1 instruction cache organization

This figure shows the organization of the L1 instruction cache.

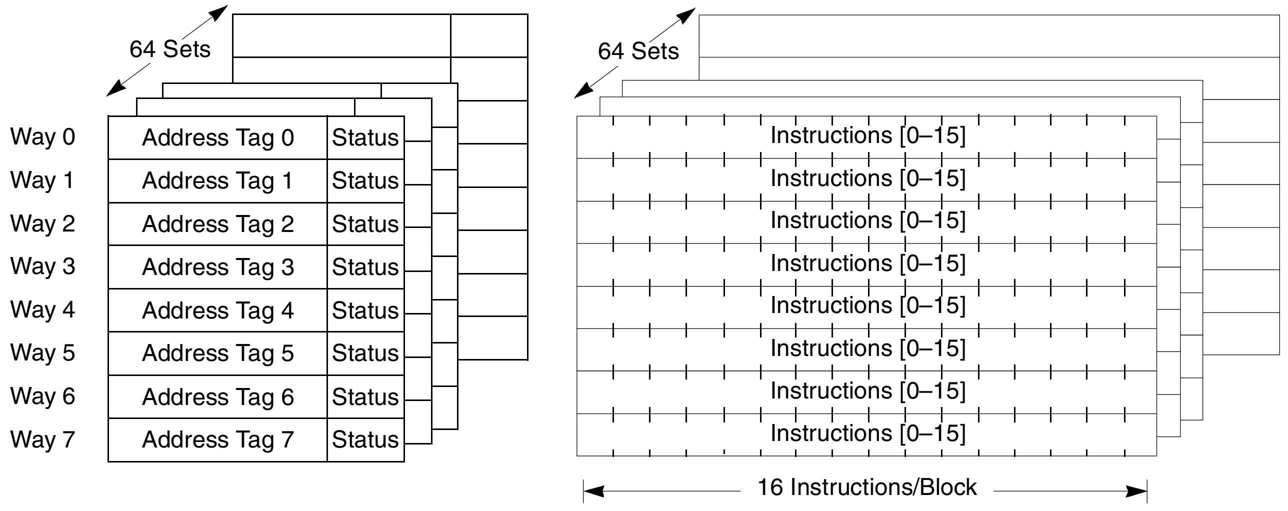


Figure 5-3. L1 instruction cache organization

The instruction cache has a shared tag array and a shared data array, which are accessed in round-robin fashion by each thread. Eight instructions are loaded from the instruction cache per access and up to four instructions are forwarded to the instruction unit for the winning thread. In the following cycle when the next thread is accessing the cache, up to four more instructions are forwarded to the instruction unit. This allows two threads to continually feed their instruction units.

Each tag entry contains an address tag, status information, and lock information. Each data entry contains 16 instructions. Also, although it is not shown in Figure 5-3, the instruction cache has one parity bit/word (eight 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 34–39 of the physical addresses are zero). Instruction cache blocks are also aligned on page boundaries. The tags consist of physical address bits PA[0:27]. Address translation occurs in parallel with set selection. Physical address bits PA[28:33] provide the index to select a set, and physical address bits PA[34:37] select an instruction within a block.

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 one cycle. However, 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 parity

The L1 caches are protected by parity. Parity information is written into the L1 caches whenever one of the following occurs:

- A store instruction (or **dcbz**, **dcbzep**, **dcbzl**, **dcbzlep**, **dcba**, or **dcbal**)
- A reload occurs into the instruction or data cache

L1 cache parity is checked whenever:

- A load instruction hits in the L1 data cache
- An instruction fetch hits in the L1 instruction cache

Also, the e6500 core implements a cache tag parity bit per entry/set. Cache tag parity is checked for all cache transactions, including snoops.

L1 cache parity checking is disabled by default and can be enabled by writing 1 to L1CSR0[DCPE] and to L1CSR1[ICECE].

If an instruction cache data or tag parity error is detected, the following occurs:

- The instruction cache is automatically flash invalidated.
- 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:

- The data cache is automatically flash invalidated.
- 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 L1 cache parity error injection

Cache parity error injection provides a way to test error recovery software by intentionally injecting parity errors into the instruction and data caches, as follows:

- If L1CSR1[ICEI] = 1, any instruction cache line fill has all instruction parity bits inverted in the instruction cache. The tag parity is not inverted.
- If L1CSR0[CPI] = 1, any data cache line fill has all data and tag parity bits inverted in the data cache. Additionally, inverted parity bits are generated for any bytes stored into the data cache by store instructions, **dcbz**, **dcbzl**, **dcba**, or **dcbal**.

NOTE

Parity checking for the L1 instruction cache must be enabled (L1CSR1[ICECE] = 1) when L1CSR1[ICEI] = 1. Similarly for the data cache, L1CSR0[DCPE] must be set if L1CSR0[DCPI] = 1. L1CSR0[DCPI] cannot be set (using **mtspr**) without setting L1CSR0[DCPE]. L1CSR1[ICEI] cannot be set without setting L1CSR1[ICECE].

If a cache parity error is detected, a machine check interrupt occurs. Sources for cache parity errors are described in Section 4.9.3, "Machine check interrupt—IVOR1."

5.5 L1 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 data cache supports only invalid and valid states.

The core provides full hardware support for cache coherency and ordering instructions and for TLB management instructions.

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 5.6.5, “L1 data cache flushing.” Also, the flushing requirement of modifying code from the data cache is described in *EREF*.

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 the 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 *EREF*.

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 in the L1 data cache updates the line and allocates into the Store Gather Buffer to access the shared L2 cache.

A write-back store that misses in the L1 data cache allocates into the Store Gather Buffer to access the shared L2 cache. The L1 data cache is not reloaded.

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 that hits in the L1 data cache updates the line and allocates into the Store Gather Buffer to access the shared L2 cache.

A write-through store the misses in the L1 data cache allocates into the Store Gather Buffer to access the shared L2 cache. The L1 data cache is not reloaded.

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 CoreNet as a single-beat read. If a caching-inhibited store hits in the cache, the cache may be altered but the store is performed on CoreNet as a single-beat write.

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 byte ordering (BO) bit in the Exception Syndrome (ESR) register 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. Software should assume that any page that is marked as execute-permitted will generate 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 addresses 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 *EREF*.

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*, and the implementation simplify this greatly. In practice, this requires that the programmer only 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 that may affect 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 this case.

5.5.5.1 Architecture ordering requirements

Power Architecture and *EREF* 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. In fact, the processor must ensure 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. Guarded caching-inhibited loads must also 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, if a load is followed by a store, then the load is always performed before the store. This is an *EREF* 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 five memory barriers that can be used on the e6500 core 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 weakest are as follows:

- **sync** (or **sync** 0 or **msync**)—**sync** creates a barrier such that *all* memory accesses that occur before the **sync** (regardless of WIMGE attributes) 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 is not an issue or if no other barrier orders the memory accesses.
- **mbar** (or **mbar** 0)—**mbar** creates the same barrier that **sync** creates; 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**.
- **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.

  lwsync is a better performing memory barrier than **sync**, **mbar**, or **mbar** 1.
- Elemental memory barriers (**sync** x,E) create targeted barriers when the storage locations accessed by the instructions are neither write-through required nor caching inhibited.
  - If E_0 = 1, then elemental barrier load with load.
  - If E_1 = 1, then elemental barrier load with store.
  - If E_2 = 1, then elemental barrier store with load.
  - If E_3 = 1, then elemental barrier store with store.

  Any combination of these memory barriers can be specified simultaneously by setting the appropriate bit in the E field. The memory barrier orders accesses described above to the local caches of the processor. This is the most efficient barrier for normal SMP programming when dealing with multiprocessor locks and critical regions.

Another method exists for ordering all caching-inhibited guarded loads and stores. The HID0[CIGLSO] bit can be set to force all caching-inhibited guarded loads and stores 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. This is likely to perform better than inserting **mbar** in specific places.

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 write HID0[CIGLSO] = 1 at boot time. This should order all such caching-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 or appropriate **esync** 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 **lbarx**, **lharx**, **lwarx**, **ldarx** and **stbcx.**, **sthcx.**, **stwcx.**, **stdcx.** should be used to ensure the lock is properly acquired.
  - Insert a **lwsync** barrier or appropriate **esync** 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

The following table displays the Power ISA and *EREF* 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. In this table, 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 will enforce all barriers.

Table 5-1. 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
Cacheable write-through	0b10xxx	**mbar** 1	**mbar** 0	**mbar** 0	Yes
Cacheable write-back	0b00xxx	**lwsync** **sync** x,E=xxx1	**lwsync** **sync** x,E=1xxx	**mbar** 0 **sync** x,E=xx1x	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 when a processor 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 do, which is 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 will guarantee 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 caching inhibited store.

5.5.5.5 Atomic memory references

The e6500 core implements **lbarx**, **lharx**, **lwarx**, **ldarx** and **stbcx.**, **sthcx.**, **stwcx.**, **stdcx.** as described in *EREF*.

The e6500 core takes a data storage interrupt if the location is write-through required but does not take the interrupt if the location is caching inhibited (that is, if caching-inhibited reservations are permitted). Software should avoid using reservations on storage that is caching inhibited because future cores may not support this.

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 e6500 core makes reservations on behalf of aligned 64-byte blocks of the memory address space.

If the reservation is 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. Those events are described in Section 3.4.10, "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 e6500 core implements the cache control instructions as described in Section 3.4.11.1, "User-level cache instructions," and Section 3.4.12.4.1, "Supervisor-level cache instruction." Note that on the e6500 core, 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 no-op.

If the EA cannot be translated, all cache control instructions generate TLB miss exceptions, except **dcba**, **dcbal**, **dcbt**, **dcbtep**, **icbt**, **dcbtst**, and **dcbtstep**, which are treated as no-ops 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 no-op.

5.6.2 Enabling and disabling the L1 caches

The instruction and data caches are enabled and disabled using 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] = 0 and L1CSR1[ICE] = 0).

Disabling the data cache has the following effects:

- Touch instructions (**dcbt**, **dcbtst**, **dcbtls**, **dcbtstls**, **dcblc**, **dcblq.**, **icbt**, **icbtls**, **icblc**, and **icblq.**) targeting a 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 invalidated to prevent coherency problems when it is enabled again.
- Data accesses bypass the data cache and are forwarded to the memory subsystem as caching-allowed. Returned data is forwarded to the requesting execution unit but is not loaded into the data cache.
- Other cache management instructions do not affect the disabled cache.

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. When the instructions are returned, they are forwarded to the instruction unit but are not loaded into the instruction cache.

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.

Note that enabling either L1 cache without first enabling the L2 cache is not supported.

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 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.

Software must set the CFI bits if invalidation is desired after a warm reset. 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 core that performs them, and other core's L1 caches are not affected. Software should always poll the CFI bits after setting them to determine when the invalidation is completed and then perform an **isync**.

Individual instruction or data cache blocks can be invalidated by using **icbi** and **dcbi**. Also note that, with **dcbi**, the e6500 core invalidates the cache block without pushing it out to memory. See Section 3.4.12.4.1, "Supervisor-level cache instruction." Because the instruction and data caches support persistent locks, invalidating the caches does not reset lock bits.

Exceptions and other events that can access the L1 cache should be disabled during this time so that the replacement algorithm can function undisturbed.

5.6.4 Instruction and data cache line locking and unlocking

User-mode instructions perform cache line locking and unlocking based on the complete address of the cache line. **dcbtls**, **dcbtstls**, and **dcblc** are used for data cache locking and unlocking, and **icbtls** and **icblc** are used for instruction cache locking and unlocking. In addition, the e6500 also provides **dcblq.** and **icblq.** to query the lock status. For descriptions, see Section 3.4.11.2, "Cache locking instructions."

The CT operand is used to indicate the cache target of the cache line locking instruction. See Section 3.4.11.1.1, "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 and 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.

This table 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 5-2. Cache locking based on MSR[GS,PR,UCLE] and MSRP[UCLEP]

MSR[GS]	MSR[PR]	MSR[UCLE]	MSRP[UCLEP]	Result
0	0	x	x	Execute
x	1	0	x	DSI, ESR[DLK or ILK] set
x	1	1	x	Execute
x	0	x	0	Execute
1	0	x	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 3.4.11.2, “Cache locking instructions” for a description of what conditions set these bits.

It is acceptable to lock all ways of a cache set. A non-locking reload for data to a new address in a completely locked cache set is dropped and not put into the cache.

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 an instruction causes an invalidate operation on a line. If a **dcbi**, **icbi**, **icbiep**, **dcbf**, **dcbfep**, **dcbst**, or **dcbstep** targets a locked line, the line is invalidated but the lock is persistent.

5.6.4.2 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 power-up. A subsequent reset operation does not clear the lock bits automatically. Software must use the CLFC controls if flash clearing of the lock bits is desired after a warm reset. Setting CLFC bits causes a flash lock clear performed in a single CPU cycle, after which the CLFC bits are automatically cleared (CLFC bits are not sticky).

5.6.5 L1 data cache flushing

Because the L1 Data Cache contains no modified data, no flush routine is required.

5.7 L1 cache operation

This section describes operations performed by the L1 instruction and data caches.

5.7.1 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.

5.7.1.1 Data cache reloads

The core data cache blocks are reloaded from an L2 cache or the memory subsystem when data load misses occur for caching-allowed accesses, as described in Section 5.3.1, "Dual Load/Store Unit (LSU)."

When the data cache is disabled (L1CSR0[CE] = 0), data accesses bypass the data cache and are forwarded to the memory subsystem as caching-allowed. Returned data is forwarded to the requesting execution unit but is not loaded into the data cache. 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.

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.

5.7.1.2 Instruction cache reloads

On an L1 instruction cache hit, up to four instructions can be made available to the instruction unit in a single clock cycle. On a miss, the cache line is loaded in one 64-byte beat; the instruction cache is non-blocking, providing for hits under misses.

The instruction cache operates similarly to the data cache when all eight ways of a set are locked. When the instruction cache is disabled (L1CSR1[ICE] = 0), instruction accesses bypass the instruction cache and are forwarded to the memory subsystem as caching-allowed. When the line is returned, up to four instructions are forwarded to the instruction unit in a single clock cycle, but the line is not loaded into the instruction cache.

For caching-inhibited fetches, a full cache line of data is fetched from the memory subsystem. When the line is returned, up to four instructions are forwarded to the instruction unit in a single clock cycle, but the line is not loaded into the instruction cache.

5.7.1.3 Cache allocation on misses

If there is a data cache miss for a caching-allowed load and the line is not already going to be allocated into the data cache as a result of a previous load miss, the data load miss causes a new line to be allocated into the data cache on a First-In-First-Out (FIFO) basis, provided the cache is not completely locked or disabled. Store misses in the data cache do not cause an allocation. Also, cache operations such as **dcbi** and **dcbf** that miss in the cache do not cause an allocation.

Instruction cache misses cause a new line to be allocated into the instruction cache on a Pseudo-Least-Recently-Used (PLRU) basis, provided the cache is not completely locked or disabled.

5.7.2 L1 cache block replacement

When a new block needs to be placed in the data cache, the FIFO replacement algorithm is used. Data cache replacement selection is performed at reload time and not when the data load miss occurs. Because the L1 data cache contains no modified data, no castout activities are performed.

When a new block needs to be placed in the instruction cache, the PLRU replacement algorithm is used. Instruction cache replacement selection is performed at reload time and not when an instruction cache miss is first recognized.

5.7.2.1 FIFO replacement

FIFO replacement is performed using a binary decision tree. FIFO replacement is implemented by only updating the tree when a new coherency granule is allocated.

Because the cache supports persistent locking, it does not replace locked lines. Lock bits are used at reload time to steer the decision tree away from selecting locked cache lines.

5.7.2.2 PLRU replacement

PLRU replacement is performed using a binary decision tree. 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 (the 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 the following table.

Table 5-3. 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

This figure shows the decision tree used to generate the victim line in the PLRU algorithm.

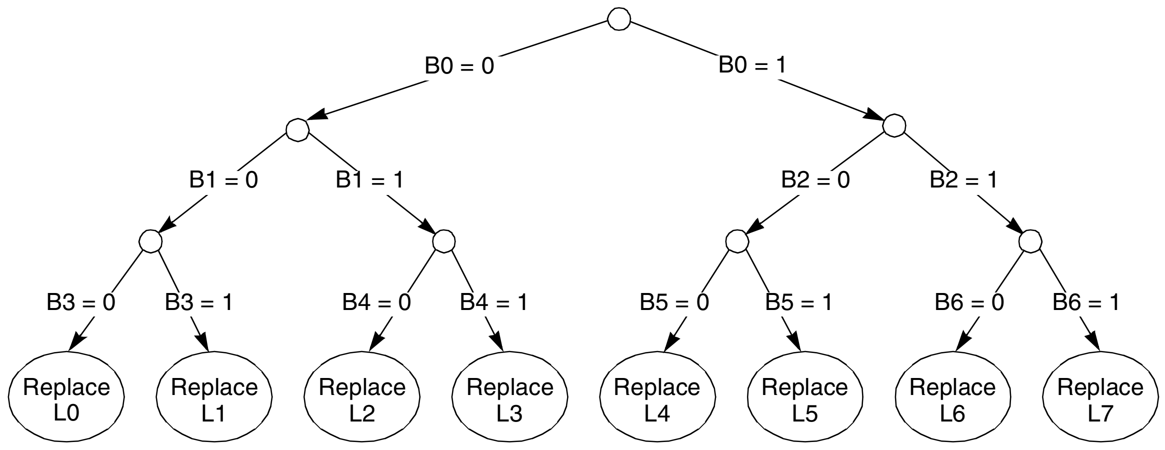


Figure 5-4. PLRU replacement algorithm

During power-up or hard reset, the valid bits of the L1 caches are automatically cleared to point to way L0 of each set.

5.7.2.3 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 the following table.

Table 5-4. 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.

Because the cache supports persistent locking, it does not replace locked lines. Lock bits are used at reload time to steer the decision tree away from selecting locked cache lines.

5.8 Cluster shared L2 cache

The cluster shared L2 cache has the following features:

- Dynamic Harvard architecture, merged instruction, and data cache
- 2048 KB array divided into four banks of 512 KB, organized as 512 sixteen-way sets of 64-byte cache lines
- 40-bit physical address
- Modified, exclusive, shared, invalid, incoherent, locked, and stale states
- Support for modified, exclusive, and shared intervention from the L2 cache
- Support for cache stashing to the L2 cache from other devices in the integrated device
- 16-way set associativity with Streaming Pseudo Least Recently Used with Aging (SPLRU with Aging) replacement. Additional support for Pseudo Least Recently Used (PLRU), Streaming Pseudo Least Recently Used (SPLRU), and First-In-First-Out (FIFO) replacement.
- Supports way partitioned cache operation. See Section 5.8.4.5, "L2 cache partitioning."
- 64-byte (16-word) cache-line, coherency-granule size.
- Support for individual line locking with persistent locks. See Section 5.8.4.4, "L2 cache line locking and unlocking."
- Inclusive for data lines and generally inclusive for instruction lines
- Reloaded whenever the L1 instruction cache makes a request, 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 have a copy of the line.
- Pipelined data array access with two-cycle repeat rate
- ECC protection for data, tag and status arrays
- ABIST support

5.8.1 Cluster memory subsystem block diagram

This figure shows the cluster memory subsystem.

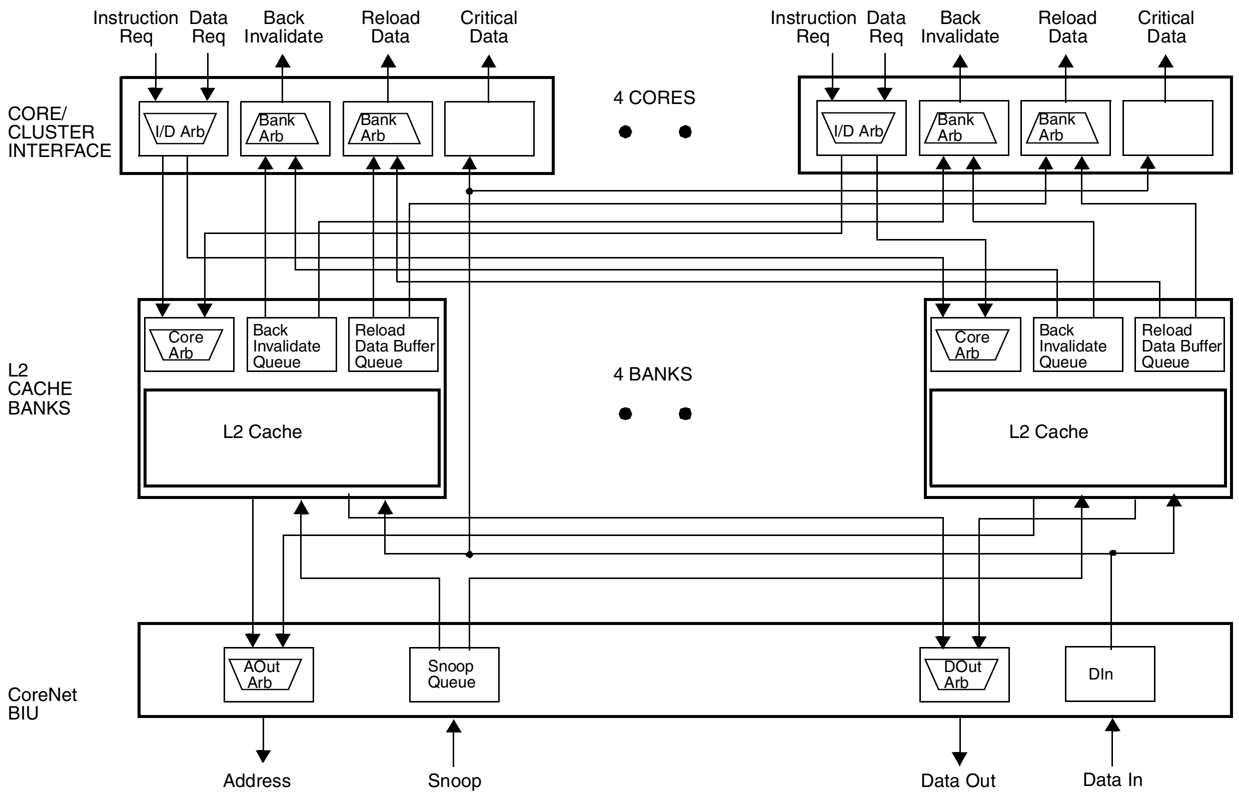


Figure 5-5. Cluster memory subsystem block diagram

5.8.1.1 Core/Cluster interface

The Core/Cluster interface is the interface between the core and the cluster shared L2 cache. There is one Core/Cluster interface block per core in the cluster.

The block arbitrates between instruction and data requests, forwards back invalidate requests to the core, forwards critical data to the core, and forwards reload data to the core Instruction Reload Data Buffer (IRLDB) and Data Reload Data Buffer (DRLDB).

5.8.1.2 L2 cache

The L2 cache is a shared L2 cache that supports the cores in the cluster. It it fully coherent with the system. The L2 cache is broken into 4 banks to support simultaneous access from all cores in the cluster, provided those accesses are to different banks.

The block arbitrates among core accesses to the L2 and provides back invalidate and reload queuing.

5.8.1.3 CoreNet Bus Interface Unit (BIU)

The CoreNet BIU handles all ordering and bus protocol and is the interface between the cluster and the external memory and caches. The CoreNet BIU also handles all snoop transactions for the cluster.

5.8.2 L2 cache structure

The L2 cache is divided into 4 identical 512 KB banks.

This figure shows the organization of a single bank of the L2 cache.

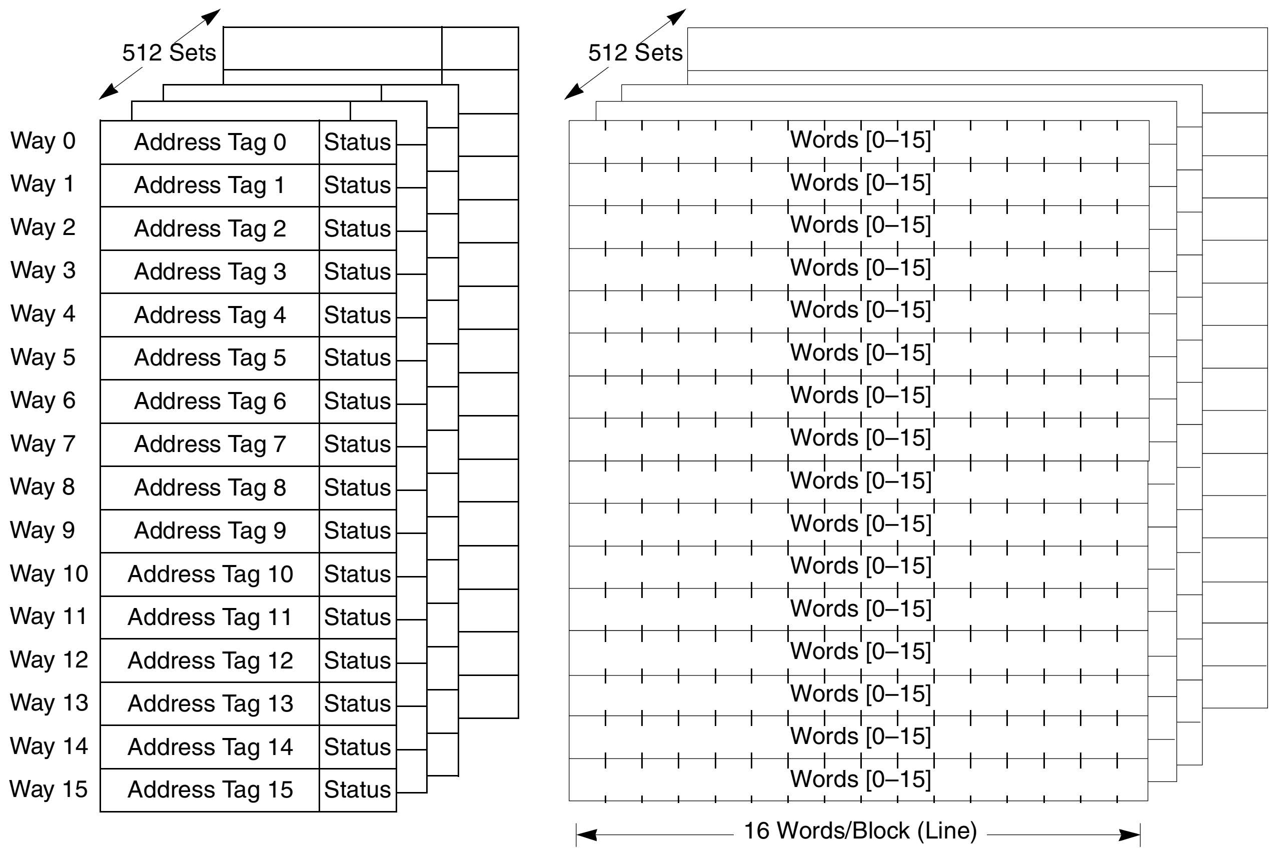


Figure 5-6. L2 cache bank organization

Each block (line) consists of 64 bytes of data, an address tag, and status bits. Also, although it is not shown in the previous figure, the cache has 8 ECC bits/doubleword and 7 ECC bits/tag.

Each cache block contains 16 contiguous words from memory that are loaded from a 16-word boundary (that is, physical addresses bits 34–39 are zero). In Decode Bank Hash Mode, physical address bits PA[23:31] provide the index to select a cache set and physical address bits PA[32:33] select the cache bank. In XOR Bank Hash mode, physical address bits PA[25:33] provide the index to select a cache set and an XOR of physical address bits PA[18:33] select the cache bank. Lower address bits PA[34:39] locate a byte within the selected block in both bank hash modes.

5.8.3 L2 cache coherency support and memory access ordering

This section describes the L2 cache coherency and coherency support.

5.8.3.1 L2 cache coherency model

The L2 cache supports a Modified/Exclusive/Shared/Invalid (MESI) 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 defines the coherency state of the cache line. The CoreNet interface uses this status to support coherency protocols and to direct coherency operations.

This table describes general MESI cache states.

Table 5-5. 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.

The cluster provides full hardware support for cache coherency and ordering instructions.

The cluster broadcasts cache management instructions (**dcbst**, **dcbstep**, **dcbf**, **dcbi** (M = 1), **mbar**, **sync 0**, **tlbsync**, **icbi**, **icbiep**) and cache touch or locking instructions with CT = 1.

5.8.3.2 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.

5.8.3.3 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 load/store operations. 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 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” because 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 and fetch for that line is sent to CoreNet and marked as non global, a data transaction does not hit to that line. 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 that 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. **icbi** operations do not hit to lines that are marked as coherent (N is not set) because the operation affects only the instruction cache. 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) because the operation affects only the data cache.

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. 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 performed the architectural method of modifying instructions using **dcbf** and **icbi**.

5.8.4 L2 cache control

This section describes how cache control instructions, L2CSR*n* bits, and partitioning bits are used to control the L2 cache.

5.8.4.1 Cache control instructions

The e6500 core implements the cache control instructions as described in Section 3.4.11.1, “User-level cache instructions,” and Section 3.4.12.4.1, “Supervisor-level cache instruction.” The L2 cache is identified as CT = 2.

5.8.4.2 Enabling and disabling the L2 cache

The L2 cache is enabled and disabled with the cache enable bit, L2CSR0[L2E]. Disabling the L2 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 the L2 cache is disabled, the cache tag state bits are ignored and the cache is not accessed. The L2 cache is disabled at start-up (L2CSR0[L2E] = 0).

Disabling the data cache has the following effects:

- Touch instructions (**dcbt**, **dcbtst**, **dcblc**, **dcbtls**, **dcbtstls**, **dcblc**, **dcblq.**, **icblc**, **icbtls**, and **icblq.**) targeting a disabled cache do not affect the cache.
- A **dcbz**, **dcbzl**, **dcba**, or **dcbal** instruction to a disabled cache zeros the cache line in memory but does not affect the cache when it is disabled.
- Cache lines are not snooped. Before the L2 cache is disabled, it must be flushed and invalidated to prevent coherency problems when it is enabled again.
- Accesses bypass the L2 cache and are forwarded to the CoreNet interface as caching-allowed. Returned data is forwarded to the requesting core.
- Other cache management instructions do not affect the disabled cache.

When an L2 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 a flash invalidation by setting L2CSR0[L2FI], and then continue to read L2FI 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 L2CSR0[L2E] 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.

Note that enabling either L1 cache without first enabling the L2 cache is not supported.

5.8.4.3 L2 cache flash invalidation

The L2 cache can be flash invalidated by setting L2CSR0[L2FI]. Note that this operation takes many cycles.

Software must set the L2FI bit if invalidation is desired after a reset. This causes a flash invalidation, after which the L2FI bit is cleared automatically (L2FI bit is not sticky). A flash invalidate operation is local only to the cluster that performs it, and other cluster's L2 caches are not affected. Software should always poll the L2FI bit after setting it to determine when the invalidation has been completed and then perform an **isync**.

Note that if L2CSR0[L2FI] and L2CSR0[L2LFC] are set simultaneously with the same register write operation, then the L2 flash invalidate function and the L2 lock flash clear function are performed in parallel. In this case, hardware will clear both the valid and lock bits for each coherency granule, as well as all the tag and PLRU bits, with a single pass through all indices of the cache.

5.8.4.4 L2 cache line locking and unlocking

The L2 supports persistent locking. If a coherency granule is invalidated in any way while locked, the lock bit remains set and the line in the cache remains un-allocatable by other coherency granules. If the same address accesses the cache with an allocatable request, the request misses but allocates to the same line in the cache to which it was previously allocated.

Lines are locked in the L2 cache by software using a series of "touch and lock set" instructions. The following instructions can lock a line in the 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 the L2 cache by software using a series of "lock clear" instructions. The following instructions are used to clear the lock in the 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 as 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 setting L2CSR0[L2LFC], as described in Section 2.12, "L2 cache registers." Note that this operation takes many cycles.

5.8.4.5 L2 cache partitioning

The L2 cache supports a flexible allocation and partitioning policy. Each transaction that misses in the cache looks in a table to determine whether or not to allocate and which ways are available for allocation. Table entries are matched by comparing Partition IDs. Allocation is then controlled dependent on transaction type and WIMG attributes, as well as stashing control signals.

Each entry in the table is composed of a set of three registers: L2PIR*n*/L2PAR*n*/L2PWR*n*.

L2PIR contains a bit for each possible Partition ID value. A 1 indicates that the Partition ID must follow the allocation rules as defined by the L2PAR/L2PWR registers.

L2PAR controls allocation of the following individual allocation types:

- Instruction read
- Data read
- Store
- Stash

L2PWR controls which ways are allocatable for the given set of allocation rules. All bits of the L2PIR/L2PAR/L2PWR registers default to 1 at reset so that all Partition IDs and transaction types are allocatable to all ways. For more information on how to partition the ways of the L2 cache, see Section 2.12.4, "L2 cache partitioning registers."

5.8.4.6 L2 cache flushing

The L2 cache can be flushed of modified data by setting L2CSR0[L2FL]. The L2 cache must be enabled for the flush operation to take effect. Note that this operation takes many cycles.

To flush the L2 cache and ensure that no valid entries exist after the flush, the following instruction sequence should be used:

- Clear all bits of L2PAR or L2PAR*n* to prevent future operations from allocating in the L2 cache.
- Write 1 to L2CSR0[L2FL].
- Wait for L2CSR0[L2FL] to be cleared by hardware.

5.8.5 L2 cache operation

5.8.5.1 L2 cache block replacement

If a transaction must allocate in the L2 cache, it must select a location in the cache and determine whether or not an existing coherency granule needs to be cast out before the new coherency granule can be established in its place. Victim selection involves decoding a PLRU binary tree and the current state of the lock bits and involves partitioning information to select an available location in the cache.

If all ways of the cache are already locked, then an overflow condition occurs and the L2CSR0[L2LO] bit is set. If the L2CSR0[L2LOA] = 1, the allocating transaction replaces one of the current locked lines. If the L2CSR0[L2LOA] = 0, the allocating transaction is not allowed to allocate.

The L2CSR0[L2_REP] field determines the L2 replacement policy, as described in the following table.

Table 5-6. L2 replacement policy

L2CSR0[L2_REP]	Mode	Description
00	SPLRU with Aging	Streaming-Pseudo-Least-Recently-Used with Aging (default)
01	FIFO	First-In-First-Out
10	SPLRU	Streaming-Pseudo-Least-Recently-Used
11	PLRU	Pseudo-Least-Recently-Used

5.8.5.1.1 PLRU replacement

PLRU replacement is performed using a binary decision tree. There is an identifying bit for each cache way, L[0–15]. There are fifteen PLRU bits, B[0–14], for each set in the cache to determine the line to be cast out (the 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 the following table.

Table 5-7. L2 PLRU replacement way selection

PLRU Bits								Way Selected for Replacement
B0	0	**B1**	0	**B3**	0	**B7**	0	L0
	0		0		0		1	L1
	0		0		1	**B8**	0	L2
	0		0		1		1	L3
	0		1	**B4**	0	**B9**	0	L4
	0		1		0		1	L5
	0		1		1	**B10**	0	L6
	0		1		1		1	L7
	1	**B2**	0	**B5**	0	**B11**	0	L8
	1		0		0		1	L9
	1		0		1	**B12**	0	L10
	1		0		1		1	L11
	1		1	**B6**	0	**B13**	0	L12
	1		1		0		1	L13
	1		1		1	**B14**	0	L14
	1		1		1		1	L15

5.8.5.1.2 SPLRU and SPLRU with Aging replacement

SPLRU and SPLRU with Aging replacement can be used to help detect streaming data that is transient and should not remain in the cache.

5.8.5.1.3 FIFO replacement

FIFO replacement is performed using a binary decision tree. FIFO replacement is implemented by only updating the tree when a new coherency granule is allocated.

5.8.5.2 Special scenarios for L2 cache

This section describes special scenarios of operations in the L2 cache.

5.8.5.2.1 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 will also be snooped when the **icbi** is sent out on the CoreNet interface.

5.8.6 L2 cache errors

The L2 data is protected by 8 bits of ECC per doubleword. L2 tag is protected by 7 bits of ECC per entry/set.

L2 tag and data ECC is written whenever one of the following occurs:

- A store instruction (or **dcbz**, **dcbzep**, **dcbzl**, **dcbzlep**, **dcba**, or **dcbal**)
- A reload into the L2 cache

L2 data ECC is checked whenever:

- A load instruction hits in the L2 cache
- An instruction fetch hits in the L2 cache

L2 tag ECC is checked for all cache transactions, including snoops.

L2 error checking is disabled by default and can be enabled by writing 1 to L2CSR0[L2PE].

Each type of L2 error is recorded in the L2ERRDET register, provided L2CSR0[L2PE] = 1.

Each type of L2 error checking can be enabled or disabled with the L2ERRDIS register. By default, each type of L2 error checking is enabled, provided L2CSR0[L2PE] = 1.

Each type of L2 error interrupt can be enabled or disabled with the L2ERRINTEN register. By default, each type of L2 error interrupt is disabled, provided L2CSR0[L2PE] = 1. L2 error interrupts are serviced by the on-chip MPIC block.

Thresholds for L2 ECC errors can be set in the L2ERRCTL register. By default, the threshold is set to 0, provided L2CSR0[L2PE] = 1.

This table describes how L2ERRDET is updated for L2 tag errors.

Table 5-8. L2 tag errors

Error	L2CSR0	L2ERRDIS			L2ERRINTEN			L2ERRDET		
	L2PE	TMHITDIS	TMBECC DIS	TSBECC DIS	TMHIT INTEN	TMBECC INTEN	TSBECC INTEN	TMHIT	TMBECC ERR	TSBECC ERR
Tag multi-way hit	0	x	x	x	x	x	x	0	0	0
	1	0	x	x	0	x	x	1	0	0
	1	0	x	x	1	x	x	1	0	0
	1	1	x	x	0	x	x	0	0	0
	1	1	x	x	1	x	x	0	0	0

Table 5-8. L2 tag errors (continued)

Error	L2CSR0	L2ERRDIS			L2ERRINTEN			L2ERRDET		
	L2PE	TMHITDIS	TMBECC DIS	TSBECC DIS	TMHIT INTEN	TMBECC INTEN	TSBECC INTEN	TMHIT	TMBECC ERR	TSBECC ERR
Tag single bit ECC error	0	x	x	x	x	x	x	0	0	0
	1	x	x	0	x	x	0	0	0	1
	1	x	x	0	x	x	1	0	0	1
	1	x	x	1	x	x	x	0	0	0
Tag multi bit ECC error	0	x	x	x	x	x	x	0	0	0
	1	x	0	x	x	0	x	0	1	0
	1	x	0	x	x	1	x	0	1	0
	1	x	1	x	x	x	x	0	0	0

This table describes how L2ERRDET is updated for L2 data errors.

Table 5-9. L2 data errors

Error	L2CSR0	L2ERRDIS		L2ERRINTEN		L2ERRDET	
	L2PE	MBECC DIS	SBECC DIS	MBECC INTEN	SBECC INTEN	MBECC ERR	SBECC ERR
Data single-bit ECC error	0	x	x	x	x	0	0
	1	x	0	x	0	0	1
	1	x	0	x	1	0	1
	1	x	1	x	x	0	0
Data multi-bit ECC error	0	x	x	x	x	0	0
	1	0	x	0	x	1	0
	1	0	x	1	x	1	0
	1	1	x	x	x	0	0

5.8.6.1 L2 cache ECC error injection

ECC error injection provides a way to test error recovery software by intentionally injecting ECC errors into the L2 cache.

L2ERRINJCTL provides bits to inject ECC errors into the L2 tag (L2ERRINJCTL[TERRIEN]) and the L2 data (L2ERRINJCTL[DERRIEN]). In addition, L2ERRINJCTL also provides an error injection mask for ECC syndrome bits (L2ERRINJCTL[ECCERRIM]).

L2ERRINJHI and L2ERRINJLO provide error injection masks for the tag and data itself.

5.8.7 L2 cache performance monitor events

Performance monitor events associated with the L2 cache are described in 9.12.6, “Event selection.”

5.9 CoreNet Bus Interface Unit (BIU)

The CoreNet BIU handles all ordering and bus protocol and is the interface between the cluster and the external memory and caches. The CoreNet BIU also handles all snoop transactions for the cluster.

CoreNet itself is not described in this document.

Chapter 6
Memory Management Units (MMUs)

This chapter describes the implementation details of the e6500 MMU. *EREF* 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 e6500 MMU overview

The e6500 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. The e6500 core implements MMU architecture version 2, described in *EREF* as "MMU V2."

6.1.1 MMU features

The e6500 core has the following features:

- 64-bit effective address (EA) translated to 40-bit real (physical) address (using an 86-bit virtual address)
- Two-level MMU containing a total of eight TLBs for maximizing TLB hit rates
- Logical Partition ID (LPIDR) register for supporting up to 64 partitions at any time in the TLB
- Process ID (PID) register for supporting up to 16 K translation IDs at any time in the TLB per partition
- TLB entries for variable-sized, 4 KB to 1 TB pages in powers of two and fixed-size (4 KB) pages
- Support for both software and hardware tablewalk. Hardware tablewalk is supported for 4 KB pages, which are loaded into TLB0 on a TLB miss
- TLBs maintained by system software through the TLB instructions and nine MMU assist (MAS) registers and through hardware tablewalk for TLB0
- An eight entry fully associative logical-to-real address translation (LRAT) allowing guest operating systems to perform TLB writes without hypervisor intervention. The LRAT also performs logical-to-real address translation during hardware tablewalks for guest TLB misses.

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 a large range of variable size page (VSP) page sizes, as shown in Section 6.2.5, "Variable-sized pages."
- Two 64-entry, four-way set-associative TLB arrays (one for instruction accesses and one for data accesses) that support only 4 KB 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 TLB1 entries are a proper subset of TLB1 entries resident in L2 MMU (completely maintained by the hardware).
- L1 TLB0 entries are not a proper subset of TLB0 entries resident in L2 MMU.

The Level 2 MMU has the following features:

- A 64-entry, fully-associative unified (for instruction and data accesses) L2 TLB array (TLB1) supporting a large range of VSP page sizes, as shown in Section 6.2.5, "Variable-sized pages"
- A 1024-entry, 8-way set-associative unified (for instruction and data accesses) L2 TLB array (TLB0) supports only 4 KB pages.
- Hardware assistance for TLB miss exceptions
- TLB1 and TLB0 managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, **tlbivax**, **tlbilx**, **mfspr**, and **mtspr** instructions
- TLB0 also managed through hardware tablewalk
- Parity detection for TLB0
- Performs invalidations in TLB1 and TLB0 caused by **tlbivax** and **tlbilx** instructions executed by this core. Snoops TLB1 and TLB0 for **tlbivax** invalidations executed by other processors.
- 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 e6500 core provides support for maintaining TLB entries in software with the resources shown in the following table. Section 6.6, "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	6.5.1/6-27
	tlbwe	TLB Write Entry instruction	6.5.2/6-27
	tlbsx r*A*, r*B* (software must use the preferred form: **tlbsx 0,** r*B*)	TLB Search for Entry instruction	6.5.3/6-29
	tlbilx	TLB Invalidate Local instruction	6.5.4/6-29
	tlbivax r*A*, r*B*	TLB Invalidate instruction	6.5.5/6-30
	tlbsync	TLB Synchronize Invalidations	6.5.6/6-31

Table 6-1. TLB maintenance programming model (continued)

Features		Description	Section/Page
Registers	PID	Process ID register	Table 6-10
	LPIDR	Logical Partition ID register	
	LRATCFG	Logical-to-Real Address Translation Configuration register	
	LRATPS	Logical-to-Real Address Translation Page Size register	
	MMUCSR0	MMU Control and Status register	
	MMUCFG	MMU Configuration register	
	TLB0CFG–TLB1CFG	TLB configuration registers	
	TLB0PS–TLB1PS	TLB page size registers	
	EPTCFG	Embedded Page Table configuration register	
	MAS0–MAS8	MMU assist registers	
	(G)DEAR	(Guest)Data Exception Address register	
	(G)ESR	(Guest)Exception Syndrome register	
	LPER, LPERU	Logical Page Exception register (upper)	
	EPLC	External PID Load Context register	
	EPSC	External PID Store Context register	
Exceptions/ interrupts	Instruction TLB miss	Causes instruction TLB error interrupt	4.9.15/4-34
	Data TLB miss	Causes data TLB error interrupt	4.9.14/4-33
	Execute access control Page table fault	Causes ISI interrupt	4.9.5/4-23
	Read access control Write access control Virtualization fault Page table fault Byte ordering Cache locking Storage synchronization	Causes DSI interrupt	4.9.4/4-20
	LRAT error	Causes LRAT error interrupt	4.9.20.1/44-45

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 64-bit effective addresses. The MMU translates these addresses to 40-bit real addresses, which are used for memory accesses, using an interim virtual address. In multicore implementations, such as the e6500 core, the virtual address is formed by concatenating MSR[GS] || LPIDR || MSR[IS|DS] || 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 that hit in the L2 MMU, the L1 TLB entries are replaced from their L2 TLB counterparts using a true LRU algorithm. If the L2 MMU misses, a page table translation is attempted and the L2 MMU is searched for a matching indirect entry and, if found, performs a page table translation. If the page table translation finds a valid page table entry (PTE), the PTE is loaded into the L2 MMU and the translation is performed using the loaded entry. See Section 6.2.2, "Page table translation."

The e6500 core contains one set of L1 MMU structures for both threads for instruction address translation and a set of private L1 MMU structures per thread for data translations.

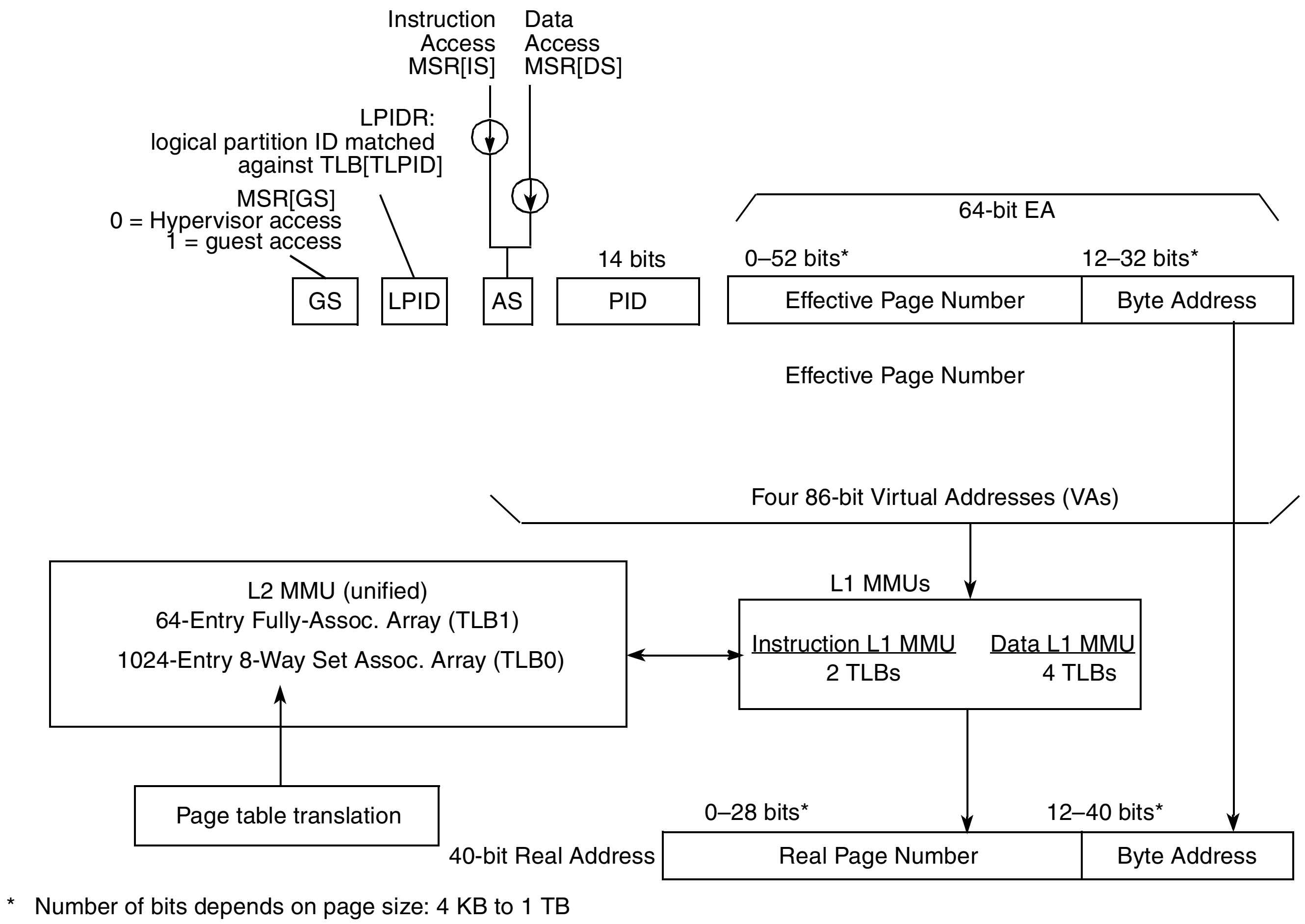


Figure 6-1. Effective-to-real address translation flow in e6500

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 an embedded hypervisor.

6.2.2 Page table translation

Hardware page tables are used to perform translations that miss in the L2 TLB. The result of such a translation is that TLB0 is updated with a new entry from the hardware page table, and the original access that missed is translated using that new TLB entry.

A page table translation occurs when a virtual address is presented to the MMU for translation and no matching TLB entry exists in any TLB array. This is commonly referred to as a hardware page tablewalk. A page table translation is performed by hardware by locating and reading a page table entry (PTE) from memory and using the contents of the PTE to perform the virtual address to real address translation. In guest mode, the contents of the PTE are translated through the LRAT to obtain the real address.

6.2.2.1 Locating a hardware page table and PTE

A hardware page table is defined by an indirect entry in TLB1. An indirect entry is an entry in TLB1 that has the indirect bit (IND) equal to 1. Software writes indirect entries into TLB1 with **tlbwe**, which allows the hardware to locate and read the PTE entry associated with a virtual address. Indirect entries cannot be written to TLB0 because TLB0 does not support the indirect bit.

An indirect entry matches a VA in the same manner that a non-indirect entry does with the exception that the indirect entry has the IND bit set and the lookup that occurs is during a page table translation. Thus the address range defined by the EPN field and page size (TSIZE) field describe the range of addresses that the indirect entry maps. All other virtual address identifiers are used in the same manner as non-indirect entry lookup does. The RPN field of an indirect entry contains the real address of the start of a page table, which maps the virtual address space of this indirect entry.

If no indirect entry matches for the VA with IND = 1 lookup, then an instruction TLB miss exception (and instruction TLB error interrupt) occurs if the original translation was an instruction fetch or a data TLB miss exception (and data TLB error interrupt) occurs if it was not an instruction fetch.

If a matching indirect entry is found and the indirect entry has VF = 1, then a virtualization fault or an instruction virtualization fault exception (and data storage interrupt or instruction storage interrupt) occurs.

Page tables consist of consecutive 8-byte PTEs, each of which represents 4 KB of effective address space. Each 4 KB of effective address offset from the start of the effective address specified by EPN corresponds to 8 bytes of offset of real address from RPN. During a page table translation, the offset from the EA of the original memory reference that caused the page table translation from the EPN of the indirect entry is used to determine which PTE in the page table in real memory addressed by RPN is used to map the virtual address that caused the page table translation. The real address of the 8-byte PTE is equal to the RPN of the indirect entry shifted left 12 bits plus the number of 4 KB blocks of effective address that EA is in from EPN times the size of a PTE:

PTE_real_addr = (RPN << 12) + ((EA - (EPN<<12)) / 4096) * 8

For example, assume an indirect entry exists in the TLB, which has the contents described in the following table.

Table 6-2. Example indirect TLB entry

Field	Value
TGS	1
TLPID	4
TID	22
TS	1
EPN	0x0_0000_0001_0000 corresponds to an EA of: 0x0000_0000_1000_0000
TSIZE	11 (2 MB)
RPN	0x0_0000_0000_0501 corresponds to an RA of: 0x0000_0000_0050_1000

In this example, the virtual address space identifiers are GS = 1, LPID = 4, PID = 22, and AS = 1. This virtual address space might be a typical virtual address space for a user program. Within that address space, the effective address range that is mapped by this entry is 0x0000_0000_1000_0000 to 0x0000_0000_101f_ffff because the EPN starts at effective address 0x0000_0000_1000_0000 and the page size is 2MB.

Assume that a memory reference (a **lwz** instruction, for example) for the same virtual address space (MSR[GS] = 1, LPIDR = 4, PID = 22, and MSR[DS] = 1) produces an EA of 0x0000_0000_1000_3010 and that there is not a direct translation for this virtual address present in the TLB.

The miss in the TLB for this virtual address starts a page table translation and the following actions occur:

— The virtual address is looked up as an indirect entry in TLB1: IND=1 with the same VA.

— If the indirect entry is not found, a data TLB error interrupt occurs and system software normally writes an indirect TLB1 entry that covers this virtual address and possibly creates and initializes the PTE (if it has not already done so). Then, system software would return from the interrupt to re-execute the instruction which caused the TLB error.

— If the indirect entry is found, and the indirect entry has VF = 1, a virtualization fault exception occurs and a DSI interrupt is taken.

— If the indirect entry is found and VF is not set, the offset of the EA from the start address of the page from the indirect entry EPN field is calculated. In this example:

```
offset = EA - (EPN || 0x000)
0x3100 = x0000_0000_1000_3100 - (0x0_0000_0001_0000 || 0x000)
```

— This offset is divided by 4 K, which represents an index into 4 KB pages, which are defined by this indirect entry:

```
page_4K_index = offset / 4096
3 = 0x3100 / 0x1000
```

— The 4 KB page index is multiplied by eight to account for the 8 bytes of memory that each PTE requires. The result is added to the RPN, giving the real address of the PTE:

```
PTE_real_addr = (RPN || 0x000) + (page_4K_index * 8)
0x0000_0000_0050_1018 = (0x0_0000_0000_0501 || 0x000) + (3 * 8)
```

6.2.2.2 Translation and TLB update using a PTE

Once the appropriate PTE is located, it is read from memory and is used to perform translation for the original virtual address that caused the page table translation. The PTE memory access uses WIMGE settings of 0b00100. The PTE is an 8-byte structure described in Section 6.2.3, “Page table entry (PTE).”

If the PTE is not valid (PTE[V] = 0) then a page table fault exception occurs. In addition, an instruction storage interrupt or data storage interrupt occurs depending on whether the original translation was an instruction fetch or a data access.

If the PTE is valid, then a TLB entry (*entry*) is constructed from the fields of the PTE, the virtual address of the indirect entry (*ind_entry*), and the EA of the original translation as follows:

```
$bap \leftarrow PTE_{BAP}$
$ux \leftarrow bap_0 \ \& \ PTE_R$
$sx \leftarrow bap_1 \ \& \ PTE_R$
$uw \leftarrow bap_2 \ \& \ PTE_R \ \& \ PTE_C$
$sw \leftarrow bap_3 \ \& \ PTE_R \ \& \ PTE_C$
$ur \leftarrow bap_4 \ \& \ PTE_R$
$sr \leftarrow bap_5 \ \& \ PTE_R$

$entry_{UX,SX,UW,SW,UR,SR} \leftarrow ux,sx,uw,sw,ur,sr$
$entry_V \leftarrow PTE_V$
$p \leftarrow 0b00010$
$entry_{TSIZE} \leftarrow p$
$entry_{IND} \leftarrow 0$
$entry_{WIMGE} \leftarrow PTE_{WIMGE}$
$entry_{TID} \leftarrow ind\_entry_{TID}$
$entry_{TLPID} \leftarrow ind\_entry_{TLPID}$
$entry_{TS} \leftarrow ind\_entry_{TS}$
$entry_{TGS} \leftarrow ind\_entry_{TGS}$
$entry_{EPN} \leftarrow EA_{0:53-p}$
$entry_{VF} \leftarrow 0$
$entry_{IPROT} \leftarrow 0$
$entry_{U0} \leftarrow PTE_{U0}$
```

```
entryU1 ← $PTE_{U1}$
$entry_{U2..U3}$ ← 0
$entry_{X0\ X1}$ ← $PTE_{48:49}$
```

The RPN field of the TLB entry is set depending on whether the page table translation is a hypervisor or guest translation.

If the translation is a hypervisor translation (the GS value of the virtual address is 0), then the RPN field of the TLB entry is set to the value of the lower 28 bits of the ARPN field of the PTE:

```
$entry_{RPN}$ ← $PTE_{ARPN}$
```

If the translation is a guest translation, the ARPN field of the PTE is translated through the LRAT, as described in Section 6.4.5, "LRAT translation" and the resulting real page number is written to the TLB entry. If no matching LRAT translation exists, then an LRAT error exception (and LRAT error interrupt) occurs and ESR(GSR), DEAR (GDEAR), and LPER are set so that hypervisor software can load an appropriate LRAT translation:

```
if not instruction fetch then
    $ESR_{DATA}$ ← 1
    DEAR ← EA
$ESR_{PT}$ ← 1
$LPER_{ALPN}$ ← $PTE_{ARPN}$
$LPER_{WIMGE}$ ← $entry_{WIMGE}$
$LPER_{LPS}$ ← $entry_{TSIZE}$    // always 4K page size
```

Only 4 KB page sizes are allowed for translations represented by PTEs. Setting PTE[PS] to any other value is ignored.

The BAP, R, and C fields in the PTE are used to determine access control. Note that all permissions require that the PTE[R] bit be set and that write permissions also require the PTE[C] bit be set. PTE[R] (referenced) is set by system software when the page has been read or written, and PTE[C] (changed) is set by system software when the page has been modified. The combination of the base access permissions with the referenced and changed bits for performing permission checks allows software to more easily perform referenced and change bit recordings without having to change the base access permissions in the PTE.

Access control using the permission bits derived from the PTE is performed in the same manner as access control for translations that hit in the TLB. If an access control exception (read, write, or execute) occurs, an instruction storage or data storage interrupt occurs the same as normal translation.

If no errors occur during page table translation, the constructed TLB entry is written to TLB0.

If page table translation results in an error (ISI, DSI, or LRAT Error), the constructed TLB entry is not written to TLB0 and ESR[PT] (or GESR[PT]) is set to 1.

On the e6500 core both software and hardware must ensure that multiple TLB entries for the same virtual address are not created in the TLB. Hardware ensures that simultaneous page table translations from different threads for the same virtual address do not write multiple TLB entries. Similarly, software must ensure that it does not write a TLB entry using **tlbwe** if another thread can establish the same TLB entry either with **tlbwe** or through a page table translation.

NOTE

The assignment of the X0 and X1 bits from bits 48:49 of the PTE is specific to the e6500 core implementation. Future processors may assign these bits differently.

6.2.3 Page table entry (PTE)

The PTE entry is an 8-byte descriptor containing the information required to create a TLB entry during a successful page table translation. PTEs are contained in memory in groups that form a page table that is located by a corresponding indirect entry in the TLB.

A PTE represents a 4 KB mapping. The PTE[PS] field is ignored on the e6500 core.

This figure shows the in-memory format of a PTE.

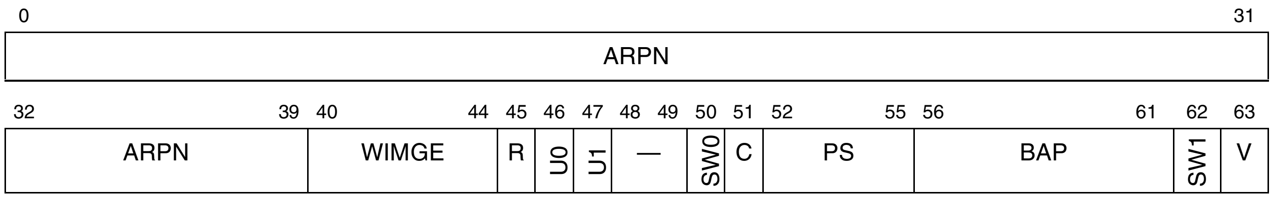


Figure 6-2. Page table entry (PTE)

This table describes the PTE fields.

Table 6-3. Page table entry field descriptions

Bits	Name	Description
0–39	ARPN	Abbreviated real page number. Contains the value to be placed in the TLB[RPN] field when creating the TLB during a page table translation. If this is a guest PTE, then the ARPN is a logical page number (and will be translated through the LRAT when a guest page table translation occurs). When the e6500 core reads this field during a page table translation, only the low-order 28 bits are used.
40–44	WIMGE	Storage control attributes. Contains the value to be placed in the TLB[WIMGE] field when creating the TLB during a page table translation.
45	R	Referenced bit. Used by software to denote that the page has been referenced. During page table translation, this bit is also used for access control.
46	U0	User defined storage control bit 0. Contains the value to be placed in the TLB[U0] field when creating the TLB during a page table translation.
47	U1	User defined storage control bit 1. Contains the value to be placed in the TLB[U1] field when creating the TLB during a page table translation.
48–49	—	Reserved. For the e6500 core, contains the value to be placed in the TLB[X0,X1] fields when creating the TLB during a page table translation.
50	SW0	Available for software use.
51	C	Changed bit. Used by software to denote that the page has been modified. During page table translation, this bit is also used for access control.

Table 6-3. Page table entry field descriptions (continued)

Bits	Name	Description
52–55	PS	Page size. Represents the page size of this PTE entry. This value is ignored for the e6500 core and 0b00010 is placed in the TLB[TSIZE] field when creating the TLB during a page table translation.

56–61	BAP	Base access permissions. Contains the value to be placed in the TLB[UR,SR, UW, SW, UX, SX] fields when creating the TLB during a page table translation as follows:
		UX = BAP_0 & R
		SX = BAP_1 & R
		UW = BAP_2 & R & C
		SW = BAP_3 & R & C
		UR = BAP_4 & R
		SR = BAP_5 & R
		
		Note: The ANDing of the base access permissions with the R and C bits allows software to receive appropriate DSI/ISI interrupts for referenced and changed bit handling.

62	SW1	Available for software use.
63	V	Valid 0 The PTE entry is not valid (invalid) 1 The PTE entry is valid

6.2.4 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 e6500 core, 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**, **lfdepx**, **lvepx**, and **lvepxl**
- Store-type instructions: **stbepx**, **sthepx**, **stwepx**, **stdepx**, **dcbzep**, **dcbzlep**, **stfdepx**, **stvepx**, and **stvepxl**

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 6-3:
  - — 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 the 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].

This figure shows how to form a virtual address using external PID.

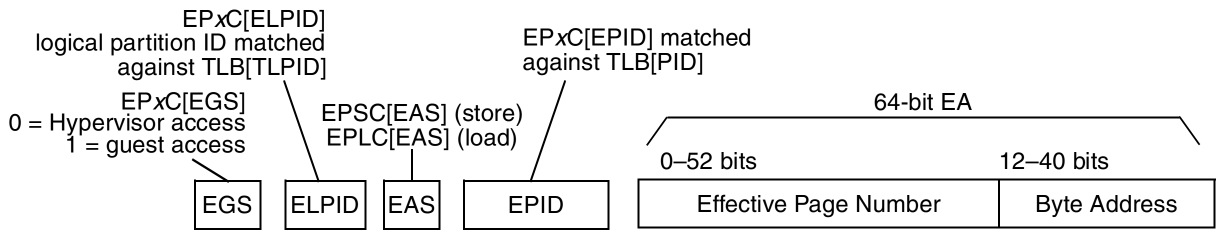


Figure 6-3. Forming a virtual address using external PID

6.2.5 Variable-sized pages

The following shows page sizes supported by the fully-associative TLB1 array that support variable-sized pages (VSPs). Note that the e6500 core only implements the low-order 28 bits of the architected 52-bit RPN field of the TLB.

Table 6-4. TLB1 page sizes

TSIZE (TLB_{SIZE})	Page Size	EPN and RPN Bits Required to be Zero	Real Address after Translation
0b00010	4KB	none	$RPN_{24:51}$ ll $EA_{52:63}$
0b00011	8KB	51	$RPN_{24:50}$ ll $EA_{51:63}$
0b00100	16KB	50:51	$RPN_{24:49}$ ll $EA_{50:63}$
0b00101	32KB	49:51	$RPN_{24:48}$ ll $EA_{49:63}$
0b00110	64KB	48:51	$RPN_{24:47}$ ll $EA_{48:63}$
0b00111	128KB	47:51	$RPN_{24:46}$ ll $EA_{47:63}$
0b01000	256KB	46:51	$RPN_{24:45}$ ll $EA_{46:63}$
0b01001	512KB	45:51	$RPN_{24:44}$ ll $EA_{45:63}$
0b01010	1MB	44:51	$RPN_{24:43}$ ll $EA_{44:63}$
0b01011	2MB	43:51	$RPN_{24:42}$ ll $EA_{43:63}$
0b01100	4MB	42:51	$RPN_{24:41}$ ll $EA_{42:63}$
0b01101	8MB	41:51	$RPN_{24:40}$ ll $EA_{41:63}$
0b01110	16MB	40:51	$RPN_{24:39}$ ll $EA_{40:63}$
0b01111	32MB	39:51	$RPN_{24:38}$ ll $EA_{39:63}$
0b10000	64MB	38:51	$RPN_{24:37}$ ll $EA_{38:63}$
0b10001	128MB	37:51	$RPN_{24:36}$ ll $EA_{37:63}$
0b10010	256MB	36:51	$RPN_{24:35}$ ll $EA_{36:63}$
0b10011	512MB	35:51	$RPN_{24:34}$ ll $EA_{35:63}$
0b10100	1GB	34:51	$RPN_{24:33}$ ll $EA_{34:63}$
0b10101	2GB	33:51	$RPN_{24:32}$ ll $EA_{33:63}$
0b10110	4GB	32:51	$RPN_{24:31}$ ll $EA_{32:63}$
0b10111	8GB	31:51	$RPN_{24:30}$ ll $EA_{31:63}$
0b11000	16GB	30:51	$RPN_{24:29}$ ll $EA_{30:63}$
0b11001	32GB	29:51	$RPN_{24:28}$ ll $EA_{29:63}$
0b11010	64GB	28:51	$RPN_{24:27}$ ll $EA_{28:63}$
0b11011	128GB	27:51	$RPN_{24:26}$ ll $EA_{27:63}$
0b11100	256GB	26:51	$RPN_{24:25}$ ll $EA_{26:63}$

Table 6-4. TLB1 page sizes (continued)

TSIZE (TLB_{SIZE})	Page Size	EPN and RPN Bits Required to be Zero	Real Address after Translation
0b11101	512GB	25:51	RPN_{24} ‖ $EA_{25:63}$
0b11110	1TB	24:51	$EA_{24:63}$

6.2.5.1 Checking for TLB entry hit

A hit to multiple matching TLB entries is considered a programming error. If this occurs, the TLB generates an invalid address, TLB entries may be corrupted, and a machine check or error report interrupt is generated if HID0[EN_L2MMU_MHD]. If HID0[EN_L2MMU_MHD] is not set when the error occurs, the resulting translation is undefined.

This figure 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.

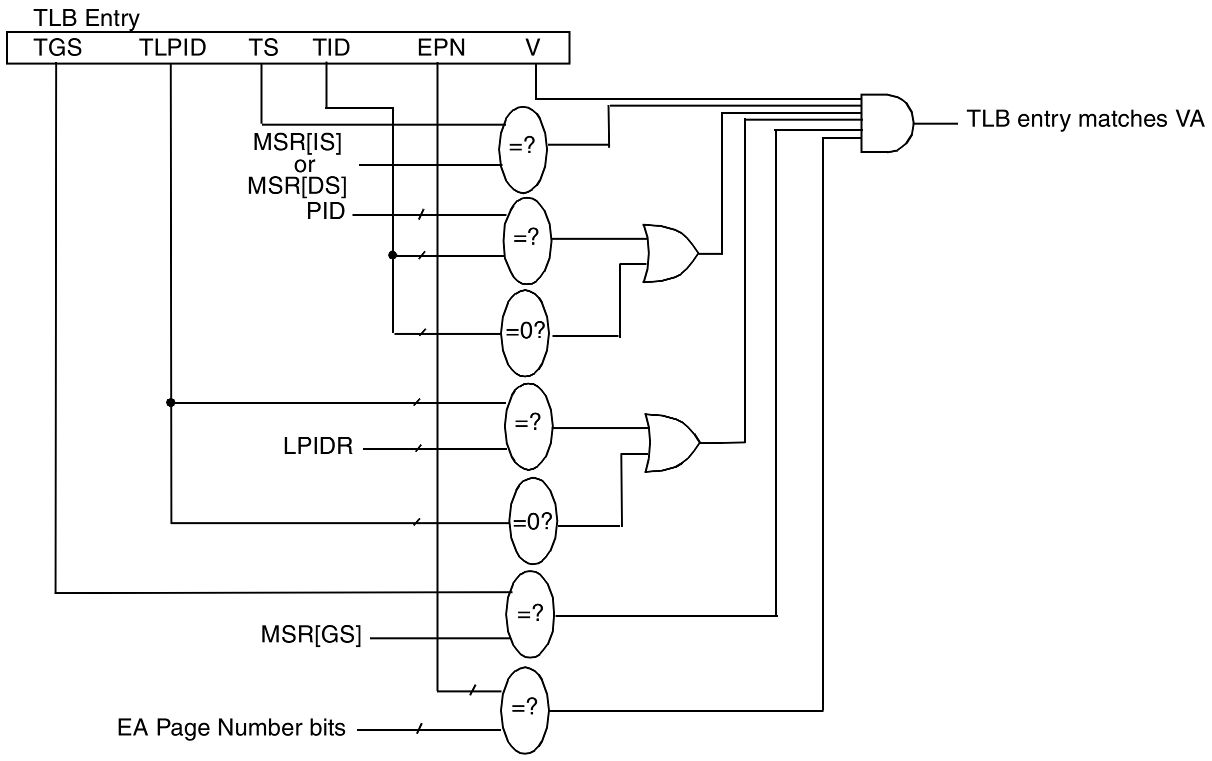


Figure 6-4. Virtual address and TLB-entry compare process

6.2.6 Checking for access permissions

When a TLB entry matches with a virtual addresses 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 e6500 core functions as described in *EREF*.

Note that a page table translation uses PTE[R] and PTE[C] (referenced and changed bits) to form the permissions when a TLB entry is loaded from a page table walk.

The e6500 core also requires write permission for caching-inhibited accesses using decorated storage loads.

6.3 Translation lookaside buffers (TLBs)

To maximize address translation performance and to provide ample flexibility for the operating system, the e6500 core implements eight TLB arrays. Figure 6-5 contains a more detailed description of the two-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 thread's private 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. The thread's private data L1 MMUs also operate in parallel with the other thread's private data L1 MMUs, and each thread can hit in the data L1 MMU in the same clock cycle.

This figure shows the 40-bit real addresses and the eight-way set associative TLB0 used in the e6500 core.

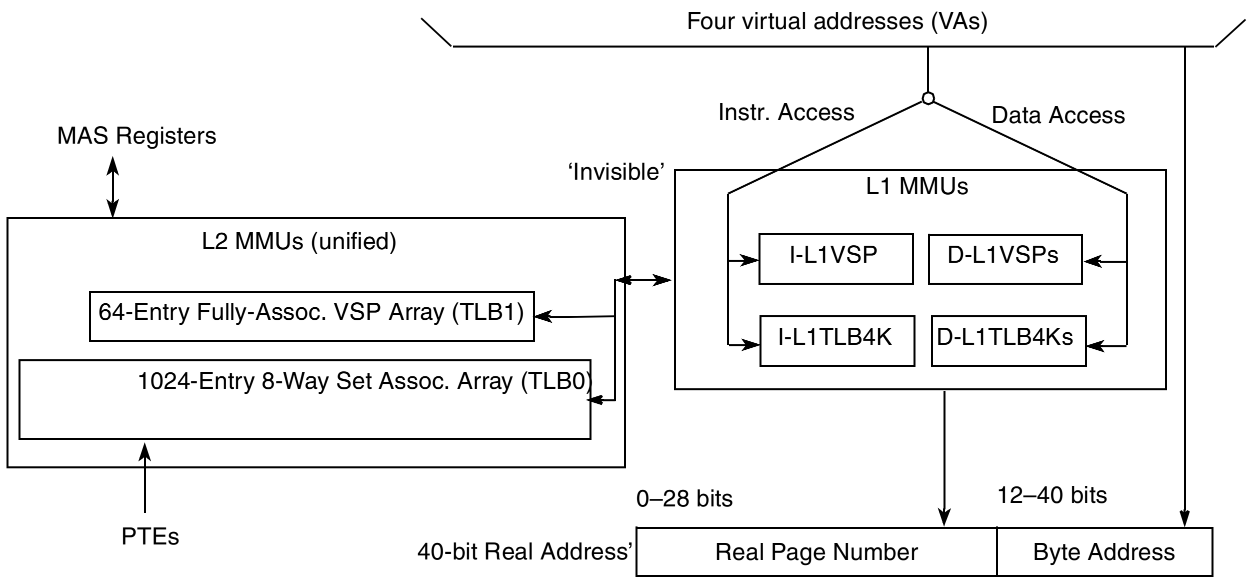


Figure 6-5. Two-level MMU structure

As this figure shows, 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 caches of the corresponding L2 MMU TLB entries. Conversely, the L2 MMU is managed by the TLB instructions by way of the MAS registers or through page table translations that load PTEs.

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.

This table lists the various TLBs and describes their characteristics.

Table 6-5. Index of TLBs

Location	Name	Page Sizes Supported	Associativity	Number of TLB Entries	Translations	Filled by
Instruction L1 MMU	I-L1VSP	Multiple page sizes [1]	Fully associative	8	Instruction	TLB1 hit
	I-L1TLB4K	4 KB	4-way	64	Instruction	TLB0 hit
Data L1 MMU (one set per thread)	D-L1VSP	Multiple page sizes [1]	Fully associative	8	Data	TLB1 hit
	D-L1TLB4K	4 KB	4-way	64	Data	TLB0 hit
L2 MMU	TLB1	Multiple page sizes [1]	Fully associative	64	Unified (I and D)	**tlbwe**
	TLB0	4 KB	8-way	1024	Unified (I and D)	**tlbwe or valid PTE**

[1] See Section 6.2.5, "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 6-5 and Table 6-5, the instruction and data L1 MMUs each implement an eight-entry, fully associative L1VSP array and a 64-entry, four-way set associative L1TLB4K array, comprising the following L1 MMU arrays:

- Instruction L1VSP—shared between threads; eight-entry, fully associative
- Instruction L1TLB4K—shared between threads; 64-entry, four-way set-associative
- Data L1VSP—per thread; eight-entry, fully associative
- Data L1TLB4K—per thread; 64-entry, four-way set-associative

As their names imply, L1TLB4K arrays support fixed, 4 KB pages, and L1VSP arrays support variable 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.

Unlike cores prior to the e6500 core, the L1 MMU entries are not an inclusive set of some entries in the L2 MMU. It is possible that a valid L1 MMU entry can exist where no corresponding L2 MMU entry exists. In particular, L1TLB4K may not be included in the TLB0 array if the hardware page table translation mechanism is used or software writes entries to TLB0 using the hardware entry select mechanism (MAS0[HES] = 1). Valid entries in L1VSP are always present in TLB1.

This figure shows the organization of the L1 TLBs in both the instruction and data L1 MMUs.

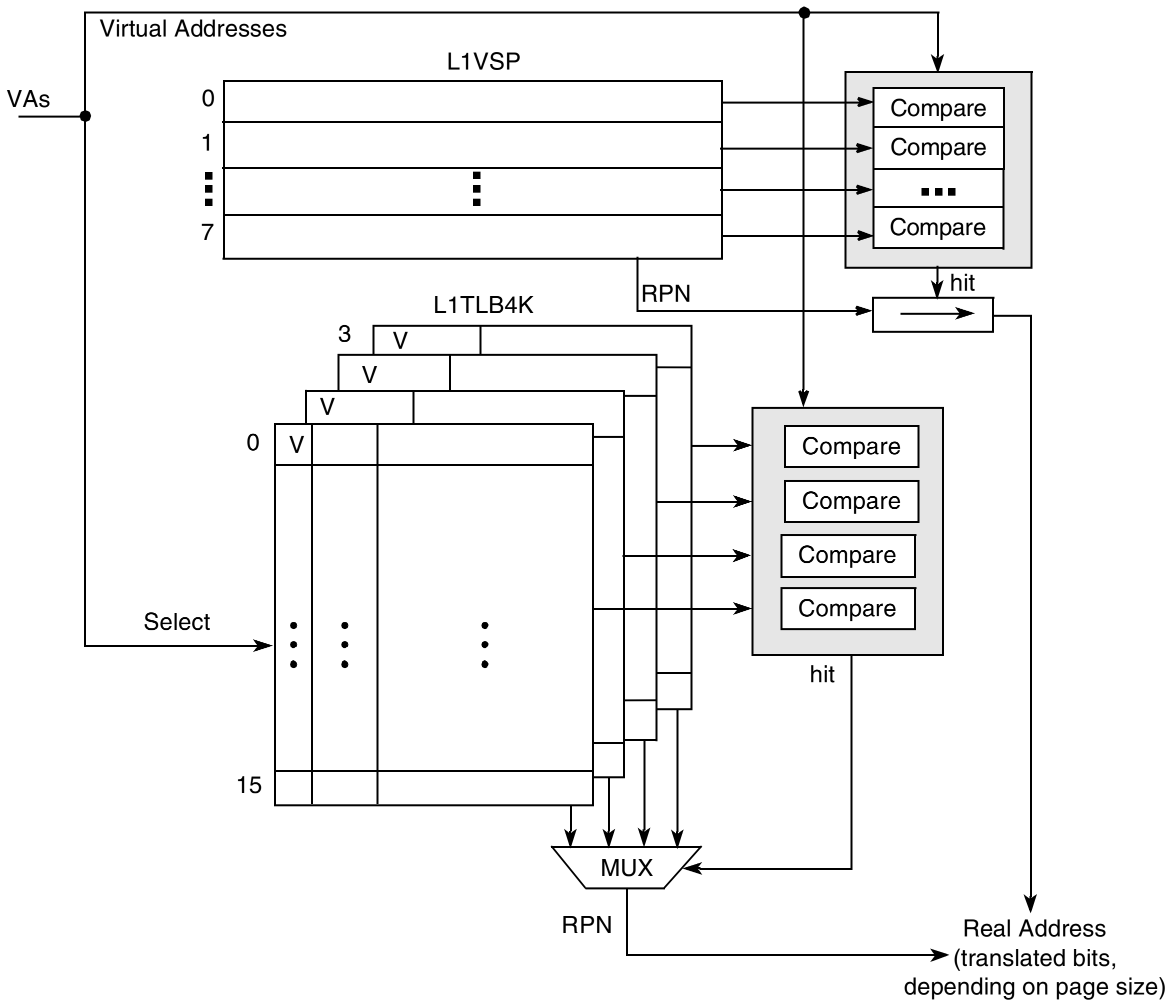


Figure 6-6. L1 MMU TLB organization

6.3.2 L2 TLB arrays

The level 1 MMUs are backed up by a unified L2 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 multiple page sizes.
- TLB0: 1024-entry, eight-way set associative array that supports only 4 KB page sizes.

This figure shows the L2 TLBs.

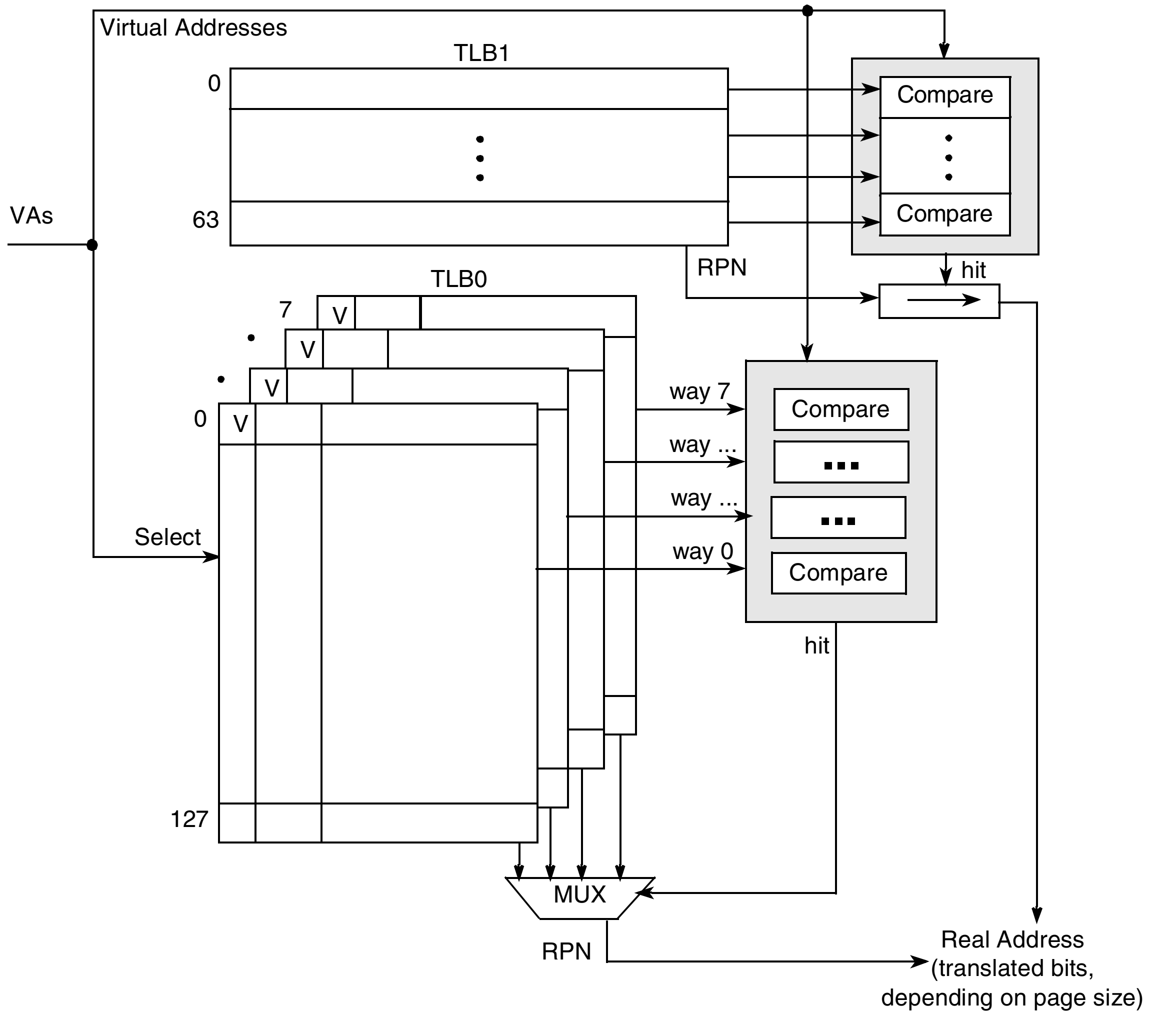


Figure 6-7. L2 MMU TLB organization

The L2 MMUs are shared between threads and shared between instruction fetches and data accesses.

6.3.2.1 Invalidation protection (IPROT) in TLB1

TLB1 entries with IPROT set can never be invalidated by a **tlbivax** or **tlbilx** instruction executed by this processor, by a **tlbivax** instruction executed by another processor, or by a flash invalidate initiated by writing to 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. This is done by using the MAS registers and executing the **tlbwe** instruction.

Only TLB entries in TLB1 can be protected from invalidation; entries in TLB0 cannot be protected because they do not implement IPROT. Software should assume that TLB0 entries are transient and can become invalid at any time.

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 with MAS0[HES] = 0. Note that successful invalidation operations in the L2 MMU also invalidate matching entries in the L1 MMU.

In general, software should avoid using **tlbwe** to invalidate entries in TLB0 and should instead rely on **tlbivax** or **tlbilx** to perform invalidations. The e6500 core is a multi-threaded processor that shares TLB0 among both threads. Each thread can either write to TLB0 with **tlbwe** or can have an entry loaded by the hardware due to a page table translation. In order to perform a precise invalidation with **tlbwe**, the other thread must not be allowed to load another entry that replaces the entry to be invalidated before the **tlbwe** has occurred. This requires software to disable the other thread while reading the TLB entry, setting its valid bit to 0 and writing it back to the TLB. In addition, writes to TLB0 that use MAS0[HES] = 1 do not back invalidate the victimized entry because software does not know which entry is being written, even if the entry is written with its valid bit as 0.

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 implemented by software and hardware. 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's value points to a way in the set that 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. When hardware allocates a new entry (through a successful page table translation or during **tlbwe** when MAS0[HES] = 1) in TLB0, it uses this value in the set to determine the way of the victim and updates the NV value after allocation to point to the next entry.
- MAS0[NV]—the next victim field of MAS0
- MAS0[ESEL]—selects the way to be replaced on **tlbwe**

Table 6-11 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.

Hardware also uses (and updates) NV in TLB0 when doing writes of TLB entries from PTE entries during a successful page table translation or during **tlbwe** if MAS0[HES] = 1.

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 e6500 core has an eight-way set-associative TLB0 and fully implements the round-robin scheme with a simple 3-bit counter that increments the 3-bit value of NV from the selected set of TLB0 entries on each TLB error interrupt 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.

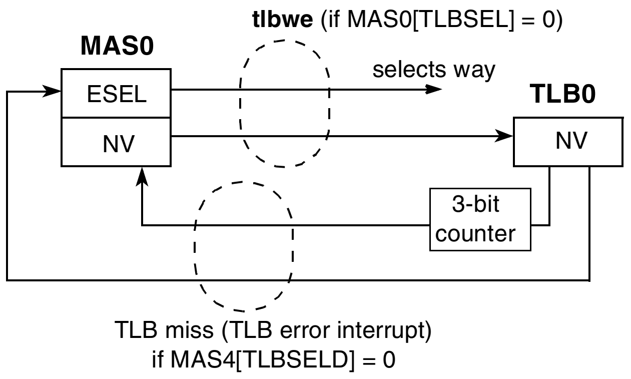

tlbwe (if MAS0[TLBSEL] = 0)
MAS0
ESEL
NV
selects way
TLB0
NV
3-bit counter
TLB miss (TLB error interrupt) if MAS4[TLBSELD] = 0

Figure 6-8. Round-robin replacement for TLB0

When **tlbwe** executes, MAS0[ESEL] selects the way of TLB0 to be loaded. If MAS0[TLBSEL] = 0 (selecting TLB0), TLB0[NV] is loaded with the MAS0[NV] value. When a TLB error interrupt occurs and 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**.

In general, software relies on page table translations to fill entries into TLB0; however, if software does do **tlbwe** instructions to TLB0 and is using page table translations, software should always allow hardware to select the entry to victimize by setting MAS0[HES] = 1 when performing **tlbwe**.

6.3.3 Consistency between L1 and L2 TLBs

The L1 TLBs are used to improve performance because they have a faster access time than the larger L2 TLBs. The relationship between TLBs is shown in the following figure.

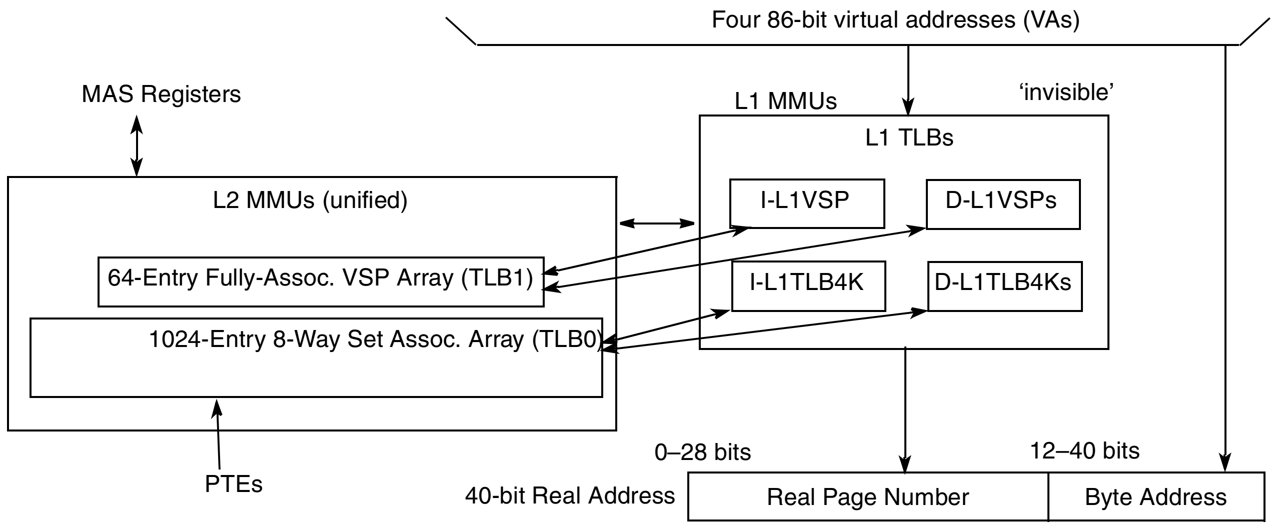


Figure 6-9. L1 MMU TLB relationships with L2 TLBs

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.

Valid entries in the L1TLB4K array may exist that have no matching valid entries in TLB0 if hardware page table translation is used or software writes to TLB0 using the hardware entry select mechanism (MAS0[HES] = 1). This is because back invalidations in the L1TLB4K are not performed for the victimized entry that is replaced due to a successful hardware page table translation or **tlbwe** to TLB0 with MAS0[HES] = 1.

NOTE

When any L2 TLB entry is invalidated through any invalidation mechanism or written with MAS0[HES] = 0, MAS1[V] = 0, and MAS0[TLBSEL] = 0, the corresponding entries in any L1 TLB will also be invalidated. (For the data L1 MMU, only the thread's private data L1 MMU will be invalidated.) Changing PID, LPIDR, EPLC, or EPSC or executing **tlbsx** may cause all instruction L1 entries to be invalidated and the thread's private data L1 MMU entries to be invalidated.

6.3.4 TLB entry field definitions

This table summarizes the fields of the e6500 TLB entries. These fields are defined by the architecture and described in detail in *EREF*.

Table 6-6. 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[0–13]	Translation ID—compared with PID. For external PID accesses, TID is compared with EPLC[EPID] or EPSC[EPID].
EPN[0–51]	Effective page number—compared with EA[0–51] for 4 KB pages
RPN[0–27]	Real page number—translated address RA[24–51] for 4 KB pages
SIZE[0–4]	Encoded page size. See Table 6-4. Only present in TLB1; however, software should always set page sizes for TLB0 for future compatibility.
UX,SX, UW,SW, UR,SR	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
IPROT	Invalidation protection—exists in TLB1 only
TGS	Translation guest space
VF	Virtualization fault. Identifies a page that always takes a data storage interrupt during data translation that is directed to the hypervisor, regardless of the setting of any other page attributes. Set by the hypervisor for pages associated with a device for which the hypervisor is providing a “virtual” device through emulation. Also identifies a page table that is in virtualized memory when it is set in a TLB entry that is an indirect entry (IND=1) causing both data accesses and instruction fetch accesses that perform page table translation through this entry to take a data storage or instruction storage interrupt directed to the hypervisor.
TLPID[0–5]	Translation logical partition ID
IND	Indirect bit—exists in TLB1 only. When set, this TLB entry is an indirect entry used to locate a page table.

6.4 LRAT concept

In a partitioned environment, a guest operating system is not allowed to manipulate (or even be aware of) real page numbers. Instead, the hypervisor virtualizes the real memory and the guest operating system manages the virtualized memory using logical page numbers (LPNs). The hypervisor prevents the guest operating system from seeing real page numbers by managing a logical-to-real address translation (LRAT) array. The LRAT is used to translate LPNs into RPNs when **tlbwe** is executed by the guest operating system or when a page table translation occurs from a guest virtual address. This avoids trapping to the hypervisor when the guest operating system writes a TLB entry or performs a page table translation and performs the necessary logical-to-real translation from what the guest operating system believes is the RPN to a true RPN. The **tlbwe** instruction is allowed to execute in the guest operating system, and page table translations can occur in the guest operating system with the true RPN written to the TLB entry using LRAT translation.

The hypervisor sets up translations in the LRAT providing the LPN to RPN translation by writing entries to the LRAT array using **tlbwe** and reading entries using **tlbre** with the ATSEL field in the MAS registers set to 1. If a guest operating system executes **tlbwe** and there is no matching LRAT translation, then an LRAT error interrupt occurs. If the TLB array that is being written to is ineligible for LRAT translation (for the e6500 core, any guest **tlbwe** that attempts to write TLB1 is ineligible), then an embedded hypervisor privilege exception occurs. In both of these cases, the hypervisor can write the appropriate LRAT entry or simply perform the **tlbwe** on behalf of the guest operating system, substituting the true RPN for the LPN in MAS3 and MAS7 before performing the **tlbwe**. The e6500 core supports LRAT translations for guest **tlbwe** execution or page table translation that writes to TLB0.

No analogous hardware translation mechanism exists for **tlbre** or **tlbsx**. If a guest operating system executes these instructions, an embedded hypervisor privilege interrupt occurs, and the hypervisor must emulate the **tlbre** or **tlbsx** and replace the resulting RPN with the guest's LPN in the MAS3 and MAS7 registers before returning to the guest operating system.

The e6500 core supports an eight-entry, fully associative LRAT.

6.4.1 LRAT entries

An LRAT entry consists of the fields listed in the following table.

Table 6-7. LRAT fields

Field	Description
V	Valid
LPID[0–5]	Logical partition ID value. Identifies the logical partition ID (LPID) value for this LRAT entry. LPID is compared with LPIDR during translation to help select an LRAT entry.
LPN[0–27]	Logical page number. Describes the logical address of the start of the page. The number of bits that are valid (used in translation) depends on the size of the page. For guest execution of **tlbwe**, LPN is compared to the RPN fields specified by the MAS registers (MAS7 and MAS3) under a mask based on the LSIZE field of the LRAT entry. For guest page table translations, LPN is compared to PTE[ARPN] under a mask based on the LSIZE field of the LRAT entry.
LSIZE[0–4]	Logical page size. Describes the size of the logical page of the LRAT entry. Logical page sizes are in powers of two, such that the size of the page is 2^{LSIZE} KB. Page sizes supported by the implementation are specified in LRATPS. Writing an LRAT entry with a a page size not supported by the LRATPS register sets LSIZE to 2 (4 KB).
LRPN[0–27]	LRAT real page number. For guest execution of **tlbwe**, bits 0:*n*-1 of LRPN are used to write the RPN field of a TLB entry instead of MASS3 and MAS7 (where *n* = 40-LSIZE). For guest page table translations, bits 0:*n*-1 of LRPN are used to write the RPN field of a TLB entry instead of PTE[ARPN] (where *n* = 40-LSIZE).

6.4.2 LRAT entry page size

Each LRAT entry has a page size associated with it. This defines the how many bytes this particular LRAT entry supports. The possible sizes that are supported is implementation dependent and is reflected in the associated LRATPS register. An LRAT entry page size is established by writing an LRAT entry with MAS1[TSIZE] set to a value that represents an LRAT page size. LRAT page sizes are defined as powers of 2 KB values, such that the size of an LRAT page is 2^{TSIZE}KB. Bits in the EPN and RPN fields associated with page offsets should be 0 based on LRAT page size.

This table shows the valid page sizes supported by the e6500 core. Note that the e6500 core only implements the low-order 28 bits of the architected 52-bit LRPN field of the LRAT.

Table 6-8. LRAT page sizes

TSIZE ($LRAT_{LSIZE}$)	Page Size	EPN and RPN Bits Required to be Zero when LRAT Written	RPN Written to TLB Entry on tlbwe LRAT hit (LRAT[LRPN] || MAS[RPN])
0b00010	4 KB	none	$LRPN_{0:27}$
0b00011	8 KB	51	$LRPN_{0:26}$ || RPN_{51}
0b00100	16 KB	50:51	$LRPN_{0:25}$ || $RPN_{50:51}$
0b00101	32 KB	49:51	$LRPN_{0:24}$ || $RPN_{49:51}$
0b00110	64 KB	48:51	$LRPN_{0:23}$ || $RPN_{48:51}$
0b00111	128 KB	47:51	$LRPN_{0:22}$ || $RPN_{47:51}$
0b01000	256 KB	46:51	$LRPN_{0:21}$ || $RPN_{46:51}$
0b01001	512 KB	45:51	$LRPN_{0:20}$ || $RPN_{45:51}$
0b01010	1 MB	44:51	$LRPN_{0:19}$ || $RPN_{44:51}$
0b01011	2 MB	43:51	$LRPN_{0:18}$ || $RPN_{43:51}$
0b01100	4 MB	42:51	$LRPN_{0:17}$ || $RPN_{42:51}$
0b01101	8 MB	41:51	$LRPN_{0:16}$ || $RPN_{41:51}$
0b01110	16 MB	40:51	$LRPN_{0:15}$ || $RPN_{40:51}$
0b01111	32 MB	39:51	$LRPN_{0:14}$ || $RPN_{39:51}$
0b10000	64 MB	38:51	$LRPN_{0:13}$ || $RPN_{38:51}$
0b10001	128 MB	37:51	$LRPN_{0:12}$ || $RPN_{37:51}$
0b10010	256 MB	36:51	$LRPN_{0:11}$ || $RPN_{36:51}$
0b10011	512 MB	35:51	$LRPN_{0:10}$ || $RPN_{35:51}$
0b10100	1 GB	34:51	$LRPN_{0:9}$ || $RPN_{34:51}$
0b10101	2 GB	33:51	$LRPN_{0:8}$ || $RPN_{33:51}$
0b10110	4 GB	32:51	$LRPN_{0:7}$ || $RPN_{32:51}$

Table 6-8. LRAT page sizes (continued)

TSIZE ($LRAT_{LSIZE}$)	Page Size	EPN and RPN Bits Required to be Zero when LRAT Written	RPN Written to TLB Entry on tlbwe LRAT hit (LRAT[LRPN] || MAS[RPN])
0b10111	8 GB	31:51	$LRPN_{0:6}$ || $RPN_{31:51}$
0b11000	16 GB	30:51	$LRPN_{0:5}$ || $RPN_{30:51}$
0b11001	32 GB	29:51	$LRPN_{0:4}$ || $RPN_{29:51}$
0b11010	64 GB	28:51	$LRPN_{0:3}$ || $RPN_{28:51}$
0b11011	128 GB	27:51	$LRPN_{0:2}$ || $RPN_{27:51}$
0b11100	256 GB	26:51	$LRPN_{0:1}$ || $RPN_{26:51}$
0b11101	512 GB	25:51	$LRPN_{0}$ || $RPN_{25:51}$
0b11110	1 TB	24:51	$RPN_{24:51}$

6.4.3 Reading and writing LRAT entries

LRAT entries may only be read or written in the hypervisor state (MSR[GS,PR] = 0b00). An LRAT entry can be read using **tlbre** and written using **tlbwe** with MAS0[ATSEL] = 1. In both cases, the LRAT entry is selected for reading and writing by setting the low-order 3 bits of MAS0[ESEL] to indicate the entry to read. Valid values for MAS0[ESEL] are from 0 to 7.

When an LRAT entry is read with **tlbre**, the MAS registers are set from the selected LRAT entry as follows:

- MAS1[V] is set to LRAT[V].
- MAS2[EPN] is set to $LRAT[LPN]_{0:27}$.
- MAS1[TSIZE] is set to LRAT[LSIZE].
- MAS3[RPN] (low order bits of the real page number) is set to $LRAT[LRPN]_{8:27}$, MAS7[RPN] (high order bits of the real page number) is set to $LRAT[LRPN]_{0:7}$.
- MAS8[TLPID] is set to LRAT[LPID].

When an LRAT entry is written with **tlbwe**, the selected LRAT entry is written from the MAS registers as follows:

- LRAT[V] is set to MAS1[V].
- $LRAT[LPN]_{0:27}$ is set to MAS2[EPN].
- LRAT[LSIZE] is set to MAS1[TSIZE].
- $LRAT[LRPN]_{8:27}$ is set to MAS3[RPN] (low-order bits of the real page number), $LRAT[LRPN]_{0:7}$ is set to MAS7[RPN] (high-order bits of the real page number).
- LRAT[LPID] is set to MAS8[TLPID].

6.4.4 Invalidating LRAT entries

LRAT entries can only be invalidated by writing the LRAT entry with MAS1[V] = 0.

6.4.5 LRAT translation

LRAT translation is performed when the guest operating system writes a TLB entry by executing **tlbwe** or a page table translation occurs during the translation of a guest virtual address. The following sections detail how LRAT translation is performed for the two separate cases.

6.4.5.1 LRAT translation during tlbwe

During guest execution of **tlbwe**, the RPN fields of MAS7 and MAS3 are considered to be a logical page number and must be translated by the LRAT into a real page number.

The RPN fields in MAS7 and MAS3 are compared against the LPN field of the LRAT entries under mask based on the LRAT entry page size. When a matching entry is found, the bits of the RPN fields of MAS7 and MAS3, based on the page size of the LRAT entry, are replaced with the corresponding bits of the LRPN field of the LRAT entry. Which bits are replaced are determined by the size of the matching LRAT entry in the same type of fashion that occurs during address translation when EA bits are replaced by bits in the RPN based on the matching TLB page size.

Note that the translation process does not change the RPN fields in MAS7 and MAS3. Rather, the translated real page number is written to the TLB entry.

The execution of **tlbwe** through LRAT translation is described as follows:

```
if $MSR_{GS}$ = 1 & $MSR_{PR}$ = 0 then                    // this is a guest TLB write
    if $MAS0_{TLBSEL}$ = 1 | $EPCR_{DGTMI}$then
        embedded hypervisor privilege exception (interrupt)
    else
        for n = 0 to $LRATCFG_{NENTRY}$
            if (LRAT[n]$_{V}$ = 0) | (LRAT[n]$_{LPID}$ != LPIDR) then
                next    // not valid or wrong partition, check next entry
            mask ← ~((1 << LRAT[n]$_{LSIZE}$-2) - 1)
            if (($MAS7_{RPN}$ || $MAS3_{RPN}$) & mask) = LRAT[n]$_{LPN}$ then
                rpn ← LRAT[n]$_{LRPN}$ | ($MAS7_{RPN}$ || $MAS3_{RPN}$) & ~mask)
                goto lratdone
        endfor
        LRAT error exception (interrupt)
else
    rpn ← $MAS7_{RPN}$ || $MAS3_{RPN}$// hypervisor write, no LRAT translation done

lratdone:
// value to write to $TLB_{RPN}$ is in rpn
// continue on with writing selected TLB entry..
```

It is a serious programming error for more than one LRAT translation to match any given value for the RPN fields in MAS7 and MAS3. The hypervisor can have such an occurrence trigger a machine check interrupt if HID0[EN_L2MMU_MHD] = 1.

6.4.5.2 LRAT translation during page table translation

LRAT translation is also performed when a guest memory access results in a page table translation. The abbreviated real page number bits of the ARPN field in the PTE are treated as a logical page number and are replaced with the LRAT real page number bits from the LRPN field of the selected LRAT entry. The selection process is based on a masked comparison of each LRAT entry LPN field with the ARPN field in the PTE. The LRAT real page number bits are concatenated with the logical page offset bits from the ARPN field in the PTE to form the real page number, which is written to a hardware selected TLB entry. Note that the page translation process does not change any fields in the PTE, but that the translated real page number is written to the TLB entry (*entry*).

The LRAT translation during page table translation is described as follows:

```
//
// assume a valid PTE is located.
if instruction fetch then
    gs ← $MSR_{GS}$
    lpid ← LPIDR
else if external PID load then
    gs ← $EPLC_{EGS}$
    lpid ← $EPLC_{ELPID}$
else if external PID store then
    gs ← $EPSC_{EGS}$
    lpid ← $EPSC_{ELPID}$
else        // normal load or store
    gs ← $MSR_{GS}$
    lpid ← LPIDR

if gs then                                     // this is a guest TLB tablewalk
    for n = 0 to $LRATCFG_{NENTRY}$
        if ($LRAT[n]_{V}$ = 0) | ($LRAT[n]_{LPID}$ != lpid) then
            next    // not valid or wrong partition, check next entry
        mask ← ~((1 << $LRAT[n]_{LSIZE}$-2) - 1)
        if ($PTE_{ARPN}$ & mask) = $LRAT[n]_{LPN}$ then
            $entry_{rpn}$ ← $LRAT[n]_{LRPN}$ | ($PTE_{ARPN}$ & ~mask)
            goto lratdone
    endfor
    LRAT error exception (interrupt)
else
    $entry_{rpn}$ ← $PTE_{ARPN[12:39]}$ // hypervisor tablewalk, no LRAT translation done

lratdone:
// value to write to $TLB_{RPN}$ is in rpn
```

It is a serious programming error for more than one LRAT translation to match any given value for a logical address to be translated. The hypervisor can have such an occurrence trigger a machine check interrupt if HID0[EN_L2MMU_MHD] = 1.

6.5 TLB instructions—implementation

As described in *EREF*, TLBs are accessed indirectly through MMU assist (MAS) registers or through PTE entries. 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 or LRAT into the MAS registers with a TLB Read Entry (**tlbre**) instruction, and data is written to the TLBs or LRAT 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.

6.5.1 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. Upon completion of the **tlbre**, the MAS registers contain the contents of the indexed TLB entry.

Selection of the TLB entry to read is performed by setting MAS0[ATSEL] = 0 and 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. MAS0[ATSEL] determines whether the **tlbre** instruction reads from the TLB or from the LRAT. See Section 6.4.3, "Reading and writing LRAT entries" for how LRAT reads are performed.

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, IND, 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 are undefined. The update of MAS registers as a result of a **tlbre** instruction is summarized in Table 6-11.

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 the e6500 core, 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 *EREF* for information at the architecture level.

For the e6500 core, MAS0[HES] is ignored when executing **tlbre**.

6.5.1.1 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.

6.5.2 TLB Write Entry (tlbwe) instruction

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.

To write a specific TLB entry, the entry to write is determined by setting MAS0[ATSEL] = 0, MAS0[HES] = 0, and 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 because the value in MAS0[ESEL] fully identifies the TLB entry. MAS0[ATSEL] determines whether the **tlbwe** instruction writes to the TLB or to the LRAT. See Section 6.4.3, "Reading and writing LRAT entries" for how LRAT writes are performed. When **tlbwe** is executed in the guest-supervisor state, the value of MAS0[ATSEL] is ignored and is always assumed to be 0.

To write an entry to TLB0 allowing the processor to select the entry using hardware entry select (HES), software should set MAS0[HES] = 1, MAS0[ATSEL] = 0, and MAS0[TLBSEL] and MAS2[EPN] values. In this case, hardware chooses the way using the stored NV value from the set selected by MAS2[EPN].

Hardware entry select is only available for TLB0 and for writes to TLB1; MAS0[HES] is ignored.

The selected entry is then written with following MAS fields: V, IPROT, TID, TS, TSIZE, EPN, WIMGE, U0—U3, X0, X1, TLPID, TGS, VF, IND, and permission bits. If **tlbwe** is executed in the hypervisor state, RPN from the MAS registers is used to write the TLB entry. If **tlbwe** is executed in the guest-supervisor state and EPCR[DGTMI] = 0, RPN from the MAS registers is used as a logical address and is translated to a real address through the LRAT. The resulting translated real address is written to the TLB entry. If the TLB array supports NV, it is written with the NV value if MAS0[HES] = 0. If executing in 32-bit mode, the written TLB entry has the upper 32 bits of the EPN field set to 0.

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. If the changes to the TLB need to be synchronized in the other thread, the other thread must be disabled and re-enabled after the context synchronizing operation has been performed in the thread that executed **tlbwe**.

If EPCR[DGTMI] is set, MAS0[TLBSEL] = 0, and **tlbwe** is executed in the guest-supervisor state, an embedded hypervisor privilege interrupt occurs and no TLB entry is written. If EPCR[DGTMI] is not set, MAS0[TLBSEL] = 0, **tlbwe** is executed in the guest-supervisor state, and. if a matching LRAT translation is not found, an LRAT error interrupt occurs. See Section 6.4.5, "LRAT translation."

Note that architecturally, if the instruction specifies a TLB entry that is not found, the results are undefined. However, for the e6500 core, the TLBSEL, ESEL and EPN fields always index to an existing L2 TLB entry, and that indexed entry is written. EPN bits are only used to index into TLB0. In the case of TLB1, the EPN field is unused for **tlbwe**. See *EREF* for additional architecture information.

6.5.2.1 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 equal 1, MAS0[ATSEL] must equal 0, 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.

Executing **tlbwe** with MAS0[TLBSEL] = 1 in the guest-supervisor state always causes an embedded hypervisor privilege exception. Writes to TLB1 are never translated through the LRAT.

Indirect entries can be written to the TLB1 array.

6.5.2.2 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 equal 0, MAS0[ATSEL] must equal 0, MAS0[ESEL] must point to the desired way (unless MAS0[HES] = 1), and EPN[45–51] in MAS2 must be loaded with the desired index. When **tlbwe** executes in hypervisor mode, the TLB entry information in MAS0–MAS3, MAS5, MAS7, and MAS8 is written into the selected entry in TLB0. When **tlbwe** executes in guest-supervisor mode, the TLB entry information in MAS3 and MAS7 is translated through the LRAT, and the other non-RPN fields of MAS0–MAS3, MAS5, MAS7, and MAS8 are written into the selected entry in TLB0 along with translated RPN.

Indirect entries cannot be written to the TLB0 array. The e6500 core always uses 0 for MAS1[IND] when writes to the TLB0 array are performed.

6.5.3 TLB Search (tlbsx) instruction—searching TLB1 and TLB0 arrays

Software can search the MMU by using **tlbsx**, which uses GS, LPIDR, IND, AS, and PID values 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, MAS6[SIND] with an IND 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 6.10.1, "MAS register updates," describes how MAS registers are updated.

NOTE

Note that **r**A = 0 is the preferred form for **tlbsx** and some Freescale implementations, including the e6500, take an illegal instruction exception program interrupt if **r**A!= 0.

6.5.4 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 $\text{entry}_{\text{TLPID}}$ = MAS5[SLPID] are invalidated.
- If T = 1, all TLB entries for which $\text{entry}_{\text{TLPID}}$ = MAS5[SLPID] and $\text{entry}_{\text{TID}}$ = MAS6[SPID] are invalidated.

- If T = 3, all TLB entries for which $\text{entry}_{\text{TLPID}}$ = MAS5[SLPID], $\text{entry}_{\text{TID}}$ = MAS6[SPID], $\text{entry}_{\text{IND}}$ = MAS6[SIND], $\text{entry}_{\text{TGS}}$ = MAS5[SGS], and ($\text{entry}_{\text{EPN}}$&*m*) = ($\text{EA}_{0:51}$&*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 that executes **tlbilx**. The **tlbilx** instruction invalidates TLB entries for all threads, but the results are only synchronized in the thread that executes **tlbilx**. Software should arrange to execute **tlbilx** on the other thread and perform appropriate synchronization.

EPCR[DGTMI] controls whether attempted execution of **tlbilx** causes an embedded hypervisor privilege interrupt when the processor is in the guest-supervisor state.

NOTES

The **tlbilx** instruction is the preferred way of performing TLB invalidations, especially for operating systems running as a guest to the hypervisor because 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.

The hypervisor should always write MAS5[SLPID] = LPIDR and MAS5[SGS] = 1 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.

6.5.5 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**.

Software should provide the address of a byte within a page to be invalidated as the EA (compared with EPN under mask based on page size) and also write the following MAS registers to further describe the virtual address of the TLB entry to be invalidated:

MAS5[SLPID]	the LPID value to match (compared with TLPID)
MAS5[SGS]	the GS value to match (compared with TGS)
MAS6[SPID]	the PID value to match (compared with TID)
MAS6[SAS]	the AS value to match (compared with TS)
MAS6[SIND]	the IND value to match (compared with IND)

Information about the invalidation from the **tlbivax** instruction and the MAS registers is encoded into a CoreNet transaction and is broadcast to all other processors in the coherence domain. Each processor in the coherence domain decodes the CoreNet transaction (including the processor that executed **tlbivax**) and performs the invalidation. All TLB entries that match the criteria above are invalidated, except entries for which IPROT is set.

Other TLB entries may be invalidated, but in no case will any TLB entries (including ones that match the invalidation criteria) with the IPROT attribute set be invalidated.

Because the virtual address can be much larger than the address available for the CoreNet transaction, a subset of the full virtual address is broadcast that fits within the space of the implemented physical addressing mode, which may result in a more generous invalidation.

NOTE

Note that **r**A = 0 is the preferred form for **tlbivax** and some Freescale implementations take an illegal instruction exception program interrupt if **r**A!= 0.

6.5.6 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 processors must acquire before executing **tlbsync**.

6.6 TLB entry maintenance—details

TLB entries can be loaded and maintained by the system software or can be loaded by hardware during a page table translation. The e6500 core provides some hardware assistance for these software tasks. Note that the system software cannot directly access the L1 TLBs. The L1 TLBs are completely and automatically maintained in hardware and are filled from the contents of the L2 TLBs during translation.

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 registers (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), LRAT error interrupt (on the data access), or permissions violation (DSI interrupt).
- Automatic loading into SRR0 of the EA of the instruction that causes a TLB miss interrupt, LRAT error interrupt, or a data storage interrupt.

- Automatic loading of the LPER with information about the PTE during a page table translation that caused an LRAT error interrupt.
- Automatic loading of a TLB entry into TLB0 as the result of a successful page table translation.
- 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 with MAS0[HES] = 0, 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 when software is controlling which entries are written. Additionally, the new TLB entry contained in MAS0–MAS3, MAS7, and MAS8 is written into the selected TLB.

Note that back invalidations are not performed in the L1 MMU when an L2 MMU entry is victimized by being written in the following cases:

- When an L2 MMU entry is written by hardware due to a page table translation that was successful
- When a TLB0 L2 MMU entry is written with **tlbwe** and MAS0[HES] = 1

In both cases, software does not control which entry is being victimized and, therefore, should not expect the victimized entry to be removed from the L1 MMU.

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 6.10.1, "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.

EREF provides more information on some actions taken on MMU exceptions.

6.7 TLB and LRAT states after reset

During reset, all LRAT, L1, and L2 MMU TLB entries are flash invalidated. Then entry 0 of TLB1 is loaded with the values shown in Table 6-9. 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 KB, 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 6-9. 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
TLPID	0	TLPID value for hypervisor page
TID	0	TID value for shared (global) page
EPN	0x00000000_FFFFF	Address of last 4 KB page in 32-bit address space. This does not include page offset bits—specifies the last 4 K page of 32-bit address space —EA of 0x00000000_FFFFF000.
RPN	0x00FFFFF	RPN. This does not include page offset bits —specifies the upper 28 bits of the 40-bit real address 0x00FFFFF000.
SIZE	0b00010	4 KB page size
SX/SR/SW	0b111	Full supervisor mode access allowed
UX/UR/UW	0b000	No user mode access allowed
WIMGE	0b01010	Caching-inhibited, noncoherent, 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
IND	0	Page is not an indirect entry

6.8 The G bit (of WIMGE)

The 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 e6500 core, 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 complete 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 begun 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 is returned to and re-executes performing the load operation again to both parts of the misaligned access.

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 designated as no-execute (with the UX/SX page permission bits cleared).

6.9 MMU parity detection and injection

6.9.1 TLB0 parity detection

The TLB0 array is equipped with parity detection. In general, software is notified of any parity errors that occur through a machine check interrupt. MCSR[TLBPERR] is set when a parity error is discovered. Software should log the error through the machine check handler (and clear the status in MCSR); however, the hardware will perform the following actions at the time the error is detected:

- If the error occurred during a **tlbre** or **tlbsx** instruction, the TLB entry read is returned as invalid (MAS0[V] = 0). The entire TLB0 array is invalidated. In the case of **tlbsx**, the MAS registers are set as if a **tlbsx** miss occurred.
- If the error occurred during a **tlbwe** instruction, the entry written and the rest of the TLB0 array is invalidated.
- If the error occurred during an invalidation (from a **tlbilx** or **tlbivax**), the entire TLB0 array is invalidated.
- If the error occurred during translation, the load, store, or instruction fetch takes an error report exception. The entire TLB0 array is invalidated.

Parity detection (and resulting invalidations) in TLB0 assume that software treats TLB0 as an array that can be invalidated at any time and that such invalidations do not cause errors because software reloads the TLB entries in the miss handler from a page table in memory. Most modern operating systems treat TLB0 as such.

There is no parity detection for the TLB1 array.

6.9.2 TLB0 parity injection

Parity errors can be injected into the TLB0 array for the purposes of testing error detection and recovery. If software writes MMUCSR0[TLB_EI] = 1, then parity associated with a TLB0 entry is inverted (set to indicate a parity error) when the TLB0 entry is written. In this case, a parity error is not detected on the **tlbwe** operation when injection is performed. To test TLB0 parity detection, software should do the following:

- Ensure the other thread is disabled by writing to TENC then polling TENSR to see that the thread is disabled. This prevents the other thread from accessing the MMU.
- Ensure the testing code and any data references it may make are running in addresses mapped by TLB1 only.
- Block any interrupts to prevent the test code from being interrupted.
- Write MMUCSR0[TLB_EI] = 1. Perform an **isync**.
- Perform a **tlbwe** operation that creates a test TLB entry for which the address mapping can be accessed from the current state, which is used to test the parity detection.
- Perform a **sync** followed by an **isync**.
- Execute an instruction that accesses the test TLB entry.
- A machine check should occur on the parity detection.
- After testing is completed, software should write MMUCSR0[TLB_EI] = 0 and perform an **isync**.

6.10 MMU registers

This table provides cross-references to sections with more detailed bit descriptions for the e6500 core MMU registers. *EREF* lists the architecture definitions for these registers.

Table 6-10. Registers used for MMU functions

Registers	Section/Page
Process ID (PID) register	2.13.2/2-62
Logical Partition ID (LPIDR) register	2.13.1/2-62
LRAT Configuration (LRATCFG) register	2.13.8/2-67
LRAT Page Size (LRATPS) register	2.13.9/2-69
Logical Page Exception (LPER/LPERU) register	2.9.3/2-30
MMU Control and Status (MMUCSR0) register	2.13.3/2-62
MMU Configuration (MMUCFG) register	2.13.4/2-63
TLB configuration registers (TLB0CFG–TLB1CFG)	2.13.5/2-64
TLB page size registers (TLB0PS–TLB1PS)	2.13.6/2-66
Embedded Page Table Configuration (EPTCFG) register	2.13.7/2-66
MMU assist registers (MAS0–MAS8)	2.13.10/2-69
Data Exception Address (DEAR/GDEAR) register	2.9.2/2-30
External PID Load Context (EPLC) register	2.13.11.1/2-79
External PID Store Context (EPSC) register	2.13.11.2/2-80

6.10.1 MAS register updates

The following table 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 that 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.

Note that MAS registers are not updated when an LRAT error interrupt occurs.

Table 6-11. MMU assist register field updates

MAS Field	Inst TLB Error Data TLB Error	tlbsx Hit	tlbsx Miss	tlbre
MAS0				
ATSEL	0	0	0	—

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
				—

HES	if TLBSELD = 0
	  1]
	else
	  0
		0	if TLBSELD = 0
			  1]
			else
			  0
				—

NV	if TLBSELD = 0
	  mod(TLB0[NV]+1,8)
	else
	  0
		if TLBSEL = 0
		  TLB0[NV]
		else
		  0
			if TLBSELD = 0
			  mod(TLB0[NV]+1,8)
			else
			  0
				On TLB read (ATSEL = 0):
				if TLBSEL = 0
				  TLB0[NV]
				else
				  0
				On LRAT read (ATSEL = 1):
				unchanged

MAS1				

IPROT	0	If TLB1 hit
		  TLB1[IPROT]
		else
		  0
			0	***On TLB read (ATSEL = 0):***
				If TLB1 hit
				  TLB1[IPROT]
				else
				  0
				On LRAT read (ATSEL = 1):
				0

TID	if ext PID load
	  EPLC[EPID]
	elseif ext PID store
	  EPSC[EPID]
	else
	  PID
		TLB[TID]	SPID	***On TLB read (ATSEL = 0):***
				TLB[TID]
				On LRAT read (ATSEL = 1):
				0

Table 6-11. MMU assist register field updates (continued)

MAS Field	Inst TLB Error Data TLB Error	tlbsx Hit	tlbsx Miss	tlbre

TSIZE	TSIZED	if TLB1 hit
		  TLB1[TSIZE]
		else
		  2
			TSIZED	***On TLB read (ATSEL = 0):***
				if TLB1 hit
				  TLB1[TSIZE]
				else
				  2
				On LRAT read (ATSEL = 1):
				LRAT[LSIZE]

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	***On TLB read (ATSEL = 0):***
				TLB[TS]
				On LRAT read (ATSEL = 1):
				0

V	1	1	0	***On TLB read (ATSEL = 0):***
				TLB[V]
				On LRAT read (ATSEL = 1):
				LRAT[V]

IND	INDD	TLB[IND]	INDD	***On TLB read (ATSEL = 0):***
				if TLB1 hit
				  TLB[IND]
				else
				  0
				On LRAT read (ATSEL = 1):
				0

MAS2				

WIMGE	WIMGED	TLB[WIMGE]	WIMGED	***On TLB read (ATSEL = 0):***
				TLB[WIMGE]
				On LRAT read (ATSEL = 1):
				0

X0, X1	X0D, X1D	TLB[X0, X1]	X0D, X1D	***On TLB read (ATSEL = 0):***
				TLB[X0,X1]
				On LRAT read (ATSEL = 1):
				0

EPN[0:31]	EA[0:31] of access
	
	Note: if MSR[CM] = 0, then EA[0:31] must be 0.
		if MSR[CM] = 0
		  unchanged
		else
		  TLB[EPN[0:31]]
			EA[0:31] of access
			
			Note: if MSR[CM] = 0, then EA[0:31] must be 0.
				On TLB read (ATSEL = 0):
				if MSR[CM] = 0
				  unchanged
				else
				  TLB[EPN[0:31]]
				On LRAT read (ATSEL = 1):
				LRAT[LPN[0:31]]

Table 6-11. MMU assist register field updates (continued)

MAS Field	Inst TLB Error Data TLB Error	tlbsx Hit	tlbsx Miss	tlbre

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] of access	***On TLB read (ATSEL = 0):***
				if TLBSEL = 0
				TLB[EPN[32:44]] || EPN[45:51]
				else
				TLB[EPN[32:51]]
				On LRAT read (ATSEL = 1):
				LRAT[LPN[32:51]]

MAS3				
UR,SR,UW, SW,UX,SX	Zeros	if SIND = 0 TLB[UR,SR,UW,SW,UX,SX] else see SPSIZE below Note: SR is always set to TLB[SR] regardless of whether the read entry is an indirect entry	Zeros	***On TLB read (ATSEL = 0):*** f SIND = 0 TLB[UR,SR,UW,SW,UX,SX] else see SPSIZE below Note: SR is always set to TLB[SR] regardless of whether the read entry is an indirect entry ***On LRAT read (ATSEL = 1):*** 0

SPSIZE	Zeros	if SIND = 1
		2
		else
		see PERMIS above
			Zeros	***On TLB read (ATSEL = 0):***
				if SIND = 1
				TLB[SPSIZE]
				else
				see PERMIS above
				On LRAT read (ATSEL = 1):
				0

U0–U3	Zeros	TLB[U0-U3]	Zeros	***On TLB read (ATSEL = 0):***
				TLB[U0-U3]
				On LRAT read (ATSEL = 1):
				0]

RPN[32:51]	Zeros	TLB[RPN[32:51]]	Zeros	***On TLB read (ATSEL = 0):***
				TLB[RPN[32:51]]
				On LRAT read (ATSEL = 1):
				LRAT[LRPN[32:51]]

MAS4				
WIMGED	—	—	—	—
WIMGED, X0D,X1D, TIDSELD, TLBSELD, TSIZED	—	—	—	—
INDD	—	—	—	—
MAS5				
SGS	—	—	—	—
SLPID	—	—	—	—

Table 6-11. MMU assist register field updates (continued)

MAS Field	Inst TLB Error Data TLB Error	tlbsx Hit	tlbsx Miss	tlbre
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
		—	—	—

SIND	INDD	—	—	—
MAS7				

RPN[24:31]	Zeros	TLB[RPN[24:31]]	Zeros	***On TLB read (ATSEL = 0):***
				TLB[RPN[24:31]]]
				On LRAT read (ATSEL = 1):
				LRAT[LRPN[24:31]]

MAS8				

TGS	—	TLB[TGS]	—	***On TLB read (ATSEL = 0):***
				TLB[TGS]
				On LRAT read (ATSEL = 1):
				0

VF	—	TLB[VF]	—	***On TLB read (ATSEL = 0):***
				TLB[VF]
				On LRAT read (ATSEL = 1):
				0

TLPID	—	TLB[TLPID]	—	***On TLB read (ATSEL = 0):***
				TLB[TLPID]
				On LRAT read (ATSEL = 1):
				LRAT[LPID]

Chapter 7
Timer Facilities

This chapter describes specific details of the e6500 core 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 *EREF*.

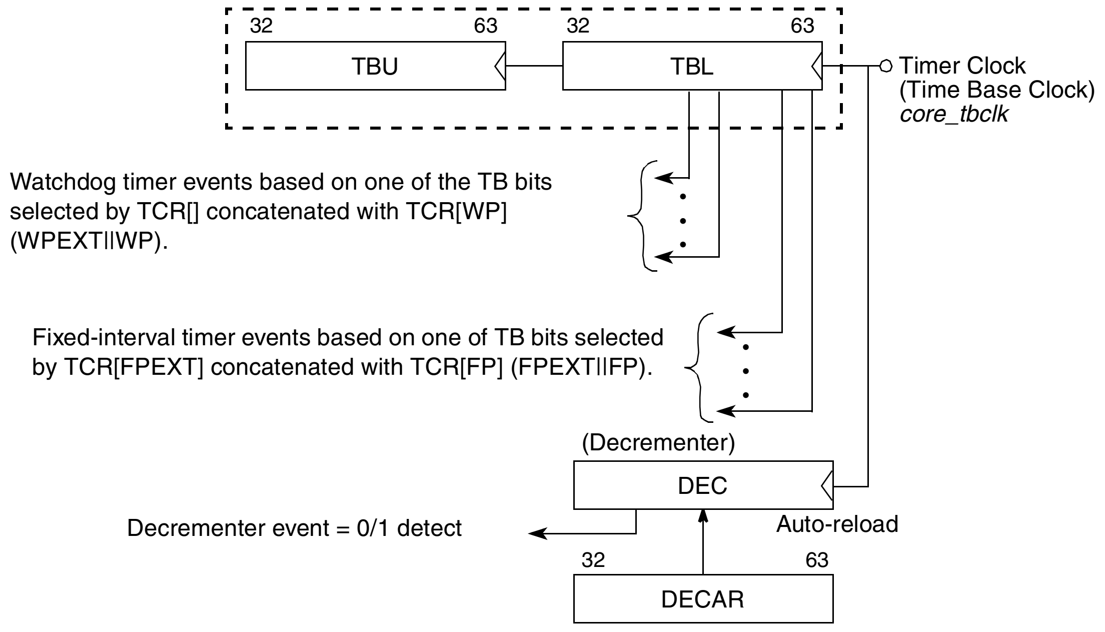
7.1 Timer facilities

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 TB is shared by all of the threads in the core.
- The DEC, 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.

  Each thread has a private DEC that 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 FIT is essentially a selected bit of the TB that provides a means of signaling an exception whenever the selected bit transitions from 0 to 1 in a repetitive fashion. The FIT is typically used to trigger periodic system maintenance functions. Software may select any bit in the TB to serve as the FIT. Each thread has a private FIT.
- The ATB provides a 64-bit timer that cannot be written and that increments at an implementation-dependent frequency. For the e6500 core, the ATB frequency is the same as the core frequency, which makes the ATB useful for measuring elapsed time in core clocks. The ATB is shared by all of the threads in the core.
- The watchdog timer is also a selected bit of the TB that provides a means of signaling 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. Each thread has a private watchdog timer.

The relationship of these timer facilities (except for the ATB) to each other is shown in the following figure.



32
63
TBU
32
63
TBL
Timer Clock
(Time Base Clock)
core_tbclk
Watchdog timer events based on one of the TB bits selected by TCR[] concatenated with TCR[WP] (WPEXT||WP).
Fixed-interval timer events based on one of TB bits selected by TCR[FPEXT] concatenated with TCR[FP] (FPEXT||FP).
(Decrementer)
DEC
Decrementer event = 0/1 detect
Auto-reload
32
63
DECAR

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 (TCR) register—Provides control information for the timers of a thread. TCR controls decrementer, fixed-interval timer, and watchdog timer options. Each thread has a private TCR.

  Section 2.8.1, "Timer Control (TCR) register," describes TCR in detail.
- Timer Status (TSR) register—Contains status on timer events and the most recent watchdog timer-initiated processor reset. Section 2.8.2, "Timer Status (TSR) register," describes TSR in detail. Each thread has a private TSR.
- Decrementer (DEC) register—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 (DEC) register," for more information. Each thread has a private DEC.
- Decrementer Auto-Reload (DECAR) register—Supports the auto-reload feature of the decrementer. The DECAR contents cannot be read. See Section 2.8.5, "Decrementer Auto-Reload (DECAR) register," for more information. Each thread has a private DECAR.

7.3 Watchdog timer implementation

When the watchdog timer expires in such a manner as requiring a reset, neither the thread nor the core performs the reset. Instead, the core output signals *core*x*_wrs_thrd*y[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 thread; 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 e6500 core 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 e6500 core and cluster. The scope of this chapter is limited to the features of the e6500 core and cluster only. Additional power management capabilities associated with an integrated device that contains one or more e6500 clusters are documented in the integrated device's reference manual.

8.1 Overview

Power management for each e6500 thread (processor) is part of a larger core-, cluster-, and integrated-device-based power management paradigm. An e6500 cluster includes from one to four cores, each with two threads, all interfacing to a shared L2 cache. Some thread-specific or core-specific power management functions are handled by the cluster, both because the cluster contains much of the interface to the rest of the integrated device, and because the cluster has a separate power domain from any of the cores in the cluster. This eases management of power within a thread or core because functions such as time-base, CoreNet interface, and processor messaging can remain active while a core is in a low power state. As part of the cluster, the L2 cache, which contains all the coherent modified states of the all of the core caches in the cluster, can maintain memory coherency regardless of the power management state of any thread or core.

A complete power management scheme for a system that includes an e6500 cluster requires the support of the integrated device. The programming model and control of power management states for a thread or core is provided by the integrated device. With the exception of the **wait** instructions and associated PW* power management states, all other thread and core power management states are achieved through registers provided by the integrated device. The e6500 cluster provides a signal interface that an integrated device can use to transition an e6500 core or cluster between different power management states.

8.2 Power management signals

Table 8-1 summarizes the power management signals of the e6500 cluster. Power management signals are used by the integrated device and the core cluster to command a core to enter or leave specific activity states that affect how much power is being consumed by the core. Core Activity States (*PHnn* and *PWnn*) are described in Table 8-2. All threads within a core must either be disabled or signaled to transition to a core activity state before the entire core will enter the directed core activity state. Entry into and exiting from some core activity states is managed entirely within the core. Thus, some transitions and states may not have power management signals associated with them. How these signals are generated from the integrated device is defined in the integrated device reference manual.

Table 8-1. Cluster power management signals (per core)

Core Cluster Signal (*x* specifies core number, *y* specifies thread number)	I/O	Signal Description
core***x***_halt_thrd***y***	I	Asserted by the integrated device to initiate thread actions that cause the core to enter the *PH10* state. Disabled threads also receive and respond to this signal.
core***x***_halted_thrd***y***	O	Asserted by the thread when it reaches the *PH10* state.
core***x***_stop	I	Asserted by the integrated device to initiate the required actions that cause the core to go from the *PH10* into the *PH15* state (as described in Table 8-2). All threads in the core must be in the *PH10* state in order to enter the *PH15* state. Negating *core**x**_stop* returns the core to the *PH10* state.
core***x***_stopped	O	Asserted by the core anytime the internal functional clocks of the core are stopped. (For example, after the integrated device asserts *core**x**_stop*.)
core***x***_pg_sr [core_pg_sr]	I	Asserted by the integrated device to initiate the required actions that cause the core to go from the *PH15* into the *PH20* state (as described in Table 8-2). The core must be in the *PH15* state prior to the assertion of this signal. Asserting core***x***_pg_sr causes the e6500 core to enter the power-gated state-retained state (*PH20*). A core indicates that it is in the *PH20* state by asserting core***x***_sh20. After the core asserts core***x***_sh20, the integrated device may deassert core***x***_pg_sr and may not deassert core***x***_stop until after core***x***_sh20 is deasserted. Negating core***x***_pg_sr returns the core to the *PH15* state. Once asserted, *core**x**_pg_sr* must not be negated until after the core has entered the *PH20* state; otherwise, the negation may not be recognized. For power management purposes, *core**x**_pg_sr* must be asserted only while the core is in the *PH15* state.
core***x***_ph20 [core_ph20]	O	Asserted by the core when it has reached the *PH20* state. Indicates to the integrated device that the core is in a power-gated state-retained state.
core***x***_off [core_off]	I	Asserted by the integrated device to power off the core. The core must be in the *PH20* state prior to asserting this signal.
core***x***_static_off [core_static_off]	I	Asserted by the integrated device to power off the core. The is a static signal that must be driven continuously to the same value and may only be changed at the start of a POR reset.

Table 8-1. Cluster power management signals (per core) (continued)

Core Cluster Signal (*x* specifies core number, *y* specifies thread number)	I/O	Signal Description
core***x***_pw20 [core_pw20]	O	Asserted by the core when it reached the *PW20* state. Indicates to the integrated device that the core is in a power-gated, state-retained state.
core***x***_tben	I	Asserted by the integrated device to enable the timebase.[1]
core***x***_wake_req_thrd***y***	O	Asserted when the thread detects an internally generated asynchronous interrupt is enabled and pending. This prompts the integrated device to bring the thread to a *PH00* activity state to service the interrupt. The interrupts that can cause the assertion of *core**x**_wake_req_thrd**y*** 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.
cluster_stop	I	Asserted by the integrated device to initiate the required actions that cause the cluster to go from *PCL00* into the *PCL10* state (as described in Table 8-2). Negating *cluster_stop* returns the cluster to the *PCL00* state if the cluster is powered up.
cluster_stopped	O	Asserted by the cluster anytime the internal functional clocks of the e6500 core are stopped. (For example, after the integrated device asserts *cluster_stop*.)

[1] The *core**x**_tben* pin is provided to be compatible with previous designs. It is strongly recommended that this pin always be asserted to guarantee software can maintain common timebase values across multiple cores and clusters (software requirement.).

8.3 Core power management states

Core power management states are called core activity states. This helps distinguish them from historical semantically overloaded terms. Core activity states determine what parts of the core are powered; thus, how much power is consumed by the core depends on the core activity state.

State transitions between core activity states are controlled through instruction execution and power management signals from the cluster and the integrated device. See Section 8.2, "Power management signals," for more details. Power management signals from the cluster and the integrated device are generated when software writes the power management control registers in the integrated device or when the integrated device determines that an asynchronous interrupt is pending.

Pending asynchronous external interrupts that include:

- external input (normal interrupts from devices in the integrated device);
- critical input;
- NMI;
- machine check (from sources on the integrated device through the *core**x**_mcp_thrd**y*** core signal)

are sensed by the interrupt controller of the integrated device.

Pending asynchronous internal interrupts that include:

- Decrementer (when TSR[DIE] = 1);

- Fixed interval timer (when TSR[FIE] = 1);
- Watchdog timer (when TSR[WIE] = 1);
- Performance monitor;
- Machine check (from internal sources);
- Processor doorbells (i.e., processor doorbell, processor doorbell critical, guest processor doorbell, guest processor doorbell critical, or guest processor doorbell machine check)

are sensed by the core or the e6500 cluster and are sent as the *core**x**_wake_req_thrd**y*** signal to the integrated device.

Generally, a core activity state (other than *PH00,* which is the normal running state) is entered when software writes the power management control registers in the integrated device or executes a **wait** instruction. Similarly, a core activity state other than *PH00* is exited, returning the core back to the normal running *PH00* state when a pending asynchronous interrupt is present. If the present core activity state is the result of **wait** instructions in all threads (that is, *PW10* or *PW20*), one of the following conditions terminates the wait and transitions the core to the normal running *PH00* state:

- An enabled asynchronous interrupt is pending. Note that some power management states disable performance monitor event counting, which precludes the use of the performance monitor asynchronous interrupt as a means to terminate the wait and transition the core to the *PH00* state.
- **wait** was executed with WC=1 (wait for reservation to clear), and the reservation held by the processor is cleared.
- A cache stash is received by the core.

This table describes the core activity states.

Table 8-2. Core activity (power management) states

Core Activity State	SoC PM State	Architectural State Retained	Architectural State not Retained	Description
PH00 (default)	—	All	None	Full-On with 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. All architectural states are retained. Exited when a **wait** instruction executes, which causes the core to enter the *PW10* or the *PW20* state or when *core**x**_halt_thrd**y*** is asserted by the integrated device because software has commanded a new power management state by writing the power management registers in the integrated device. All threads (processors) in a core must be either waiting or disabled for the core to enter the *PW10* or *PW20* states.
PH10	*doze*	All	None	Initiated by either asserting the *core**x**_halt_thrd**y*** inputs for all threads. The thread responds by stopping instruction execution. It then asserts the *core**x**_halted_thrd**y*** output to indicate that the thread is prepared for the core to enter the *PH10* state. Core clocks continue running, and snooping continues to maintain cache coherency. When all enabled threads within the core have signaled *core**x**_halted_thrd**y***, the core enters the *PH10* state. The following occur once the core is in the *PH10* state: • Suspend instruction fetching. • Complete all previously fetched instructions and associated data transactions. The thread accomplishes this by performing the semantics of a **sync 0** instruction after execution has stopped and then waits for the **sync 0** semantics to finish. This ensures that all storage accesses initiated by this processor have been completed. Exited when *core**x**_halt_thrd**y*** is deasserted by the integrated device. This occurs when an asynchronous internal or external interrupt is pending or when software from some other processor changes the power management state for a thread in the core.
PH15	*nap*	GPR, SPR, PMR, FPR, VR, BTB, Time base related functions (TB, DEC, FIT, Watchdog), L1 instruction cache[1], L1 data cache[1], L1 TLB[1], L2 TLB[1]		Initiated when *core**x**_stop* is asserted by the integrated device to the core while it is in the *PH10* state. All threads must be in the *PH10* state in order to enter the *PH15* state. The core responds by inhibiting clock distribution to most of its functional units, and then asserting the *core**x**_stopped* output. Exited when *core**x**_stop* is deasserted by the integrated device. The core responds by exiting the *PH15* state and entering the *PH10* state. Clock distribution that was disabled by the *PH15* state is restored. Note that the Alternate Time Base (ATB) of a core is not incremented when the core is in the *PH15* state because most core clocks are not running.

Table 8-2. Core activity (power management) states (continued)

Core Activity State	SoC PM State	Architectural State Retained	Architectural State not Retained	Description
PH20	*pg_sr*	GPR, SPR, PMR, FPR, VR, BTB, Time base related functions (TB, DEC, FIT, Watchdog), L1 instruction cache[1], L1 data cache[1], L1 TLB[1], L2 TLB[1]		Initiated when *core**x**_pg_sr* is asserted by the integrated device to the core while it is in the *PH15* state. The core responds by entering a power-gated state-retained state. Exited when *core**x**_pg_sr* is deasserted by the integrated device. The core responds by exiting the *PH20* state entering the *PH15* state.
PH30	*off*	Time base	All state except for Time base	Initiated when *core**x**_off* or *core**x**_static_off* is asserted by the integrated device to the core while it is in the *PH20* state. The core responds by turning off all input voltages. Exited when *core**x**_off* or *core**x**_static_off* is deasserted by the integrated device. The core responds by exiting the *PH30* state by performing a reset and entering the *PH00* state. When exiting from *PH30* via reset, all states other than the Time Base are set to POR values. TB is not controlled by core power management states and continues to increment unless the e6500 cluster is powered off. Some revisions of the e6500 core do not switch the power gating switches to off voltage, but instead switch them to state retention voltages (as done in *PH20*). Software should not depend on this behavior and future versions of the processor (as well as other future processors) are likely to switch the power gating switches to off voltage.

Table 8-2. Core activity (power management) states (continued)

Core Activity State	SoC PM State	Architectural State Retained	Architectural State not Retained	Description
PW10	—	All	None	Initiated when a **wait** instruction is executed by all active threads in the core and any conditions specified for the **wait** instructions (e.g., **wait** WC = 1, and the thread's reservation is set). Core clocks continue running, and snooping continues to maintain cache coherency. *PW10* is independent of power management signaling from the integrated device. Instruction fetching is suspended when the thread is in the *PW10* state. Exited when one of the following is true: • An enabled asserting asynchronous interrupt is ready to be taken by the thread. • If *PW10* was entered with **wait** WC = 1 when the reservation of the thread is cleared. For example, a store to the reserved coherency granule is performed by another processor. • A cache stash is received by the core. When one of these conditions occurs, the core returns to *PH00.* If *PW10* is exited due to a stash, all threads of the core exit **wait**. However, only the thread (processor) targeted by the interrupt or whose reservation has been cleared exits **wait.** Assertion of the *core**x**_halt_thrd**y*** signal due to either power management (that is, *core**x**_halt_thrd**y*** pin asserted) or debug halt, causes the machine to transition to *PH10*.

Table 8-2. Core activity (power management) states (continued)

Core Activity State	SoC PM State	Architectural State Retained	Architectural State not Retained	Description
PW20	—	GPR, SPR, PMR, FPR, VR, L1 TLB, L2 TLB, BTB, Time base related functions (TB, DEC, FIT, Watchdog), L1 data and instruction cache locks	L1 data cache[2], L1 instruction cache[3]	Initiated when the all threads of a core are in the *PW10* state, PWRMGTCR0[PW20_WAIT] = 1, and any of the following is true: • **wait** was executed with WH = 1. • *PW20* entry timer expires as specified by PWRMGTCR0[PW20_ENT_P]. The following occur once the core is in the *PW20* state: • A **sync 0** is performed. • L1 data cache is invalidated. L1 instruction cache is also invalidated if PWRMGTCR0[PW20_INV_ICACHE] is set. • Core clocks turned off. *PW20* is exited and returns to *PH00*, *PW10*, or *PH10* when any of the following occur: • A stash to the core's L1 data cache. The core enters *PW10*, then transitions to *PH00*. • If *PW10* or *PW20* states are entered with **wait** WC=1 and the reservation of the waiting thread is cleared. The core enters *PW10,* then transitions to *PH00*. • An **icbi** instruction from another processor is snooped and PWRMGTCR0[PW20_INV_ICACHE] is not set. The core enters *PW10* and processes the **icbi**. • A **tlbivax** instruction from another processor is snooped. The core enters *PW10* and processes the **tlbivax**. • An enabled asserting asynchronous interrupt is ready to be taken by the core. The core enters *PW10*. If, in *PW10,* the asynchronous interrupt is both still asserting and is enabled, the core enters *PH00.* [4] • Assertion of the *core**x**_halt_thread**y*** signal, due to either power management (*core**x**_halt_thread**y*** pin asserted) or debug halt, causes the machine to transition to *PH10*. The core will return to *PW10* and then to *PH10*. Upon exit from *PW20,* the following occurs: • Core clocks are turned on. • If a pending asynchronous interrupt caused the exit, the core returns first to *PW10*, and subsequently to *PH00* state if the interrupt is both still asserting and is enabled. The core returns only to *PW10* if the interrupt is no longer asserting or is not enabled. • If a pending asynchronous interrupt caused the exit and is subsequently taken, the core begins re-execution at the interrupt vector. • If the reservation clearing or a cache stash caused the exit, the waiting thread of the core begins re-execution at the instruction following the **wait**. Note that during *PW20*, ATB is not incremented because most core clocks are not running.

Table 8-2. Core activity (power management) states (continued)

Core Activity State	SoC PM State	Architectural State Retained	Architectural State not Retained	Description
—	—	—	—	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 the *PH20* state. For example, if the timer facility is stopped, software ordinarily uses an external time reference to update the various timing counters upon restart.

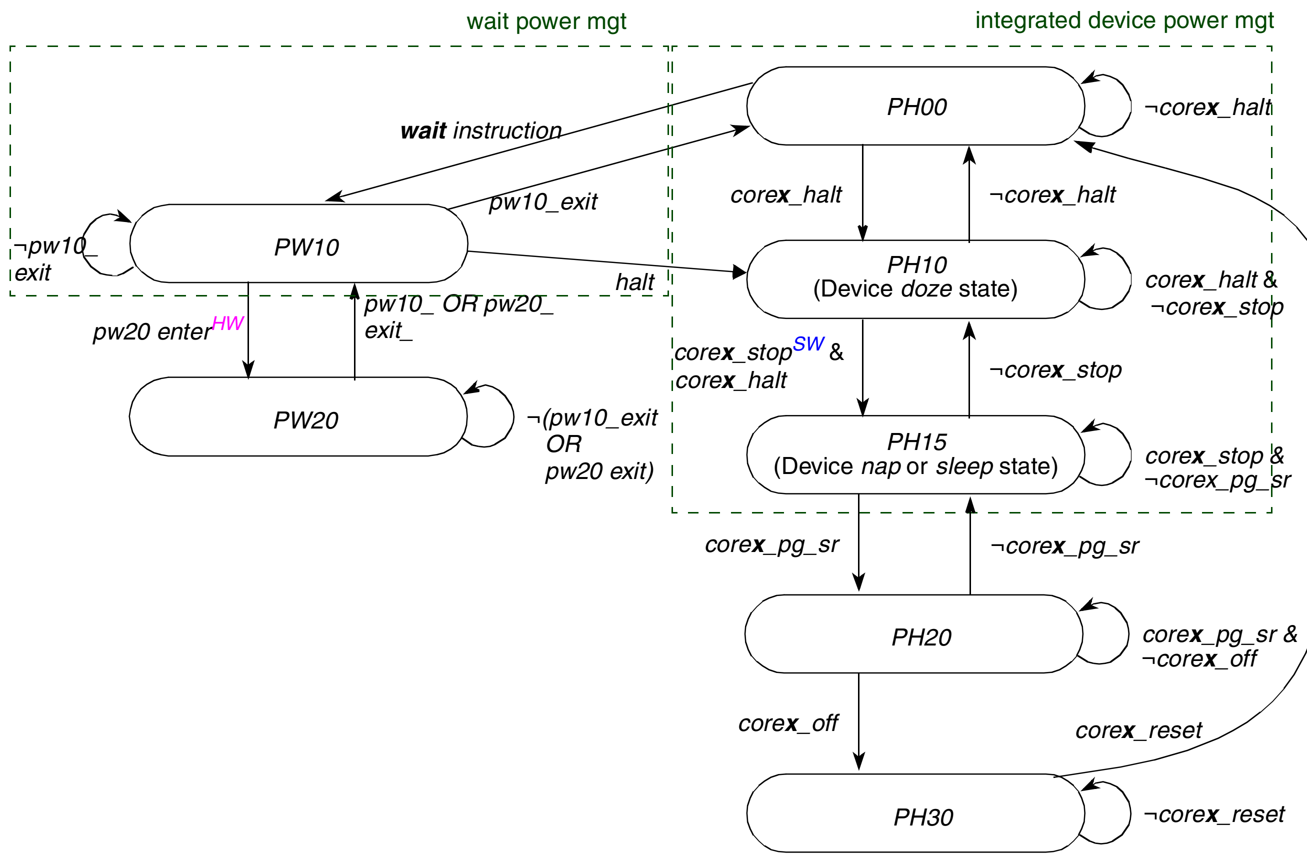
1 In *PH15* and *PH20*, the L1 instruction cache, the L1 data cache, and the TLBs are maintained but not updated with respect to coherency (that is, they do not respond to snoops, including **icbi** and **tlbivax**). Software is required to invalidate both caches and TLB entries without IPROT protection prior to entering *PH15* or *PH20* to avoid having stale state when *PH15* or *PH20* is exited.

2 The L1 data cache has all its data contents invalidated. The locks in the cache and the tags are retained.

3 The L1 instruction cache is invalidated only if PWRMGTCR0[PW20_INV_ICACHE] is set; otherwise, it is retained. Like the L1 data cache, the locks and the tags are retained.

4 A performance monitor interrupt cannot be used to transition the core to *PW10.* Performance monitor asynchronous interrupts cannot be asserted in *PW20* because *PW20* disables all performance monitor counting.

Figure 8-1. Core activity state diagram



SW Software must flush and invalidate (see previous PH20 state description).

HW Hardware invalidates the L1 data cache and L1 instruction cache if PMGTCR0[PW20_INV_ICACHE] = 1.

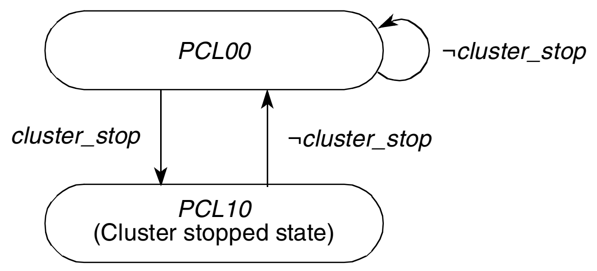
8.4 Cluster power management states

The e6500 cluster provides two different power management states: *PCL00* and *PCL10*. *PCL10* is initiated by altering power management registers in the integrated device. The cluster is eligible to move from *PCL00* to *PCL10* if all of its cores have reached the *PH30* state. Cluster power management states are described in the following table and figure.

Table 8-3. e6500 cluster activity states

State	Descriptions
PCL00 (default)	Default. Full-On. All cluster-level units operate at the full clock speed defined at power-up. The timebase continues to increment and timer functions are active. Individual cores within the cluster may independently be in any core activity state. Dynamic power management automatically stops clocking idle individual cluster internal functional units.
PCL10	Initiated by the integrated device by asserting the *cluster_stop* input. The integrated device must only assert *cluster_stop* after transitioning all cores in the cluster to one of the *PH20*, *PH30*, or *static_off* state. When all cores in the cluster have reached one of these states, the cluster responds by inhibiting clock distribution to the cluster functional units (after the CoreNet interface idles), and then asserting the *cluster_stopped* output. The L2 cache no longer continues to participate in snooping activities. Software should always flush and then invalidate the L2 cache prior to initiating the *PCL10* state to ensure that any modified data is written out to backing store. As shown in the following figure, the only exit condition to exit *PCL10* is negation of *cluster_stop* or, alternatively, assertion of *cluster_hreset_b*.

Figure 8-2. Cluster activity state diagram



8.5 Power management protocol

Each core within an e6500 cluster responds to signals driven by the integrated device platform that command the core to transition from the *PH00* state to the *PH10*, *PH15*, *PH20*, or *PH30* state by driving the *core**x**_halt_thrdy*, *core**x**_stop, core**x**_pg_sr*, and *core**x**_off* signals. When a core has entered the *PH10*, *PH15* or *PH20* state, it outputs the *core**x**_halted_thrdy*, *core**x**_stopped*, *or core**x**_ph20* signals to inform the platform that the commanded state transition is complete. Similarly, an e6500 cluster responds to the *cluster_stop* signal driven by the platform that commands the cluster to transition from *PCL00* to *PCL10* by outputting the *cluster_stopped* signal when the cluster has completed its entry of the *PCL10* state.

This figure shows the core power management handshaking.

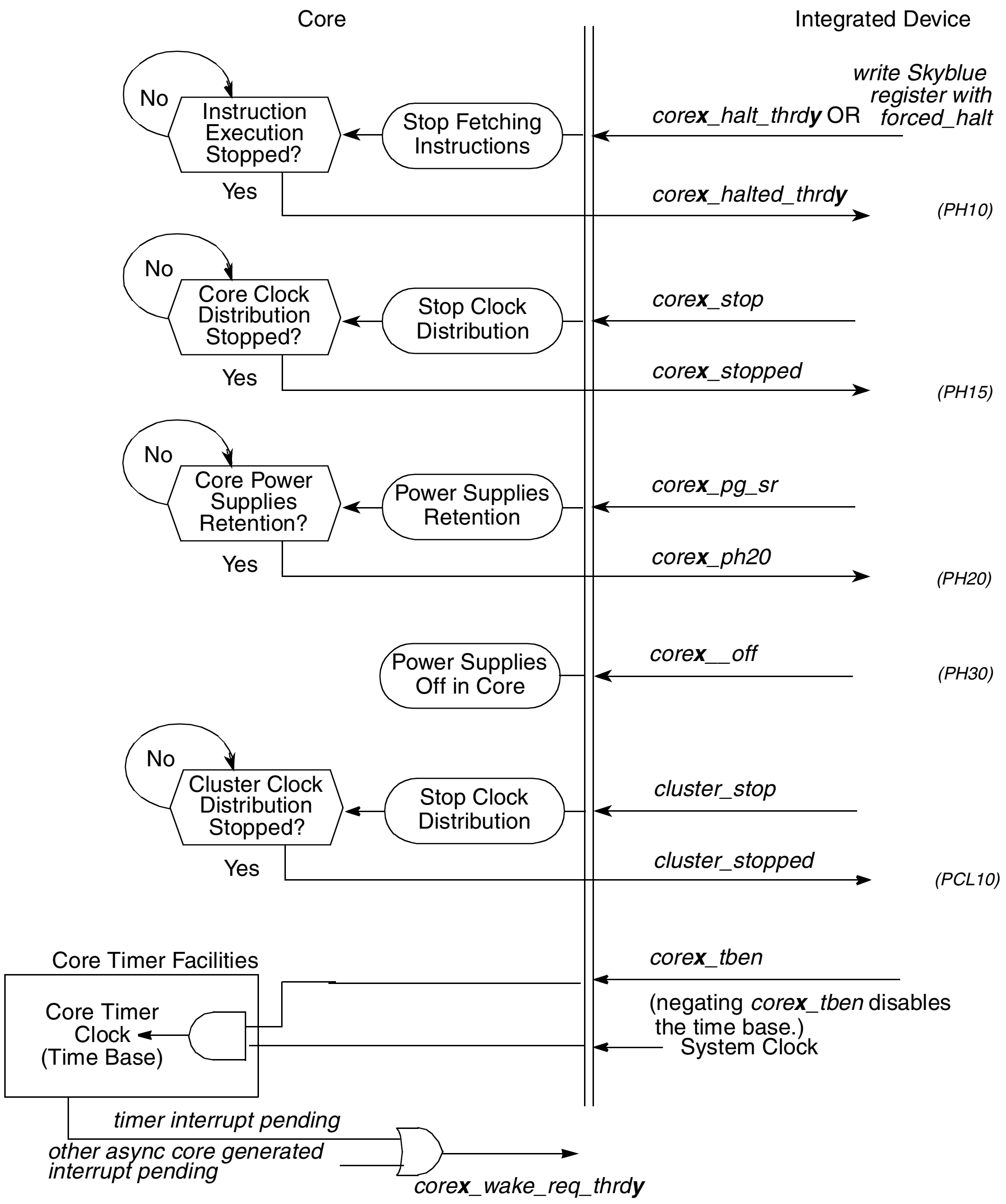


Figure 8-3. Core power management handshaking

8.6 AltiVec power down and power up

Each core in the e6500 cluster implements CDCSR0 as described in Section 2.7.6, "Core Device Control and Status (CDCSR0) register" and in *EREF*. Both the AltiVec execution unit and CDCSR0 are shared by all threads in a core.

Each core's AltiVec unit may be placed into a power-saving mode by turning off power to the unit. When the AltiVec unit is powered down, the AltiVec register file and VSCR continue to retain their architectural state to allow for faster power-up and software simplification. The power-down of the majority of the AltiVec unit allows for significant power consumption reduction either in integrated devices that do not make use of AltiVec or in currently executing applications that are known not to use any AltiVec instructions, including AltiVec loads and stores.

8.6.1 AltiVec power down—software controlled entry

1. It is recommended, but not required, for software (guest supervisor or hypervisor privilege) to clear (that is, write 0 to) MSR[SPV] of all threads in a core.
2. Software (hypervisor privilege) writes CDCSR0[CNTL] of the AltiVec device with Off (0b001).
3. Software (hypervisor privilege) then polls CDCSR0[STATE] of the AltiVec device until the state changes to 0b010 - "standby" (from 0b001 - "ready").
4. The AltiVec unit is now in a low power state.

8.6.2 AltiVec power down—hardware triggered entry

When PWRMGTCR0[AV_IDLE_PD_EN] = 1, core hardware automatically powers down the AltiVec unit after no AltiVec instructions have executed for the number of cycles specified by PWRMGTCR0[AV_IDLE_CNT_P]. When this condition is detected, the following power-down sequence occurs without software control:

1. CDCSR0[STATE] of the AltiVec device changes to 0b010 - "standby" (from 0b001 - "ready").
2. The AltiVec unit is now in a low power state.

8.6.3 AltiVec low power state retention

This table shows the state of the major portions of the AltiVec unit after either a AltiVec software controlled or hardware triggered power-down sequence has occurred.

Table 8-4. AltiVec unit low power state retention

State	AltiVec Low Power State Entry Method: Software or Hardware	Core Off
VPR, VSCR	State retained. Powered on with VDD_AV_SW.	Powered off. No state is retained.
AltiVec Unit	Powered off. Static power is reduced.	

8.6.4 AltiVec power up—hardware triggered

When the AltiVec unit is powered down (CDCSR0[STATE] of the AltiVec device is not 0b001), the AltiVec unit will power on and go to the "ready" state (0b001) when the core attempts execution of an AltiVec instruction and MSR[SPV] = 1. The instruction does not begin execution until the unit is powered up. The effect is similar to dynamic power gating during execution except that it affects static power of the AltiVec unit and the unit takes longer to power-up.

There is no mechanism to explicitly turn off the automatic power-up. However, software can prevent it from occurring by never setting MSR[SPV].

NOTE

Pre-production parts containing Rev 1 of the e6500 core do not automatically power up the core upon executing an AltiVec instruction and attempting to execute an AltiVec instruction when the unit is powered-down results in an AltiVec unavailable interrupt regardless of the state of MSR[SPV].

8.6.5 AltiVec power up sequence—software controlled

In general, software is not required to power up the AltiVec unit because it automatically powers up when an AltiVec instruction is executed with MSR[SPV] = 1. However, software may turn the unit on if desired by performing the following sequence:

1. Software (hypervisor privilege) writes CDCSR0[CNTL] of the AltiVec device with On (0b010).
2. AltiVec power gating switches are switched to "on" voltage.
3. Software (hypervisor privilege) then polls CDCSR0[STATE] of the AltiVec device until the state changes to 0b001—"ready".
4. Software (guest supervisor or hypervisor privilege) re-enables MSR[SPV] of all threads in the core.
5. AltiVec instructions can then be executed.

8.7 e6500 cluster power management sequence

This section details the sequence on how the e6500 cluster is powered off and on.

8.7.1 Cluster state PCL10 entry sequence

1. The cluster L2 cache is flushed and invalidated by software.
2. The integrated device asserts the *cluster_stop* signal.
3. Cluster clocks are turned off.
4. The cluster asserts the *cluster_stopped* signal.

8.7.2 Cluster PCL10 exit sequence

1. The integrated device negates the *cluster_stop* signal.
2. The cluster clocks are turned on.

3. The cluster negates the *cluster_stopped* signal.

8.8 Interrupts and power management

In the *PH10*, *PH15*, *PH20*, and *PH30* core activity states, the threads of the core do 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 e6500 *core**x**_wake_req_thrd**y*** outputs) to detect interrupt requests. Upon sensing an interrupt request, the integrated device ordinarily negates *core**x**_stop* and *core**x**_halt_thrd**y*** to restore the core to the *PH00* core activity state, allowing it to service the interrupt request.

The control of power management state changes, including current state and previous state status, is done completely through the integrated device. See the user manual of the integrated device for information on its power management programming model.

Chapter 9
Debug and Performance Monitor Facilities

9.1 Overview

The e6500 core provides hardware support for the following:

- Software debug agents that run natively on the e6500 processor
- External software debuggers that run on external hardware that is attached to the e6500 processor

The architecture defines a set of debug features that can be used by a software debug agent running natively on the processor or an external host debugger connected to the device. The features include:

- Instruction and data address comparators for breakpoints/watchpoints
- Real-time trace
- Performance monitor counters
- Debug events, which can cause debug interrupts to occur or the processor to halt

The e6500 core provides the flexibility to allocate the architecture-defined debug mechanisms to an external host debugger or internal software debug agent such that both debuggers can be used simultaneously. Debug resources allocated to the external debugger are protected from software, and vice versa.

The e6500 core implements debugger notify instructions (**dnh** and **dni**). **dnh** can be used to cause the processor to halt for an external debugger. **dni** can be used to cause the processor to enter debug interrupt processing for a software debug agent.

The Nexus trace facility provides real-time trace capabilities in compliance with IEEE-ISTO 5001. The development features supported are:

- Program Trace
- Data Trace
- Data Acquisition messaging
- Watchpoint messaging
- Performance Profile messaging
- Timestamp Correlation messaging
- Ownership Trace

The e6500 Nexus module also supports watchpoint triggering and processor overrun control.

The performance monitor facility allows collection of metrics on events that occur in the processor. These event metrics can be analyzed to detect less than optimal operation in order to improve system performance. The performance monitor can be configured by software to cause an interrupt on a counter event, such as counter overflow. Or, the performance monitor may be configured to use the Nexus trace facility to transmit counter values in a completely non-intrusive manner. The performance monitor facility is described in Section 9.12, "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 *EREF* and has the same definition. Some of this terminology, such as 'debug event,' appears in Power ISA and has a more limited scope, because 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 (DBSR) register or in the External Debug Status 0 (EDBSR0) register 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. Exceptions that have higher priority than a debug condition prevent the debug condition from being recorded as a debug event.

Internal debug mode (IDM) owned debug events cause a debug interrupt if the debug enable bit is set (MSR[DE] = 1). Internal debug mode is used by a software debug agent operating on the processor.

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), EDM owned debug events cause the processor to enter debug halt mode. External debug mode is used by a external host debugger, which is typically connected via JTAG or Aurora.

9.2 Debug resource sharing

The e6500 core provides registers to facilitate sharing of debug resources between an external host debugger and an internal debug agent. Both the external host debugger and internal debug agent should use these registers to ensure that sharing occurs smoothly.

9.2.1 Debug resource sharing between threads

Because the primary debug resources (IACs, DACs, Nexus, performance monitors, registers) are not shared between threads, the external host debugger and internal debug agent resource sharing mechanism, which is described in detail in subsequent sections, only applies within the thread.

For the e6500 core, each thread has its own private debug resources. This includes IACs, DACs, Nexus trace, performance monitors, and debug request/allocation/control/status registers. Each thread's resources should be treated independently, with the following exceptions:

- If one thread receives a debug halt request, both threads halt.
  - The PRSR of the primary halting thread should reflect the halting debug event.
  - The PRSR of the secondary halted thread should reflect it halted due to a cross-thread halt.
- The debug logic within both threads shares a debug reset.

All debug registers are replicated per thread and each reside in its own memory-mapped address space. See Section 9.10.3.1, "Memory-mapped access," for details.

9.2.2 Debug resource request—software debug agent

A software debug agent should always request ownership of a debug resource via the Debug Resource Request 0 (DBRR0) register. After requesting the resource, DBRR0 can be read back to determine if the resource was granted. If the resource was granted, the software can then use that resource freely. If the resource was not granted to the software debug agent, that means the debug resource is being used by an external host debugger and, thus, attempts to use it should not be made.

9.2.3 Debug resource request—external host debugger

Similar to a software debug agent, the external host debugger should always request ownership of a debug resource via the External Debug Resource Request 0 (EDBRR0) register. After requesting the resource, EDBRR0 can be read back to determine if the resource was granted. If the resource was granted, the debugger must then write 0 to the corresponding bit in the External Debug Resource Allocation Control 0 (EDBRAC0) register. Once the EDBRAC0 bit is cleared, the external host debugger can then use that resource freely. If the resource was not granted to the external host debugger, that means the debug resource is being used by an internal software debug agent and, thus, attempts to use it should not be made.

If, however, the requested (but not granted) resource is absolutely critical to continue an external host debug session, the external host debugger has the option to take away (steal) the resource from the software debug agent.

NOTE

Resource stealing should only be done as a last resort.

To take away the resource, the external debugger can write 0 to the corresponding bit in the External Debug Resource Allocation Control 0 (EDBRAC0) register. Once the EDBRAC0 bit is cleared, the external host debugger can then use that resource freely.

9.2.4 Debug resource allocation—external host debugger

Care must be taken while allocating resources for external debug usage so that resources being used by an internal software debug monitor are not unnecessarily taken away. In other words, before writing 1 to the EDBRC0[EDM] bit, the external debugger should evaluate the DBRR0 register, and then ensure the EDBRAC0 register reflects the DBRR0 settings. Only then should it write 1 to the EDBRC0[EDM] bit.

Once the EDBRAC0 register reflects any software debug monitor usage and the EDBRC0[EDM] bit has been enabled, the external host debugger should use the following guidelines for allocating resources:

1. Use EDBRR0 to request debug resources for its usage.
2. Read back EDBRR0 to see if the resources were granted.
3. Write to EDBRAC0 to allocate the granted debug resources.
4. Use the debug resources that are allocated to it.

If the external debugger requests a resource via EDBRR0, but the resource was not granted, that means the resource has already been granted to the internal software debugger via DBRR0. The external debugger has the option to override this configuration and take (that is, steal) the debug resource from the internal software debugger.

NOTE

Resource stealing should only be done as a last resort.

To steal the resource(s), the external debugger can write EDBRAC0 and allocate the resource(s) it wants to take to the external debug host.

9.2.5 Debug resource protection

Access to debug registers (other than DBSR) is conditioned so that resources are [somewhat] protected.

NOTE

The debug resource protections apply to software debug agent accesses via **mtspr/mfspr**, and external host debugger accesses via memory-mapped accesses to debug registers. These protections do not apply to software debug agent accesses to memory-mapped registers.

The debug registers are separated into three types:

1. Registers that are always owned by one side or the other. For example, external debug host registers, such as EDBCR0, EDBRAC0, EDBRR0, EDBSRMSK0, and EDBSR0, are only writable from the host side and not writable using the **mtspr** instruction. Similarly, a software debug agent has full access to registers, such as DBRR0 and DBSR, and the external host debugger cannot write these register directly.
2. Registers that can be allocated to one side or the other. Once the allocation has been made, only the side it's allocated to can access it. For example, the IAC registers, if allocated to the external debug host, are only writable via memory-mapped accesses and not via the **mtspr** instruction.
3. Registers that may have some bits allocated to one side and some bits allocated to the other. DBCR0 is a prime example of this. Each bit can be independently owned by one side or the other. Writes to DBCR0 from the software debug agent (via **mtspr**) only update bits of debug resources that it owns, and bits owned by the external host debugger are unaffected. Similarly, writes from the external host (via memory-mapped access) only update bits owned or allocated to the external host, and bits owned by the internal debug agent are unaffected.

Ownership, or allocation, of a debug resource is handled by EDBRAC0 and means full access privileges to registers, or bits within a register that are required to configure and use that resource.

9.2.5.1 Debug resource protection—from software debug agent’s perspective

For a software debug agent, accesses to debug registers (other than DBSR) are conditioned by the External Debug Mode control bit (EDBCR0[EDM]) and the settings of the Debug Resource Allocation Control (EDBRAC0) register, which are controlled by an external host debugger.

If EDBCR0[EDM] is cleared, the allocations in EDBRAC0 are ignored and software has access to all debug resources. Regardless, software should still use DBRR0 to request resources before using them!

If EDBCR0[EDM] is set, the EDBRAC0 configuration determines if the debug resource is “owned” (allocated to) the external host debugger or an internal software debug agent.

If the bit in EDBRAC0 corresponding to the resource is cleared, software is prevented from modifying that resource’s register values other than in DBSR, because the resource is not “owned” by software. Software always has ownership of DBSR. Execution of a **mtspr** instruction targeting a debug register or register field not “owned” by software does not cause modifications to occur, and no exception is signaled. In addition, because the external host debugger may be manipulating debug register values, the state of the registers or register fields not “owned” by software is not guaranteed to be consistent if accessed (read) by software with a **mtspr** instruction, other than the DBCR0[EDM] bit itself and EDBRAC0.

If the bit in EDBRAC0 corresponding to the resource is set, the software can modify that resource’s register values freely, because the resource is “owned” by the software debug agent.

NOTE

It should be noted that, because the external debugger has ultimate control of EDBRAC0, it has the ability to allocate resources however it sees fit. This includes the ability to take away or steal resources from the software debug agent. Of course, it should only do this as a last resort when no other options exist.

If a resource is taken away, there is no way for the software debug agent to be notified when this happens, and further accesses by the software debug agent to the stolen resource registers and control bits fail.

9.2.5.2 Debug resource protection—from external host debugger’s perspective

Similar to the software debug agent, accesses to debug registers for an external host debugger are conditioned by the External Debug Mode control bit (EDBCR0[EDM]) and the settings of the Debug Resource Allocation Control (EDBRAC0) register, which are controlled by an external host debugger.

As mentioned in Section 9.2.4, “Debug resource allocation—external host debugger,” care must be taken while allocating resources for external debug usage so that resources being used by an internal software debug monitor are not unnecessarily taken away. Thus, before setting the EDBRC0[EDM] bit, the external debugger should evaluate DBRR0 and ensure EDBRAC0 reflects the DBRR0 settings. Only then should it set the EDBRC0[EDM] bit.

If EDBCR0[EDM] is set, the EDBRAC0 register configuration determines if the debug resource is “owned” (allocated to) the external host debugger or an internal software debug agent.

If the bit in EDBRAC0 corresponding to the resource is set, the external debugger is prevented from modifying that resource's register values because the resource is not "owned" by the external host. A write access targeting a debug register or register field not "owned" by the external host does not cause modifications to occur. In addition, because the software debug agent may be manipulating debug register values, the state of the registers or register fields not "owned" by the host debugger is not guaranteed to be consistent if accessed (read) via the memory map interface, other than the DBCR0[EDM] bit itself and the external debugger registers (such as EDBCR0, EDBRAC0, EDBRR0, EDBSRMSK0).

If the bit in EDBRAC0 corresponding to the resource is cleared, the external debugger can modify that resource's register values freely, because the resource is "owned" by the external host.

9.3 Internal 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 2.14, "Internal debug registers."

9.4 External debug registers

The external debug registers are used for controlling the processor and reporting status while the e6500 external debug mode is enabled. When EDBCR0[EDM] is set, debug events that are allocated to the external host debugger (in EDBRAC0) and enabled (in DBCR0) cause the processor to enter debug halt mode if the events are not masked (in EDBSRMSK0), as opposed to generating debug interrupts.

9.4.1 External Debug Control 0 (EDBCR0) register

EDBCR0 is a control register that is accessible to an external host 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 to disable certain asynchronous interrupts.

EDBCR0 is not accessible by software running on the e6500 processor. However, the state of EDBCR0[EDM] is reflected as a read-only bit in DBCR0[EDM].

When EDBCR0 bits controlling asynchronous interrupt disables (EDMEO, EDCEO, EDEEO) are set, normal asynchronous interrupt enabling conditions are overridden. The setting of these bits does not modify the state of the *core***x***_wake_req* signal from the processor 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 waits 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 processor is committed to take that interrupt. The software takes the interrupt when the processor is taken out of the halted state, even if the processor writes the EDMEO, EDCEO, or EDEEO bits prior to resuming execution.

EDBCR0, shown in the following figure, contains bits for enabling external debug features.

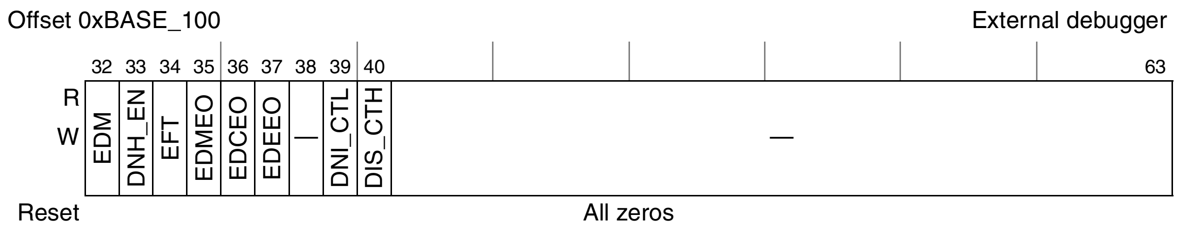


Figure 9-1. External Debug Control 0 (EDBCR0) register

This table describes EDBCR0 fields.

Table 9-1. EDBCR0 field descriptions

Bits	Name	Description
32	EDM	External Debug Mode 0 The processor is not in external debug mode. Debug events do not cause the processor to halt. 1 The processor 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. **Note:** The EFT bit applies to all timers, including the shared TB and ATB, and each thread's DEC, FIT, and watchdog timer, regardless of whether both threads halt or not.
35	EDMEO	Debugger Machine Check Interrupt Enable Override. When this bit is set, no asynchronous machine check interrupts occur. Exception conditions for asynchronous machine check interrupts that occur remain pending. This bit has no effect on error report interrupts, nor does it disable the NMI interrupt that 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 by the external debugger when the processor is in External Debug mode. Architecturally, the behavior of this bit is undefined when the processor is not in EDM mode. For the e6500 core, 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) occur. Exception conditions for critical interrupts that occur 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 by the external debugger when the processor is in External Debug mode. Architecturally, the behavior of this bit is undefined when the processor is not in EDM mode. For the e6500 core, 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) occur. Exception conditions for external interrupts that occur 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 by the external debugger when the processor is in External Debug mode. Architecturally, the behavior of this bit is undefined when the processor is not in EDM mode. For the e6500 core, this bit behaves the same, regardless of whether the processor is in EDM mode. **Note:** *EREF* 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	—	Reserved
39	DNI_CTL	**dni** Instruction Control 0 When the **dni** resource is owned by hardware, the MSR[DE] bit is cared. When MSR[DE] = 0, execution of **dni** instructions are no-oped and no entry into debug mode occurs. 1 When the **dni** resource is owned by hardware, the MSR[DE] bit is don't-cared. Execution of **dni** instructions causes entry into debug mode and a debug halt occurs, regardless of the value of MSR[DE]. **Note:** This control bit is only used when the dni resource is owned by hardware via control in EDBRAC0[DNI], and, thus, also only when EDM = 1
40	DIS_CTH	Disable Cross Thread Halt 0 This thread halts as well when a debug halt request is received by the other thread. 1 This thread does not halt when a debug halt request is received by the other thread.
41–63	—	Reserved

9.4.2 External Debug Resource Request 0 (EDBRR0) register

The External Debug Resource Request Register 0 (EDBRR0), shown in Figure 9-2, allows an external host debugger to request debug resources. After writing this register to request debug resources, reading this register back indicates which resources were granted. This register does not affect the actual allocation of debug resources. Allocation is handled by EDBRAC0.

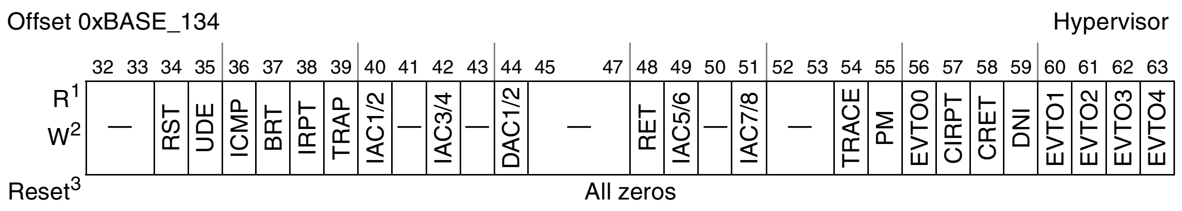


Figure 9-2. External Debug Resource Request 0 (EDBRR0) register

[1] Readable as an SPR and memory-mapped. Reads 0 for resources not granted to the external debug host (either not requested or previously owned and released). Reads 1 for resources that are granted to the external debug host.

[2] Writable via the memory-mapped interface only. Write 0 for resources not being requested or to release previously owned resources. Write 1 to request a resource be granted to the external debug host.

[3] Reset via POR (*core_trst_b*).

This table describes the EDBRR0 fields.

Table 9-2. EDBRR0 field descriptions

Bits	Name	Description
32–33	—	Reserved
34	RST	Reset Field Control (DBCR0[RST])
35	UDE	Unconditional Debug Event
36	ICMP	Instruction Complete Debug Event (DBCR0[ICMP])
37	BRT	Branch Taken Debug Event (DBCR0[BRT])
38	IRPT	Interrupt Taken Debug Condition Enable (DBCR0[IRPT])
39	TRAP	Trap Debug Event (DBCR0[TRAP])
40	IAC1/2	Instruction Address Compare 1 and 2
41	—	Reserved
42	IAC3/4	Instruction Address Compare 3 and 4
43	—	Reserved
44	DAC1/2	Data Address Compare 1 and 2
45–47	—	Reserved
48	RET	Return Debug Event (DBCR0[RET])
49	IAC5/6	Instruction Address Compare 5 and 6
50	—	Reserved
51	IAC7/8	Instruction Address Compare 7 and 8
52–53	—	Reserved
54	TRACE	Processor Nexus Trace
55	PM	Performance Monitor
56	EVTO0[1]	Event Out 0

Table 9-2. EDBRR0 field descriptions (continued)

Bits	Name	Description
57	CIRPT	Critical Interrupt Taken Debug Event (DBCR0[CIRPT])
58	CRET	Return From Critical Interrupt Debug Event (DBCR0[CRET])
59	DNI	Debug Notify Interrupt (**dni**) instruction
60	EVTO1[2]	Event Output 1
61	EVTO2[2]	Event Output 2
62	EVTO3[2]	Event Output 3
63	EVTO4[2]	Event Output 4

[1] Does not have a corresponding bit in the External Debug Resource Allocation Control (EDBRAC0) register and, thus, is only provided to identify if the resource is currently in use. No allocation of this resource is possible and, thus, internal and external debuggers should make every effort to not overwrite the configuration once the resource has been granted.

[2] Does not have a corresponding bit in the External Debug Resource Allocation Control (EDBRAC0) register and, thus, is only provided to identify if the resource is currently being used. No allocation of this resource is possible and, thus, internal and external debuggers should make every effort to not overwrite the configuration once the resource has been granted. The configuration of multiple event output bits is located in one register (DC3). If these events are shared, it is recommended that the external debugger configure these resources only when the processor is halted. Otherwise, read-modify-write problems arise when both the internal and external debugger write to this register, and the results are unpredictable.

9.4.3 External Debug Status 0 (EDBSR0) register

EDBSR0, shown in Figure 9-3, is a status register that is accessible to an external debugger through memory-mapped access. If PRSR[DE_HALT] indicates that the processor was halted by an enabled debug event when DBCR0[EDM] = 1 (see Section 9.4.7, “Processor Run Status (PRSR) register”) the corresponding status bit is set within EDBSR0.

Typically, only one EDBSR0 status bit is set based on the priorities defined in Table 9-27. However, if a non-UDE debug event is recognized as highest priority (and debug halt mode entry processing begins) and then a UDE debug event occurs before the processor fully halts, both the non-UDE debug event and the UDE debug event is set.

Once halted, EDBSR0 indicates the debug event that caused the halt and remains static while halted. All EDBSR0 status bits are cleared when the core exits debug halt mode to resume execution.

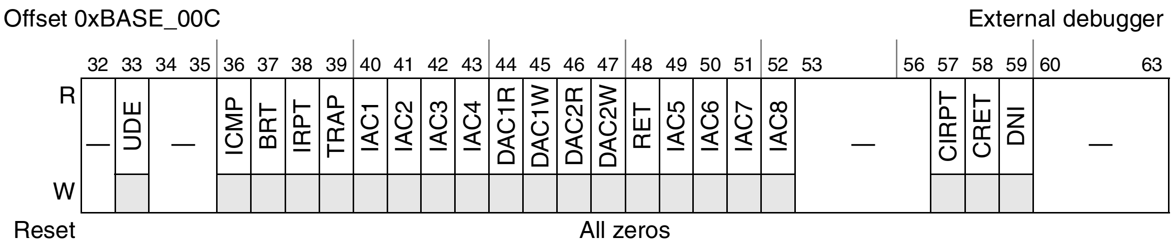


Figure 9-3. External Debug Status 0 (EDBSR0) register

This table describes the EDBSR0 fields.

Table 9-3. EDBSR0 field descriptions

Bits	Name	Description
32	—	Reserved
33	UDE	Unconditional Debug Event Set if an unconditional debug condition occurred, DBCR0[EDM] = 1, EDBSRMSK0[UDEM] = 0, and EDBRAC0[UDE] = 0.
34–35	—	Reserved
36	ICMP	Instruction Complete Debug Event Set if an instruction complete debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[ICMP] = 0, and DBCR0[ICMP] = 1.
37	BRT	Branch Taken Debug Event Set if a branch taken debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[BRT] = 0, and DBCR0[BRT] = 1.
38	IRPT	Interrupt Taken Debug Event Set if an interrupt taken debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[IRPT] = 0, and DBCR0[IRPT] = 1.
39	TRAP	Trap Instruction Debug Event Set if a Trap instruction debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[TRAP] = 0, and DBCR0[TRAP] = 1.
40	IAC1	Instruction Address Compare 1 Debug Event Set if a IAC1 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[IAC1] = 0, and DBCR0[IAC1] = 1.
41	IAC2	Instruction Address Compare 2 Debug Event Set if a IAC2 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[IAC2] = 0, and DBCR0[IAC2] = 1.
42	IAC3	Instruction Address Compare 3 Debug Event Set if a IAC3 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[IAC3] = 0, and DBCR0[IAC3] = 1.
43	IAC4	Instruction Address Compare 4 Debug Event Set if a IAC4 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[IAC4] = 0, and DBCR0[IAC4] = 1.
44	DAC1R	Data Address Compare 1 Read Debug Event Set if a read-type DAC1 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[DAC1] = 0, and DBCR0[DAC1] = 0b10 or 0b11.
45	DAC1W	Data Address Compare 1 Write Debug Event Set if a write-type DAC1 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[DAC1] = 0, and DBCR0[DAC1] = 0b01 or 0b11.
46	DAC2R	Data Address Compare 2 Read Debug Event Set if a read-type DAC2 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[DAC2] = 0, and DBCR0[DAC2] = 0b10 or 0b11.
47	DAC2W	Data Address Compare 2 Write Debug Event Set if a write-type DAC2 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[DAC2] = 0, and DBCR0[DAC2] = 0b01 or 0b11.

Table 9-3. EDBSR0 field descriptions (continued)

Bits	Name	Description
48	RET	Return Debug Event Set if a return debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[RET] = 0, and DBCR0[RET] = 1.
49	IAC5	Instruction Address Compare 5 Debug Event Set if a IAC5 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[IAC5] = 0, and DBCR0[IAC5] = 1.
50	IAC6	Instruction Address Compare 6 Debug Event Set if a IAC6 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[IAC6] = 0, and DBCR0[IAC6] = 1.
51	IAC7	Instruction Address Compare 7 Debug Event Set if a IAC7 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[IAC7] = 0, and DBCR0[IAC7] = 1.
52	IAC8	Instruction Address Compare 8 Debug Event Set if a IAC8 debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[IAC8] = 0, and DBCR0[IAC8] = 1.
53–56	—	Reserved
57	CIRPT	Critical Interrupt Taken Debug Event Set if a critical interrupt debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[CIRPT] = 0, and DBCR0[CIRPT] = 1.
58	CRET	Critical Return Debug Event Set if a critical return debug condition occurred, DBCR0[EDM] = 1, EDBRAC0[CRET] = 0, and DBCR0[CRET] = 1.
59	DNI	Debug Notify Interrupt (**dni**) instruction Set if a dni instruction condition occurred, DBCR0[EDM] = 1, EDBRAC0[DNI] = 0, EDBSRMSK0[UDEM] = 0, and DBCR0[DNI] = 1.
60–63	—	Reserved

9.4.4 External Debug Status Mask 0 (EDBSRMSK0) register

EDBSRMSK0, show in Figure 9-4, 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 an external host debugger 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.

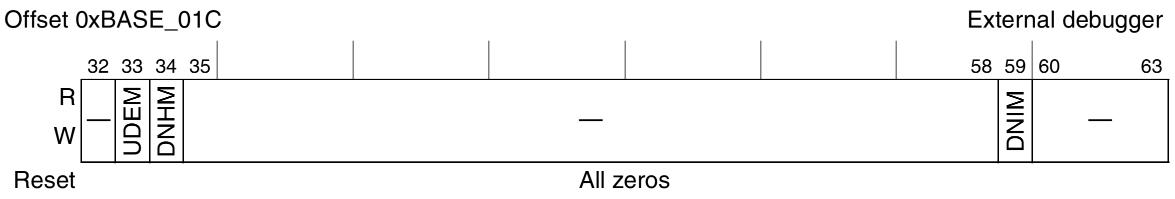


Figure 9-4. External Debug Status Mask 0 (EDBSRMSK0) register

This table describes the EDBSRMSK0 fields.

Table 9-4. EDBSRMSK0 field descriptions

Bits	Name	Description
32	—	Reserved
33	UDEM	Unconditional Debug Event Mask Set to 1 to mask entry into debug HALT state by EDBSR0[UDE].
34	DNHM	Debugger Notify Halt Event Mask Set to 1 to mask entry into debug HALT state by DNH.
35–58	—	Reserved
59	DNIM	Debug Notify Interrupt (**dni**) instruction Set to 1 to mask entry into debug HALT state by EDBSR0[DNI].
60–63	—	Reserved

9.4.5 External Debug Status 1 (EDBSR1) register

EDBSR1, shown in Figure 9-5, 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 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.

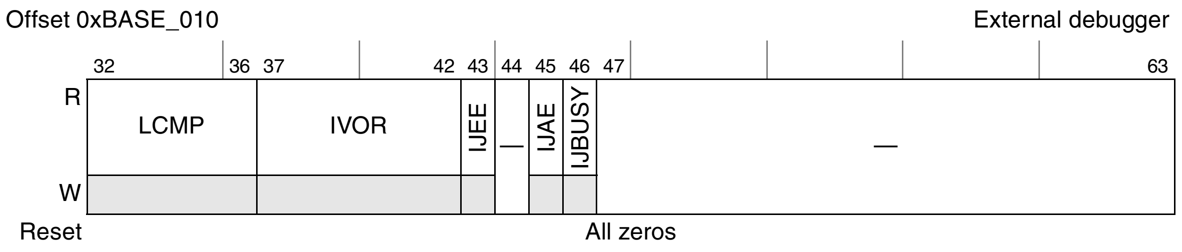


Figure 9-5. External Debug Status 1 (EDBSR1) register

This table describes the EDBSR1 fields.

Table 9-5. 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.
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 e6500 core 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.

9.4.6 External Debug Exception Syndrome (EDESR) register

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 *EREF*, with e6500-specific details listed in Section 2.9.7, "(Guest) Exception Syndrome (ESR/GESR) registers." EDESR is read only and is accessible through memory-mapped access.

9.4.7 Processor Run Status (PRSR) register

PRSR, shown in Figure 9-6, 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 processor 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, as soon as *core**x**_pm_halt* and *core**x**_stop* are deasserted. 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 processor remains halted or stopped.

Two events can cause the processor to resume execution from a halted state:

- *core**x**_pm_halt* or *core**x**_stop* is deasserted and no other halt or stop condition exists.
- *core**x**_resume* is asserted and no halt or stop condition exists.

The following sequence of events cause the processor to resume execution from the frozen (stopped) state:

1. While the processor is stopped, *core**x**_pm_halt* is asserted (by the system).
2. PRSR[STOPPED] is cleared.
3. *core**x**_resume* is asserted.

4.

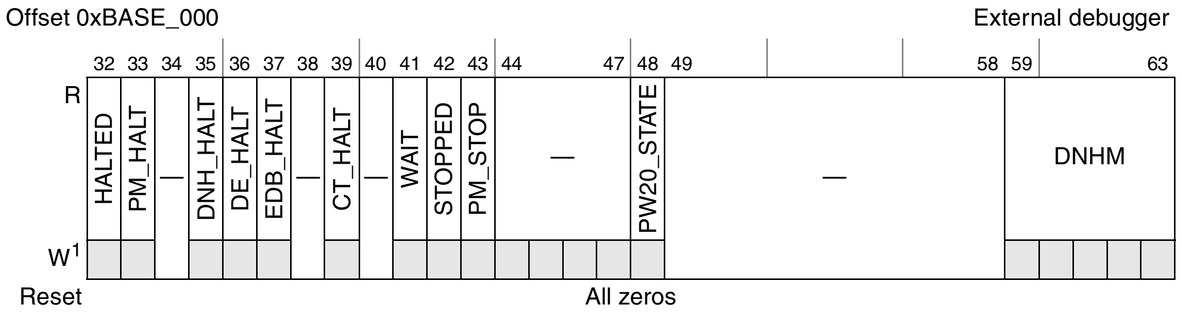


Figure 9-6. Processor Run Status (PRSR) register

[1] The writable bits of this register support a write-1-to-clear functionality. Writing zeros has no effect.

This table describes the PRSR fields.

Table 9-6. PRSR field descriptions

Bits	Name	Description
32	HALTED	Halted state. Set whenever the processor is halted. Cleared whenever the processor 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 receives a debug halt request from the system. This bit should be cleared by the debugger prior to issuing a resume command.
38	—	Reserved
39	CT_HALT	Cross-Thread Halt. Set whenever this thread is halted due to a cross-thread debug halt request from the other thread. This bit should be cleared by the debugger prior to issuing a resume command. **Note:** A cross-thread halt request includes DNH_HALT, DE_HALT, and EDB_HALT, but does not include PM_HALT or RUNN_HALT.
40	—	Reserved
41	WAIT	Processor is in a WAIT state caused by execution of a WAIT instruction.
42	STOPPED	Stopped state. Set whenever the processor is stopped. Cleared whenever the processor 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.
44–47	—	Reserved
48	PW20_STATE	Processor is in power management state PW20.
49–58	—	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.

9.4.8 Extended External Debug Control 0 (EEDCR0) register

The EEDCR0, shown in the following figure, provides extended controls not normally used in external debug operations.

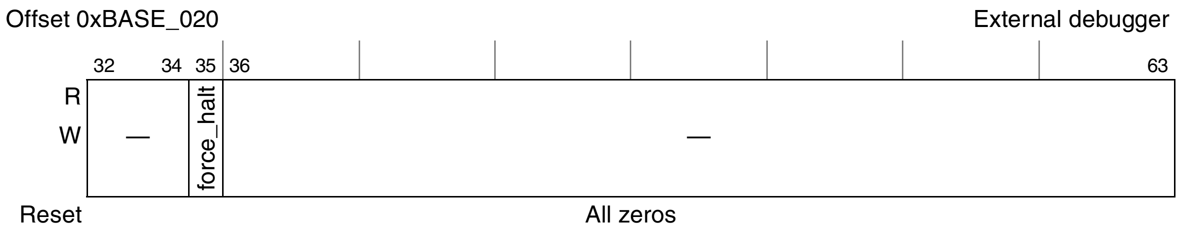


Figure 9-7. Extended External Debug Control 0 (EEDCR0) register

This table describes the EEDCR0 fields.

Table 9-7. EEDCR0 field descriptions

Bits	Name	Description
32–34	—	Reserved
35	force_halt	Force Halt. Writing a 1 to this field forces the processor to halt (for e6500, all threads 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. When read, this field always returns 0. Forcing the processor to halt using this control should only be done 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 fails to halt because part of the protocol for halting the processor 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 hangs. This control could then be used to force the processor into the halt state to examine the state of the processor. After a force_halt is commanded, the external debugger should not take any action or jam any instructions which causes the processor to attempt a transaction on the CoreNet interface (because doing so likely causes the processor to hang).
36–63	—	Reserved

9.4.9 Processor Debug Information (PDIR) register

The PIDR, shown in the following figure, provides configuration information about the debug facilities.

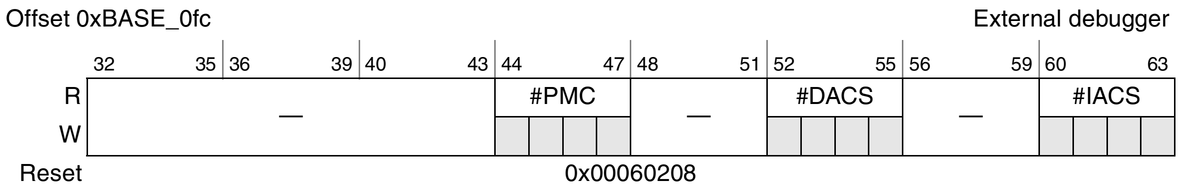


Figure 9-8. Processor Debug Information Register (PDIR)

This table describes the PDIR fields.

Table 9-8. PDIR Field Descriptions

Bits	Name	Description
32–43	—	Reserved
44–47	#PMCS	Reflects the number of performance monitor counters For e6500, #PMCS = 0110 to indicate there are 6 performance monitor counters present.
48–51	—	Reserved
52–55	#DACS	Reflects the number of Data Address Comparators For e6500, #DACS = 0010 to indicate there are 2 DACs present.
56–59	—	Reserved
60–63	#IACS	Reflects the number of Instruction Address Comparators For e6500, #IACS = 1000 to indicate there are 8 IACs present.

9.4.10 Next Instruction Address (NIA) register

NIA, shown in Figure 9-9, contains the next instruction address of the processor. The register is not accessible through software (**mtspr** or **mfspr**). The register is provided as a means for an external debugger to read and write the execution address of the processor. A **mtspr** or **mfspr** to this register can be jammed by the external debugger while the processor is halted.

An attempt to perform a **mtspr** or **mfspr** to this register while the processor is not halted results in an illegal instruction exception.

NIA is essentially an alias to the current thread's INIA register, except that NIA can read the value and NIA can only be accessed when the external debugger has halted the thread.

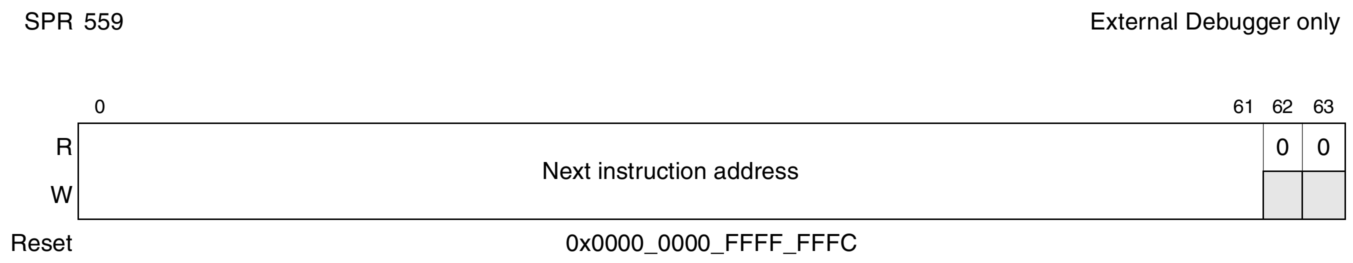


Figure 9-9. Next Instruction Address (NIA) register

9.5 Nexus registers

The Nexus control registers provide a mechanism to enable the various tracing features that are supported by the e6500 Nexus module. These registers are accessible through the NSPC and NSPD registers, as described in Section 2.14.12, "Nexus SPR access registers."

9.5.1 Nexus Development Control 1 (DC1) register

DC1, shown in the following figure, provides basic trace enable controls for the processor Nexus module.

Offset 0xBASE_408[1] External debugger

	32–35	36	37	38	39	40–48	49	50–51	52–53	54–55	56–57	58–63
R W	PPC	PMCR	—	LPTE	OTS	—	POTD	TSEN	EOC	EIC	—	TM
Reset	All zeros											

Figure 9-10. Nexus Development Control 1 (DC1) register

[1] Also accessible through NSPC/D. See Section 9.10.3.2, “Special-purpose register access (Nexus only).”

This table describes the DC1 fields.

Table 9-9. DC1 field descriptions

Bits	Name	Description
32–35	PPC	Performance Profile Configuration 0000 No profile messages are sent. xxx1 PMCC4 and PMCC5 xx1x PMCC2 and PMCC3 x1xx PMCC0 and PMCC1 1xxx PCC (NIA)
36	PMCR	Performance Monitor Counter Reset 0 Performance monitor counters are not reset on a snapshot. 1 Performance monitor counters are reset on a snapshot (results in smaller numbers for trace messages).
37	—	Reserved
38	LPTE	Lite Program Trace Enable[2] (Operates the same as the TM bits) 0 Full Program Trace is used. 1 Lite Program Trace is used.
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 00 Timestamps are disabled. (Timestamp Correlation message requests from the NPC are ignored.) 01 Timestamps are enabled and sent via Timestamp Correlation messages when requested by the NPC. Timestamps are not applied to other message types. 10 Timestamps are enabled and applied to all messages. Timestamp Correlation messages are also generated when requested by the NPC. 11 Coarse timestamps are enabled and a timestamp is periodically applied to messages. Timestamp Correlation messages are also generated when requested by the NPC.
52–53	EOC	Event Out Control 00 EVTO0 upon occurrence of any watchpoints selected by DC2[EWC0]. 01–11 Reserved; EVTO0 behaves as disabled.

Table 9-9. DC1 field descriptions (continued)

Bits	Name	Description
54–55	EIC	Event In Control[1] 00 EVTI0 for synchronization (Use EVTI0 to externally trigger a hard sync condition.) 01 Reserved 10 EVTI0 is 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 are disabled. xxxxx1 Ownership Trace is enabled. xxxx1x Data Trace is enabled. xxx1xx Program Trace is enabled. xx1xxx Watchpoint Trace is enabled. x1xxxx Profile Trace (in-circuit trace messages) is enabled. 1xxxxx Data Acquisition Trace is enabled.

1 EVTI may be used as a watchpoint condition independent of the settings of DC1[EIC]. See Table 9-57 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. See Section 9.5.5, “Nexus Watchpoint Trigger Control 1 (WT1) register,” for more information on watchpoint triggering.

9.5.2 Nexus Development Control 2 (DC2) register

DC2, shown in Figure 9-11, provides controls for the Event Out signal EVTO[0], which is a trigger output from the processor. The functions performed by EVTO*n* assertion are device-specific. See the appropriate section of the device’s reference manual for details.

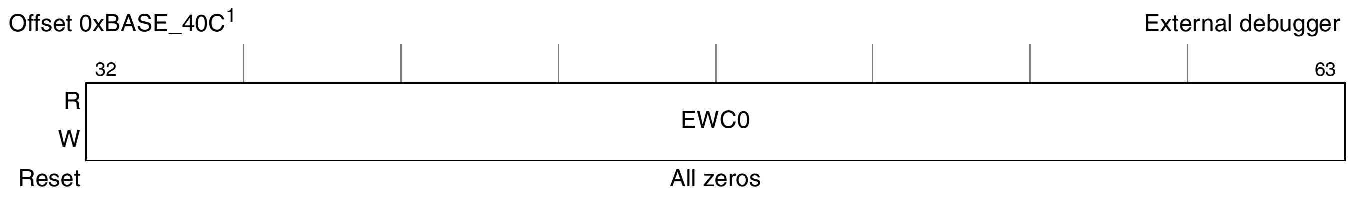


Figure 9-11. Nexus Development Control 2 (DC2) register

1 Also accessible through NSPC/D. See Section 9.10.3.2, “Special-purpose register access (Nexus only).”

This table describes the DC2 fields.

Table 9-10. DC2 field descriptions

Bits	Name	Description
32–63	EWC0	Event Out Watchpoint Control 0[1] 00000000000000000000000000000000 No watchpoints trigger EVTO0 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1 Watchpoint #1 triggers EVTO0 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1x Watchpoint #2 triggers EVTO0 xxxxxxxxxxxxxxxxxxxxxxxxxxxxx1xx Watchpoint #3 triggers EVTO0 xxxxxxxxxxxxxxxxxxxxxxxxxxxx1xxx Watchpoint #4 triggers EVTO0 xxxxxxxxxxxxxxxxxxxxxxxxxxx1xxxx Watchpoint #5 triggers EVTO0 xxxxxxxxxxxxxxxxxxxxxxxxxx1xxxxx Watchpoint #6 triggers EVTO0 xxxxxxxxxxxxxxxxxxxxxxxxx1xxxxxx Watchpoint #7 triggers EVTO0 xxxxxxxxxxxxxxxxxxxxxxxx1xxxxxxx Watchpoint #8 triggers EVTO0 xxxxxxxxxxxxxxxxxxxxxxx1xxxxxxxx Watchpoint #9 triggers EVTO0 xxxxxxxxxxxxxxxxxxxxxx1xxxxxxxxx Watchpoint #10 triggers EVTO0 xxxxxxxxxxxxxxxxxxxxx1xxxxxxxxxx Watchpoint #11 triggers EVTO0 xxxxxxxxxxxxxxxxxxxx1xxxxxxxxxxx Watchpoint #12 triggers EVTO0 xxxxxxxxxxxxxxxxxxx1xxxxxxxxxxxx Watchpoint #13 triggers EVTO0 xxxxxxxxxxxxxxxxxx1xxxxxxxxxxxxx Watchpoint #14 triggers EVTO0 xxxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx Watchpoint #15 triggers EVTO0 xxxxxxxxxxxxxxxx1xxxxxxxxxxxxxxx Watchpoint #16 triggers EVTO0 xxxxxxxxxxxxxxx1xxxxxxxxxxxxxxxx Watchpoint #17 triggers EVTO0 xxxxxxxxxxxxxx1xxxxxxxxxxxxxxxxx Watchpoint #18 triggers EVTO0 xxxxxxxxxxxxx1xxxxxxxxxxxxxxxxxx Watchpoint #19 triggers EVTO0 xxxxxxxxxxxx1xxxxxxxxxxxxxxxxxxx Watchpoint #20 triggers EVTO0 xxxxxxxxxxx1xxxxxxxxxxxxxxxxxxxx Watchpoint #21 triggers EVTO0 xxxxxxxxxx1xxxxxxxxxxxxxxxxxxxxx Watchpoint #22 triggers EVTO0 xxxxxxxxx1xxxxxxxxxxxxxxxxxxxxxx Watchpoint #23 triggers EVTO0 xxxxxxxx1xxxxxxxxxxxxxxxxxxxxxxx Watchpoint #24 triggers EVTO0 xxxxxxx1xxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #25 triggers EVTO0 xxxxxx1xxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #26 triggers EVTO0 xxxxx1xxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #27 triggers EVTO0 xxxx1xxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #28 triggers EVTO0 xxx1xxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #29 triggers EVTO0 xx1xxxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #30 triggers EVTO0 x1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #31 triggers EVTO0 1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #32 triggers EVTO0

9.5.3 Nexus Development Control 3 (DC3) register

DC3, shown in Figure 9-11, provides controls for the Event Out signals EVTO[1:4], which are trigger outputs from the processor. The functions performed by EVTO*n* assertion are device-specific. See the appropriate section of the device's reference manual for details.

Offset 0xBASE_410[1] External debugger

	32 33	34 39	40 41	42 47	48 49	50 55	56 57	58 63
R W	—	EWC4	—	EWC3	—	EWC2	—	EWC1
Reset	All zeros							

Figure 9-12. Nexus Development Control 3 (DC3) register

[1] Also accessible through NSPC/D. See Section 9.10.3.2, "Special-purpose register access (Nexus only)."

This table describes the DC3 fields.

Table 9-11. DC3 field descriptions

Bits	Name	Description
32–33	—	Reserved
34–39	EWC4	Event Out Watchpoint Control 4 000000 No watchpoints trigger EVTO4. 000001–100000 Watchpoint #1–#32 (respectively) triggers EVTO4.[1] 100001–111111 Reserved
40–41	—	Reserved
42–47	EWC3	Event Out Watchpoint Control 3 000000 No watchpoints trigger EVTO3. 000001–100000 Watchpoint #1–#32 (respectively) triggers EVTO3.[1] 100001–111111 Reserved
48–49	—	Reserved
50–55	EWC2	Event Out Watchpoint Control 2 000000 No watchpoints trigger EVTO2. 000001–100000 Watchpoint #1–#32 (respectively) triggers EVTO2.[1] 100001–111111 Reserved
56–57	—	Reserved
58–63	EWC1	Event Out Watchpoint Control 1 000000 No watchpoints trigger EVTO1. 000001–100000 Watchpoint #1–#32 (respectively) triggers EVTO1.[1] 100001–111111 Reserved

[1] See Table 9-57 for information on the events that are mapped to these watchpoints.

9.5.4 Nexus Development Control 4 (DC4) register

DC4, shown in Figure 9-13, provides additional control of Nexus debug features, Specifically, this register controls the masking of events that initiate Program Correlation messages (PCM), as well as trace filters based on MSR state.

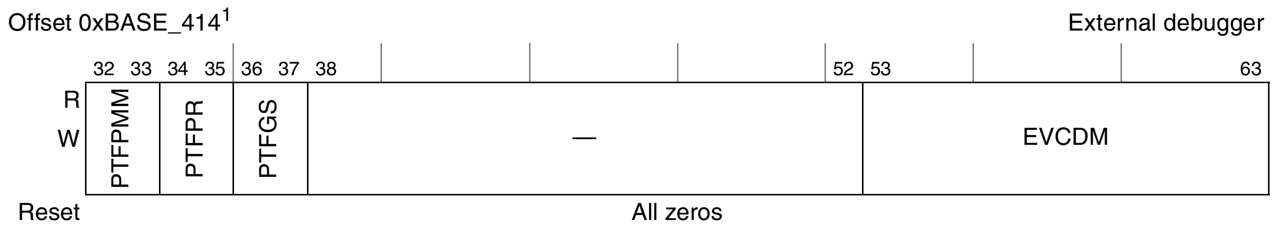


Figure 9-13. Nexus Development Control 4 (DC4) register

[1] Also accessible through NSPC/D. See Section 9.10.3.2, "Special-purpose register access (Nexus only)."

This table describes the DC4 fields.

Table 9-12. DC4 field descriptions

Bits	Name	Description
32–33	PTFPMM[1]	Program Trace Filtering on Performance Monitor Mark 00 Program Trace is unaffected by MSR[PMM]. 01 Reserved (Program Trace is unaffected by MSR[PMM].) 10 Program Trace messages are generated only when MSR[PMM] = 0. 11 Program Trace messages are generated only when MSR[PMM] = 1. System software can set MSR[PMM] to mark which execution contexts to enable performance monitor statistics to be gathered or to filter Nexus Program Trace messages from being generated. See Section 2.7.1, “Machine State (MSR) register.”
34–35	PTFPR[1]	Program Trace Filtering on Privilege 00 Program Trace is unaffected by MSR[PR]. 01 Reserved (Program Trace is unaffected by MSR[PR].) 10 Program Trace messages are generated only when MSR[PR] = 0 (supervisor mode). 11 Program Trace messages are generated only when MSR[PR] = 1 (user mode). These bits are used to provide execution context filtering. Filtering applies to Program Trace only.
36–37	PTFGS[1]	Program Trace Filtering on Guest State 00 Program Trace is unaffected by MSR[GS]. 01 Reserved (Program Trace is unaffected by MSR[GS].) 10 Program Trace messages are generated only when MSR[GS] = 0 (not in guest state). 11 Program Trace messages are generated only when MSR[GS] = 1 (guest state). These bits are used to provide execution context filtering. Filtering applies to Program Trace only.
38–52	—	Reserved
53–63	EVCDM	Event Code (EVCODE) Mask[2] 00000000000 No EVCODEs are masked for Program Correlation messages. xxxxxxxxxx1 EVCODE #1 is masked for Program Correlation messages. xxxxxxxxx1x EVCODE #2 is masked for Program Correlation messages. xxxxxxxx1xx–xxxxxxx1xxx Reserved xxxxxx1xxxx EVCODE #5 is masked for Program Correlation messages. xxxxx1xxxxx–xx1xxxxxxxx Reserved x1xxxxxxxxx EVCODE #10 is masked for Program Correlation messages. 1xxxxxxxxxx EVCODE #11 is masked for Program Correlation messages.

[1] All three conditions must be met for Program Trace to be enabled.

[2] See Table 9-43 for implemented EVCODEs.

9.5.5 Nexus Watchpoint Trigger Control 1 (WT1) register

WT1, shown in Figure 9-14, provides controls for watchpoint triggers that can be used to start and stop program and data 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 toggles 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.

Using watchpoints that result in toggling on then off Program Trace too close together (within a few cycles) may result in loss of that small window of trace because it takes a few cycles for Program Trace to generate its first synchronization message.

Offset 0xBASE_42C[1] External debugger

	32 33	34 39	40 41	42 47	48 49	50 55	56 57	58 63
R W	—	PTS	—	PTE	—	DTS	—	DTE
Reset	All zeros							

Figure 9-14. Nexus Watchpoint Trigger 1 register

[1] Also accessible through NSPC/D. See Section 9.10.3.2, "Special-purpose register access (Nexus only)."

This table describes the WT1 fields.

Table 9-13. WT1 field descriptions

Bits	Name	Description
32–33	—	Reserved
34–39	PTS	Program Trace Start 000000 Trigger is disabled. 000001–100000 Start Program Trace on Watchpoints 1–32 (see Table 9-57). 100001–111111 Reserved
40–41	—	Reserved
42–47	PTE	Program Trace End 000000 Trigger is disabled. 000001–100000 = End Program Trace on Watchpoints 1–32 (see Table 9-57). 100001–111111 = Reserved
48–49	—	Reserved
50–55	DTS	Data Trace Start 000000 Trigger is disabled. 000001–100000 Start Data Trace on Watchpoints 1–32 (see Table 9-57). 100001–111111 Reserved
56–57	—	Reserved
58–63	DTE	Data Trace End 000000 Trigger is disabled. 000001–100000 End Data Trace on Watchpoints 1–32 (see Table 9-57). 100001–111111 Reserved

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).

9.5.6 Nexus Watchpoint Trigger Control 2 (WT2) register

WT2, shown in Figure 9-15, provides controls for watchpoint triggers that can be used to start and stop in-circuit trace (performance profiling messages) and light program 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 toggles 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.

Offset 0xBASE_430[1] External debugger

	32 33	34 39	40 41	42 47	48 49	50 55	56 57	58 63
R W	—	ITS	—	ITE	—	LPTS	—	LPTE
Reset	All zeros							

Figure 9-15. Nexus Watchpoint Trigger 2 register

[1] Also accessible through NSPC/D. See Section 9.10.3.2, “Special-purpose register access (Nexus only).”

This table describes the WT2 fields.

Table 9-14. WT2 field descriptions

Bits	Name	Description
32–33	—	Reserved
34–39	ITS	In-Circuit Trace (Performance Profiling messages) Start 000000 Trigger is disabled. 000001–100000 Start ICT trace on Watchpoints 1–32 (see Table 9-57). 100001–111111 Reserved
40–41	—	Reserved
42–47	ITE	In-Circuit Trace (Performance Profiling messages) End 000000 Trigger is disabled 000001–100000 End ICT trace on Watchpoints 1–32 (see Table 9-57). 100001–111111 Reserved

Table 9-14. WT2 field descriptions (continued)

Bits	Name	Description
48–49	—	Reserved
50–55	LPTS	Lite Program Trace Start 000000 Trigger is disabled. 000001–100000 Start Lite Program Trace on Watchpoints 1–32 (see Table 9-57). 100001–111111 Reserved
56–57	—	Reserved
58–63	LPTE	Lite Program Trace End 000000 Trigger is disabled. 000001–100000 End Lite Program Trace on Watchpoints 1–32 (see Table 9-57). 100001–111111 Reserved

9.5.7 Nexus Watchpoint Mask (WMSK) register

WMSK, shown in Figure 9-16, controls which watchpoint events are enabled to produce Watchpoint Trace messages. Note that DC1[TM] must also be programmed to generate Watchpoint Trace messages.

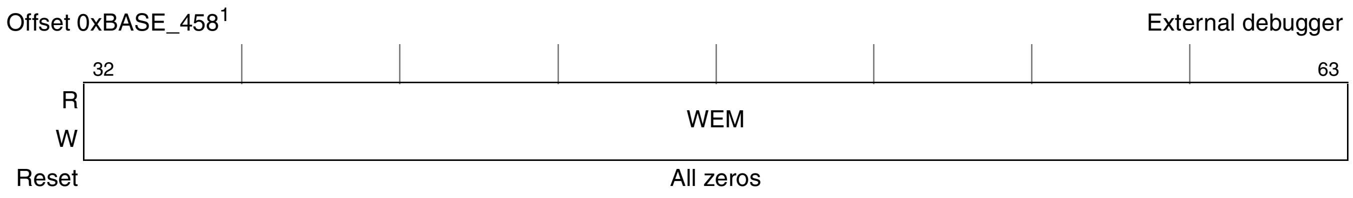


Figure 9-16. Nexus Watchpoint Mask (WMSK) register

1 Also accessible through NSPC/D. See Section 9.10.3.2, "Special-purpose register access (Nexus only)."

This table describes the WMSK fields.

Table 9-15. WMSK Field Descriptions

Bits	Name	Description
32–63	WEM	Watchpoint Enable for messaging[1] 00000000000000000000000000000000 No Watchpoints enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1 Watchpoint #1 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1x Watchpoint #2 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxxxxxxxxxxx1xx Watchpoint #3 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxxxxxxxxxx1xxx Watchpoint #4 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxxxxxxxxx1xxxx Watchpoint #5 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxxxxxxxx1xxxxx Watchpoint #6 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxxxxxxx1xxxxxx Watchpoint #7 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxxxxxx1xxxxxxx Watchpoint #8 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxxxxx1xxxxxxxx Watchpoint #9 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxxxx1xxxxxxxxx Watchpoint #10 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxxx1xxxxxxxxxx Watchpoint #11 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxxx1xxxxxxxxxxx Watchpoint #12 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxxx1xxxxxxxxxxxx Watchpoint #13 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxxx1xxxxxxxxxxxxx Watchpoint #14 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx Watchpoint #15 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxxx1xxxxxxxxxxxxxxx Watchpoint #16 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxxx1xxxxxxxxxxxxxxxx Watchpoint #17 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxxx1xxxxxxxxxxxxxxxxx Watchpoint #18 is enabled for Watchpoint Trace messaging xxxxxxxxxxxxx1xxxxxxxxxxxxxxxxxx Watchpoint #19 is enabled for Watchpoint Trace messaging xxxxxxxxxxxx1xxxxxxxxxxxxxxxxxxx Watchpoint #20 is enabled for Watchpoint Trace messaging xxxxxxxxxxx1xxxxxxxxxxxxxxxxxxxx Watchpoint #21 is enabled for Watchpoint Trace messaging xxxxxxxxxx1xxxxxxxxxxxxxxxxxxxxx Watchpoint #22 is enabled for Watchpoint Trace messaging xxxxxxxxx1xxxxxxxxxxxxxxxxxxxxxx Watchpoint #23 is enabled for Watchpoint Trace messaging xxxxxxxx1xxxxxxxxxxxxxxxxxxxxxxx Watchpoint #24 is enabled for Watchpoint Trace messaging xxxxxxx1xxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #25 is enabled for Watchpoint Trace messaging xxxxxx1xxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #26 is enabled for Watchpoint Trace messaging xxxxx1xxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #27 is enabled for Watchpoint Trace messaging xxxx1xxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #28 is enabled for Watchpoint Trace messaging xxx1xxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #29 is enabled for Watchpoint Trace messaging xx1xxxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #30 is enabled for Watchpoint Trace messaging x1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #31 is enabled for Watchpoint Trace messaging 1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #32 is enabled for Watchpoint Trace messaging

[1] See Table 9-57 for information on the events that are mapped to these watchpoints.

9.5.8 Nexus Overrun Control (OVCR) register

OVCR, shown in Figure 9-17, controls Nexus behavior as the internal message queues fill up. Response behavior options include suppressing selected message types and stalling the processor's instruction completion. See Section 9.11.6, "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 62	63
R W	—	SPTHOLD	—	SPEN	—	STTHOLD	—	STEN
Reset	All zeros							

Figure 9-17. Nexus Overrun Control (OVCR) register

[1] Also accessible through NSPC/D. See Section 9.10.3.2, "Special-purpose register access (Nexus only)."

This table describes the OVCR fields.

Table 9-16. 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.

9.5.9 Reloadable Counter Configuration (RCCR) register

RCCR, shown in Figure 9-14, provides controls for the reloadable counter. Once enabled, the reloadable counter starts from zero and begins counting the event selected within the RCEVENT field. When the reloadable counter reaches the value indicated in RCVR, a reloadable counter event asserts. Depending on the RCMODE field, the counter may then disable itself or reset back to zero and continue counting.

NOTE

The reloadable counter stops counting (freezes) when the processor enters debug halt mode or when the core enters a low power mode where the core clock is disabled. The reloadable counter resumes counting when debug halt mode is exited or when the clocks are turned back on as the low power mode is exited.

NOTE

Selecting reserved values for RCEVENT and/or RCMODE could result in unwanted watchpoints or unexpected behavior.

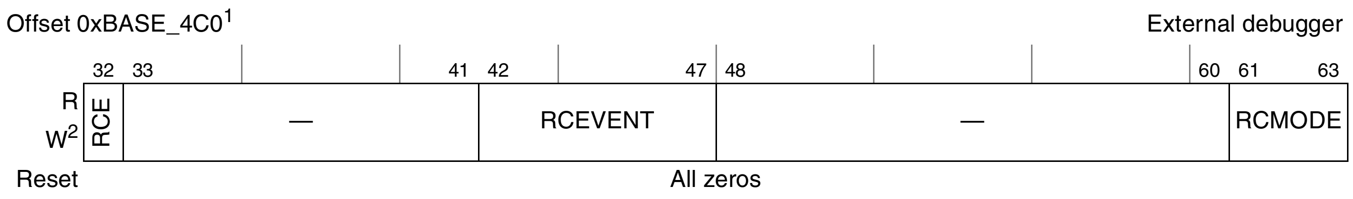


Figure 9-18. Reloadable Counter Configuration (RCCR) register

[1] Also accessible through NSPC/D. See Section 9.10.3.2, "Special-purpose register access (Nexus only)."

[2] Writes to this register reset the actual counter to 0.

This table describes the RCCR fields.

Table 9-17. RCCR field descriptions

Bits	Name	Description
32	RCE	Reloadable Counter Enable
33–41	—	Reserved
42–47	RCEVENT	Reloadable Counter Event Input Select 000000 Count clock cycles 000001–100000 Count Watchpoint 1–32 (see Table 9-57) 100001–111111 Reserved
48–60	—	Reserved
61–63	RCMODE	Reloadable Counter Mode 000 One Shot Counting Reset counter, count up until reaching the Reloadable Count Value (RCV), generate an event, and clear the RCE bit to disable the counter. 001 Continuous mode Reset counter, count up until reaching the Reloadable Count Value (RCV), generate an event, and repeat that process. 011–111 Reserved

9.5.10 Reloadable Counter Value (RCVR) register

RCVR, shown in Figure 9-14, provides the value to be compared to the reloadable counter. Upon match of the count value and RCV, an event is generated, and the counter is either disabled (1-shot mode) or is reset and event counting continues (continuous mode) depending on the RCCR[RCMODE] setting.

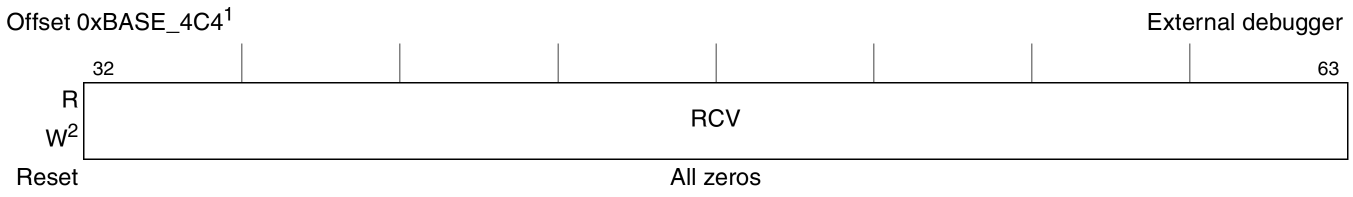


Figure 9-19. Reloadable Counter Value (RCVR) register

[1] Also accessible through NSPC/D. See Section 9.10.3.2, “Special-purpose register access (Nexus only).”

[2] Writes to this register to set a new RCV, reset the actual counter to 0.

This table describes the RCVR fields.

Table 9-18. RCVR field descriptions

Bits	Name	Description
32–63	RCV	Reloadable Count Value

9.5.11 Performance Monitor Snapshot Configuration (PMSCR) register

PMSCR, shown in Figure 9-20, controls which watchpoint events are enabled to generate a snapshot trigger to capture the performance monitor counters and PC in their capture registers.

NOTE

When Nexus performance profile messages are being used to egress the PC and performance monitor count values at each snapshot, care should be taken to ensure the snapshot event periodicity is not less than 16 clock cycles. Otherwise, some performance profile messages may be lost.

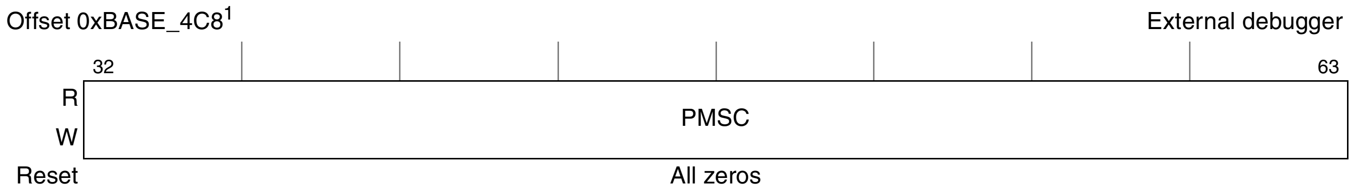


Figure 9-20. Performance Monitor Snapshot Configuration (PMSCR) register

[1] Also accessible through NSPC/D. See Section 9.10.3.2, “Special-purpose register access (Nexus only).”

This table describes the PMSCR fields.

Table 9-19. PMSCR Field Descriptions

Bits	Name	Description
32–63	PMSC	Watchpoint Enable for performance monitor snapshots[1]
		00000000000000000000000000000000 No watchpoints enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1 Watchpoint #1 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1x Watchpoint #2 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxxxxxxxxxxx1xx Watchpoint #3 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxxxxxxxxxx1xxx Watchpoint #4 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxxxxxxxxx1xxxx Watchpoint #5 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxxxxxxxx1xxxxx Watchpoint #6 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxxxxxxx1xxxxxx Watchpoint #7 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxxxxxx1xxxxxxx Watchpoint #8 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxxxxx1xxxxxxxx Watchpoint #9 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxxxx1xxxxxxxxx Watchpoint #10 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxxx1xxxxxxxxxx Watchpoint #11 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxxx1xxxxxxxxxxx Watchpoint #12 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxxx1xxxxxxxxxxxx Watchpoint #13 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxxx1xxxxxxxxxxxxx Watchpoint #14 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx Watchpoint #15 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxxx1xxxxxxxxxxxxxxx Watchpoint #16 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxxx1xxxxxxxxxxxxxxxx Watchpoint #17 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxxx1xxxxxxxxxxxxxxxxx Watchpoint #18 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxxx1xxxxxxxxxxxxxxxxxx Watchpoint #19 is enabled to generate performance monitor snapshots
		xxxxxxxxxxxx1xxxxxxxxxxxxxxxxxxx Watchpoint #20 is enabled to generate performance monitor snapshots
		xxxxxxxxxxx1xxxxxxxxxxxxxxxxxxxx Watchpoint #21 is enabled to generate performance monitor snapshots
		xxxxxxxxxx1xxxxxxxxxxxxxxxxxxxxx Watchpoint #22 is enabled to generate performance monitor snapshots
		xxxxxxxxx1xxxxxxxxxxxxxxxxxxxxxx Watchpoint #23 is enabled to generate performance monitor snapshots
		xxxxxxxx1xxxxxxxxxxxxxxxxxxxxxxx Watchpoint #24 is enabled to generate performance monitor snapshots
		xxxxxxx1xxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #25 is enabled to generate performance monitor snapshots
		xxxxxx1xxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #26 is enabled to generate performance monitor snapshots
		xxxxx1xxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #27 is enabled to generate performance monitor snapshots
		xxxx1xxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #28 is enabled to generate performance monitor snapshots
		xxx1xxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #29 is enabled to generate performance monitor snapshots
		xx1xxxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #30 is enabled to generate performance monitor snapshots
		x1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #31 is enabled to generate performance monitor snapshots
		1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx Watchpoint #32 is enabled to generate performance monitor snapshots

[1] See Table 9-57 for information on the events that are mapped to these watchpoints.

9.6 Instruction Jamming (IJAM) registers

This section discusses the following IJAM registers:

- Section 9.6.1, "IJAM Configuration (IJCFG) register"
- Section 9.6.2, "IJAM Instruction (IJIR) register"
- Section 9.6.3, "IJAM data registers 0–3 (IJDATA0, IJDATA1, IJDATA2, IJDATA3)"

9.6.1 IJAM Configuration (IJCFG) register

IJCFG, shown in Figure 9-21, controls the basic settings for jamming instructions into the e6500 core. It includes page attributes, addressing modes, and target storage space (memory or debug) for load/store instructions and other controls.

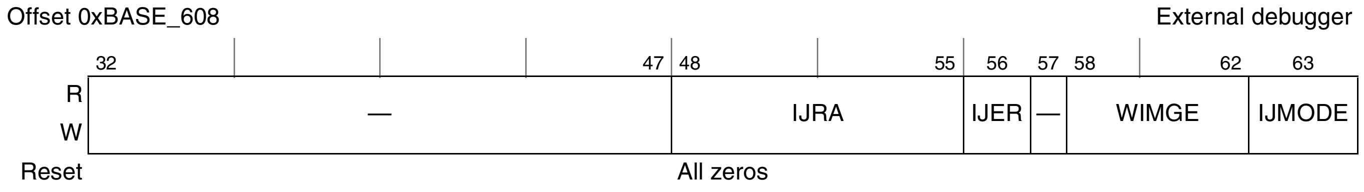


Figure 9-21. IJAM Configuration (IJCFG) register

This table describes the IJCFG fields.

Table 9-20. IJCFG field descriptions

Bits	Name	Description
32–47	—	Reserved
48–55	IJRA	Real Address (bits 24-31). If the jammed instruction is a load or store instruction and IJER = 1, these 8 bits are prepended to the 32-bit effective address to form a 40-bit physical address. (PA[24:63] is equal to the concatenation of IJRA[0:7] and 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 virtual 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 *EREF*'s Cache and MMU Background sections.
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.

9.6.2 IJAM Instruction (IJIR) register

IJIR, shown in the following figure, contains the instruction that is to be jammed into the e6500 processor.

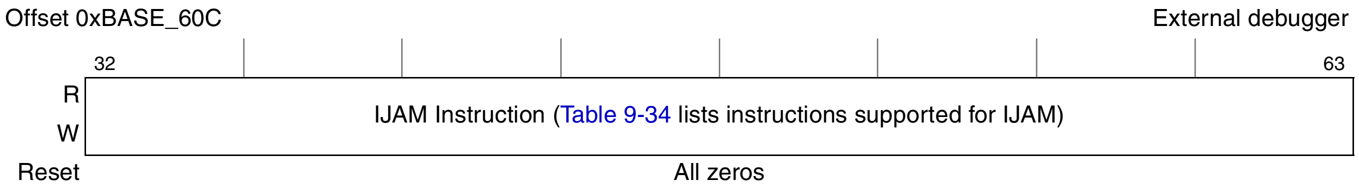

Offset 0xBASE_60C
External debugger
32
63
R
W
IJAM Instruction (Table 9-34 lists instructions supported for IJAM)
Reset
All zeros

Figure 9-22. IJAM Instruction (IJIR) register

9.6.3 IJAM data registers 0–3 (IJDATA0, IJDATA1, IJDATA2, IJDATA3)

IJDATA0–IJDATA3, shown in Figure 9-23, contain data associated with load/store instructions. Data is written to these registers when the jammed instruction requires associated write data (for example, load instructions from debug space). Data is read from these registers when the jammed instruction has associated result data (for example, stores to debug space).

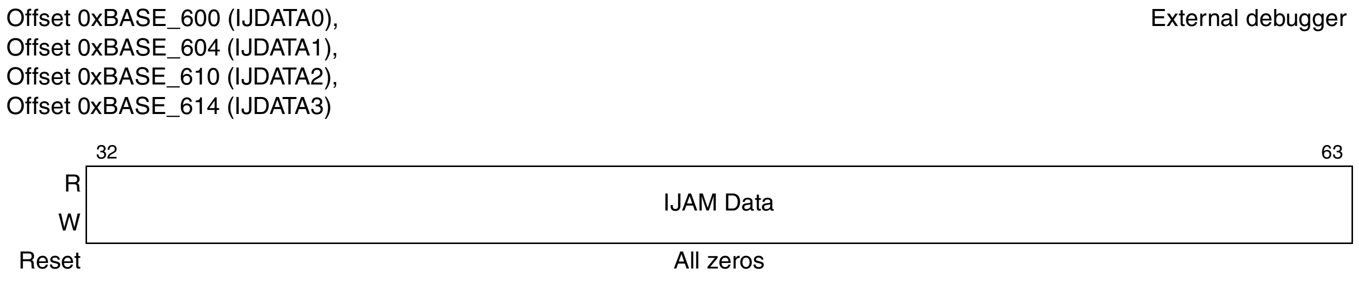

Offset 0xBASE_600 (IJDATA0),
Offset 0xBASE_604 (IJDATA1),
Offset 0xBASE_610 (IJDATA2),
Offset 0xBASE_614 (IJDATA3)
External debugger
32
63
R
W
IJAM Data
Reset
All zeros

Figure 9-23. IJAM data registers (IJDATA0–IJDATA3)

9.7 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 the e6500 core are described in Section 2.16, "Performance monitor registers (PMRs)."

9.8 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.

9.8.1 Performance monitor counter capture registers (PMCC0–PMCC5)

The performance monitor counter capture registers (PMCC0–PMCC5), shown in Figure 9-24, are 32-bit registers that capture the PMC*n* counter values based on a snapshot trigger signal. (See Section 9.5.11, "Performance Monitor Snapshot Configuration (PMSCR) register" for configuration.) Details on the performance monitor capture feature can be found in Section 9.12.4.2, "Processor performance monitor and program counter capture function."

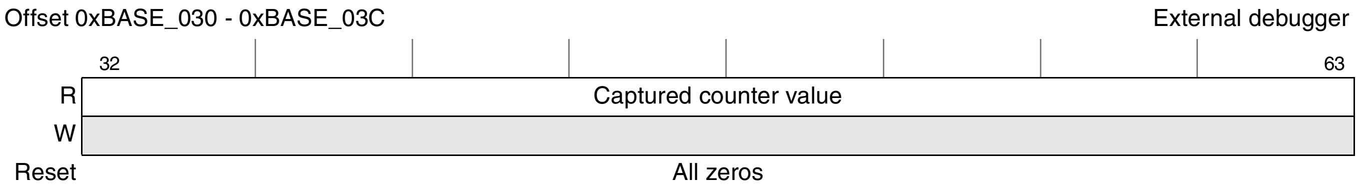


Figure 9-24. Performance monitor counter capture registers (PMCC0–PMCC5)

This table describes the PMCC field.

Table 9-21. PMCC0–PMCC5 field description

Bits	Name	Description
32–63	Counter Value	Value of the PMC*n* counter upon occurrence of the snapshot trigger

9.8.1.1 Program Counter Capture (PCC) register

PCC, shown in Figure 9-25, is a 64-bit register that captures the micro-architected program counter value based on a snapshot trigger signal. (See Section 9.5.11, "Performance Monitor Snapshot Configuration (PMSCR) register" for configuration.) For the e6500 core, the program counter is accurate to within two instruction windows from when the signal is detected by the processor and the two instructions at the bottom of the completion queue. Details on the performance monitor capture feature can be found in Section 9.12.4.2, "Processor performance monitor and program counter capture function."

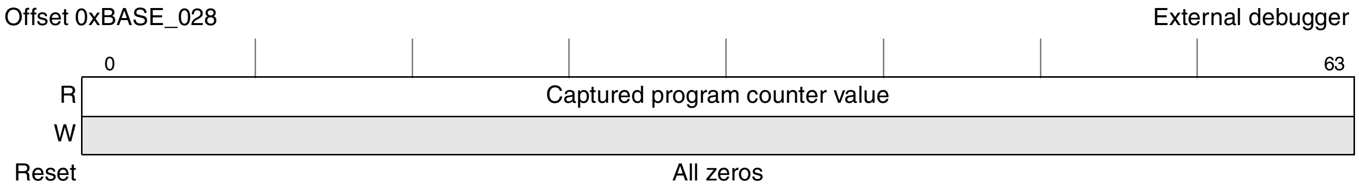


Figure 9-25. Program Counter Capture (PCC) register

This table describes the PCC field.

Table 9-22. PCC field description

Bits	Name	Description
0–63	Captured program counter value	Value of the program counter upon occurrence of the snapshot trigger

9.9 Debug events

Based on the terminology at the beginning of this chapter, debug conditions can be configured to cause debug events (setting of DBSR or EDBSR0), and debug events can result in debug interrupts or debug halt entry. In addition to configuring debug conditions to result in debug halt entry, there are several dedicated debug halt request events.

This chapter describes the debug conditions and how they may be configured to cause debug interrupts or debug halt entry. This chapter also provides details on the dedicated debug halt request events, as well as a prioritization scheme that is used to determine which debug events prevail in the case of simultaneous assertion.

9.9.1 Embedded hypervisor

In the presence of hypervisor software, debug events are modified to be suppressed when debug capabilities are enabled in the guest state. This prevents debug events from being recorded (and subsequent debug interrupts from occurring) when executing in the 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. See *EREF* for more details on the embedded hypervisor.

9.9.2 Internal and external debug modes

EREF specifies how the processor behaves in internal debug mode (IDM). This is when DBCR0[EDM] = 0 and DBCR0[IDM] = 1.

The architecture allows implementation-dependent behavior when in external debug mode (EDM) (DBCR0[EDM] = 1). In EDM, the processor behaves as follows:

- An **mtspr** that attempts to change DBCRs, IACs, DACs, or DBSR behaves as a nop if the resource is not allocated to IDM. An exception is that jamming an **mtspr** instruction alters these registers.
- DBSR is not updated for resources owned by EDM when a debug event occurs.
- When an externally allocated debug event occurs, the processor immediately halts. The debug interrupt is not taken for those debug events that are allocated to EDM. DSRR0, DSRR1, MSR, and ESR are not updated before halting. The NIA is not redirected to the first instruction of the debug interrupt handler.
- Upon halting for an EDM allocated 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 processor to halt.

Note that, in IDM, all synchronous debug events (all debug events except UDE, which is pended) are not recognized unless debug interrupts are enabled (MSR[DE] = 1). However, if BRT and ICMP are allocated to EDM and not masked in EDBSRMSK0, those events are always recognized and cause the processor to halt, even if MSR[DE] = 0.

9.9.3 Changing the debug facility state in internal debug mode

In general, care should be taken when changing the debug facility state. Changing that state requires synchronization. (See Section 3.3.3, “Synchronization requirements.”) The synchronization generally occurs naturally if changes are made in the debug interrupt handler when MSR[DE] = 0, and MSR[DE] transitions from 0 to 1 when the debug interrupt handler returns and other debug events are enabled. The synchronization also occurs naturally when context switch is performed and the debug facility registers are loaded from the incoming state. Such context switch operations should be performed with MSR[DE] = 0 and transitioning to 1 when the appropriate return from interrupt instruction is executed to transition to the newly loaded context.

Changing the debug facility state when the state might cause debug events in the currently executing instructions in the pipeline may have unpredictable results for software and should be avoided. For example, an **mtspr** instruction that enables an instruction address compare (IAC) for the current instruction, or instructions shortly after the current instruction, but before debug facility synchronization is accomplished may fail to record the debug event. Similarly, an **mtmsr** that changes MSR[DE], which allows debug events to be recorded, may not take effect until the synchronization is performed.

9.9.4 IAC, DAC, ICMP, BRT, IRPT, RET, CIRPT, CRET debug condition response table

This table lists responses for IAC*n*, DAC*n*, ICMP, BRT, IRPT, RET, CIRPT, and CRET conditions.

Table 9-23. Response—IAC*n*, DAC*n*, ICMP, BRT, IRPT, RET, CIRPT, CRET

EDM	IDM	EDBRAC0[x]	DBRCR0[x]	MSR[DE]	Actions		
					Set Status Bits	**Generate Interrupt of Halt**	**Notes**
0	0	x	x	x			
0	1	x	0	x			
0	1	x	1	0			
0	1	x	1	1	DBSR[x]	Debug (IVOR15) interrupt	
1	0	0	0	x			
1	0	0	1	x	EDBSR0[x]	Halt	
1	0	1	x	x			
1	1	0	0	x			
1	1	0	1	x	EDBSR0[x]	Halt	
1	1	1	0	x			
1	1	1	1	0			
1	1	1	1	1	DBSR[x]	Debug (IVOR15) interrupt	

9.9.5 Instruction address compare debug condition

The e6500 core implements IAC debug conditions as described in the architecture, with the following exception:

- Real Mode comparisons (DBCR1[IAC1ER] = 01 and DBCR1[IAC2ER] = 01) are not supported.

One or more instruction address compare debug conditions (IAC1-8) occur if they are enabled and execution of an instruction is attempted at an address that meets the criteria specified in DBCR0, DBCR5, IAC1 - IAC8. These debug conditions cause debug events to be recorded in DBSR if MSR[DE] = 1 and no higher priority exception exists. When MSR[DE] = 1, the IAC debug conditions are logged when the debug IAC interrupt is taken. When MSR[DE] = 0, IAC debug conditions are ignored. 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. See Section 2.14.5, “Debug Control 1 (DBCR1) register,” for details on the controls for the various IAC event modes.

IAC conditions are masked from generating IAC events if DBCR2[DACLINK1/2] bits 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[IAC], 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[IAC] if MSR[DE] = 1 and no higher priority exception exists.

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 ignored.

9.9.6 Data address compare debug condition

The e6500 core implements DAC debug conditions 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 where GS = 1. See *EREF* 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 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. 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. See

Section 2.14.6, “Debug Control 2 (DBCR2) register,” 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 no-ops 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 no-oped 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. These linked events are recorded in DBSR[DACR,DACW], according to which DAC comparator generated the debug condition. For the e6500 core, a DACLINK1/2 event only occurs if the DAC condition matches the first word of an **lmw** or **stmw** instruction.

In EDM, if no higher priority debug event is associated with the instruction, a DAC debug condition is recorded as a debug event in EDBSR0[DACR, DACW], 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[DACR,DACW] if MSR[DE] = 1 and no higher priority exception exists.

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 ignored.

9.9.7 Instruction complete debug condition

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 an exception whose interrupt would save the address of that instruction in the appropriate save/restore register 0. For example, a Trap instruction that 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 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 instruction execution.

Return-from-interrupt class instructions that enable or disable instruction complete debug events through the side effect of a change to MSR[DE] are not applied to the return instruction itself, but take 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.

9.9.8 Branch taken debug condition

A branch taken debug condition occurs if branch taken debug conditions are enabled (DBCR0[BRT] = 1) and execution of a branch instruction that is taken is attempted (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 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. Branch taken debug events are not recognized if MSR[DE] = 0 at the time of the branch instruction execution. A debug interrupt is generated, the execution of the branch instruction is suppressed, and DSRR0 is set to the address of the branch instruction.

9.9.9 Interrupt taken debug condition

An interrupt taken debug condition occurs if interrupt taken debug conditions are enabled (DBCR0[IRPT] = 1) and a non-debug, non-critical, non-machine check interrupt occurs. Only non-debug, non-critical, non-machine check class interrupts cause an interrupt taken debug condition. This condition is recorded in DBSR if MSR[DE] = 1. 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 is set to the address of the non-critical interrupt handler. No instructions at the noncritical interrupt handler execute.

In IDM, an interrupt taken debug condition is recorded as a debug event in DBSR[IRPT] if MSR[DE] = 1. If debug interrupts are enabled (MSR[DE] = 1), a debug interrupt is generated and the value saved in DSRR0 is the address of the non-critical interrupt handler. No instructions at the non-critical interrupt handler execute.

If debug interrupts are disabled (MSR[DE] = 0), the IRPT debug event is ignored.

9.9.10 Interrupt return debug condition

A return debug condition occurs if return debug conditions are enabled (DBCR0[RET] = 1), an attempt is made to execute an **rfi** instruction, and no other higher priority exception executing the **rfi** occurs. This condition causes the corresponding debug event to be recorded in DBSR if MSR[DE] = 1. 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 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. If debug interrupts are enabled (MSR[DE] = 1), a debug interrupt occurs provided no higher priority exception that is enabled to cause an interrupt exists. DSRR0 is set to the address of the **rfi** instruction.

If debug interrupts are disabled (MSR[DE] = 0) at the time of **rfi** execution (that is, before MSR is updated by the **rfi**), the RET event is ignored.

9.9.11 Critical interrupt taken debug condition

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. 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 is set to the address of the critical interrupt handler. No instructions at the critical interrupt handler execute.

In IDM, a critical interrupt taken debug condition is recorded as a debug event in DBSR[CIRPT] if MSR[DE] = 1.

If debug interrupts are enabled (MSR[DE] = 1), a debug interrupt occurs, provided no higher priority exception that is enabled to cause an interrupt exists, and the value saved in DSRR0 is the address of the critical interrupt handler. No instructions at the critical interrupt handler execute.

If debug interrupts are disabled (MSR[DE] = 0), the CIRPT debug event is ignored.

9.9.12 Critical return debug condition

A critical return debug condition occurs if critical return debug conditions are enabled (DBCR0[CRET] = 1), 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. 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 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. If debug interrupts are enabled (MSR[DE] = 1), a debug interrupt occurs, provided no higher priority exception that is enabled to cause an interrupt exists. DSRR0 is set to the address of the **rfci** instruction.

If debug interrupts are disabled (MSR[DE] = 0) at the time of **rfci** execution (that is, before MSR is updated by the **rfci**), the CRET event is ignored.

9.9.13 Unconditional debug event condition

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 does not have a corresponding enable bit in DBCR0. This condition causes the corresponding debug event to be recorded in DBSR or EDBSR0, regardless of the setting of MSR[DE].

If UDE is allocated to the external debug host (EDM) and is not masked, 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 is set to the address of the next instruction to be executed.

If UDE is allocated to the internal debug agent (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 DSRR0 is set to the address of the instruction that would be executed next (were it not for the occurrence of the debug interrupt).

If MSR[DE] = 0 when an unconditional debug condition occurs, the condition is recorded as an event in DBSR[UDE]. In the case of a delayed debug interrupt, DSRR0 contains the address of the instruction following the one that enabled debug interrupts.

This table lists responses for UDE conditions.

Table 9-24. UDE condition responses

EDM	IDM	EDBRAC0[UDE]	EDBSRMSK0[UDEM]	MSR[DE]	Actions		
					Set Status Bits	**Generate Interrupt of Halt**	**Notes**
0	0	x	x	x			
0	1	x	x	0	DBSR[UDE]		The Debug (IVOR15) interrupt is taken once the MSR[DE] bit is set.
0	1	x	x	1	DBSR[UDE]	Debug (IVOR15) interrupt	
1	0	0	0	x	EDBSR0[UDE]	Halt	
1	0	0	1	x			
1	0	1	x	x			
1	1	0	0	x	EDBSR0[UDE]	Halt	
1	1	0	1	x	EDBSR0[UDE]		
1	1	1	x	0	DBSR[UDE]		The Debug (IVOR15) interrupt is taken once the MSR[DE] bit is set.
1	1	1	x	1	DBSR[UDE]	Debug (IVOR15) interrupt	

9.9.14 TRAP debug condition

A TRAP debug condition occurs if TRAP debug conditions are enabled, a Trap instruction (**tw**, **twi**) is executed, and the conditions specified by the instruction for the Trap are met.

If a Trap is allocated to the external host debugger (EDM) and is not masked, 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 is set to the address of the Trap instruction. MSR[DE] has no effect on the updates to EDBSR0.

If a Trap is allocated to the internal debug agent (IDM), a Trap debug condition is recorded as a debug event in DBSR[TRAP] if MSR[DE] = 1 and no higher priority exception exists.

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.

In cases where the Trap instruction is not configured to cause a debug interrupt, or when debug interrupts are disabled, the TRAP debug event generates a program interrupt.

This table lists responses for TRAP debug conditions.

Table 9-25. TRAP debug condition responses

EDM	IDM	EDBRAC0[TRAP]	DBRCR0[TRAP]	MSR[DE]	Actions		
					Set Status Bits	Generate Interrupt of Halt	Notes
0	0	x	x	x		Program (IVOR6) interrupt	
0	1	x	0	x		Program (IVOR6) interrupt	
0	1	x	1	0		Program (IVOR6) interrupt	
0	1	x	1	1	DBSR[TRAP]	Debug (IVOR15) interrupt	
1	0	0	0	x		Program (IVOR6) interrupt	
1	0	0	1	x	EDBSR0[TRAP]	HALT	
1	0	1	0	x		Program (IVOR6) interrupt	
1	0	1	1	x		Program (IVOR6) interrupt	
1	1	0	0	x		Program (IVOR6) interrupt	
1	1	0	1	x	EDBSR0[TRAP]	HALT	
1	1	1	0	0		Program (IVOR6) interrupt	
1	1	1	0	1		Program (IVOR6) interrupt	
1	1	1	1	0		Program (IVOR6) interrupt	
1	1	1	1	1	DBSR[TRAP]	Debug (IVOR15) interrupt	

9.9.15 Debugger Notify Interrupt (DNI) debug condition

The **dni** instruction (see Section 3.5, “Debug instruction model”) provides a mechanism to force a debug interrupt. This instruction is not privileged and may be executed while the processor is at any privilege level. It may be compiled into code, or a debugger may substitute **dni** for another instruction at a location where a breakpoint is desired.

NOTE

Because the **dni** instruction executes as a NOP when IDM and EDM are disabled, care should be taken if **dni** is substituted by the debugger for another instruction.

Bits 11-15 of the **dni** instruction (DCTL) are implementation defined. For the e6500 core, bit 15 of the **dni** instruction is used to assert the DNI watchpoint event for event cross triggering. Thus, if bit 15 of the **dni** instruction is set, the DNI watchpoint asserts on execution of the instruction. However, if **dni** bit 15 is zero, the watchpoint event does not assert.

A **dni** debug condition occurs if the **dni** instruction is executed.

This table summarizes the responses to a DNI condition.

Table 9-26. DNI debug condition responses

EDM	IDM	EDBRAC0[DNI]	EDBSRMSK0[DNIM]	EDBCR0[DNI_CTL]	MSR[DE]	Actions		
						Set Status Bits	Generate Interrupt of Halt	Notes
0	0	x	x	x	x			**dni** executes as a NOP
0	1	x	x	x	0			**dni** executes as a NOP
0	1	x	x	x	1	DBSR[DNI]	Debug (IVOR15) interrupt	
1	0	0	0	0	0			**dni** executes as a NOP
1	0	0	0	0	1	EDBSR0[DNI]	Halt	
1	0	0	0	1	x	EDBSR0[DNI]	Halt	
1	0	0	1	x	x			**dni** executes as a NOP
1	0	1	x	x	x			**dni** executes as a NOP
1	1	0	0	0	0			**dni** executes as a NOP
1	1	0	0	0	1	EDBSR0[DNI]	Halt	
1	1	0	0	1	x	EDBSR0[DNI]	Halt	
1	1	0	1	0	0			**dni** executes as a NOP
1	1	0	1	0	1			**dni** executes as a NOP
1	1	0	1	1	x			**dni** executes as a NOP
1	1	1	x	x	0			**dni** executes as a NOP
1	1	1	x	x	1	DBSR[DNI]	Debug (IVOR15) interrupt	

If a **dni** results in a debug interrupt, DSRR0 is set to the address of the **dni** instruction, and instruction execution begins at the debug interrupt handler address. See Section 4.9.16, "Debug interrupt—IVOR15."

If a **dni** results in a debug halt, a **dni** debug condition is recorded as a debug event in EDBSR0[dni], the execution of the dni instruction is suppressed, the processor halts, and NIA is set to the address of the dni instruction.

9.9.16 Dedicated debug halt request events

In addition to being able to configure the debug conditions to cause a debug halt (which results in PRSR[DE_HALT] being set), there are several dedicated debug halt request events that can be used to place the processor in debug halt mode. When a debug halt request is asserted, the processor enters debug halt mode as a result if the request is not masked.

The dedicated debug halt requests do not affect EDBSR0 status bits but are reflected in the PRSR.

9.9.16.1 Debug Halt Request (*corex_dbg_halt_thrdn*) input

Assertion of the *core**x**_dbg_halt* input causes the processor to enter the debug halted state. PRSR[EDB_HALT] is set to indicate that *core**x**_dbg_halt* has been asserted, and PRSR[HALTED] indicates that the processor is in the halted state.

9.9.16.2 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]. This instruction is enabled by writing 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 and EDBSRMSK0[DNHM] = 0 causes the processor to halt. 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.

Execution of this instruction when EDBCR0[DNH_EN] = 1 and EDBSRMSK0[DNHM] = 1 causes the **dnh** instruction to be executed as a NOP (that is, the processor does not halt), and a DNH watchpoint is generated (if enabled).

The 10-bit DUIS field is no longer supported in the e6500 core. Furthermore, bits 11-15 of the **dnh** instruction are now implementation defined. For the e6500 core, bit 15 is used to assert the **dnh** watchpoint event, which may be used for triggering internal resources, or routed to the event output (EVTO0-4) signals for event cross-triggering.

Execution of the **dnh** instruction when EDBCR0[DNH_EN] = 0 causes an illegal instruction exception (IVOR6).

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.

9.9.16.3 Cross-thread debug halt requests

By default, if one thread receives a debug halt request (DNH_HALT, DE_HALT, or EDB_HALT), both threads halt. The PRSR of the thread that received the original halt request indicates the original halt request, whereas the PRSR of the other thread indicates it halted due to a cross-thread halt (CT_HALT).

It should be noted that both threads receive the debug halt request (original debug halt request and cross-thread debug halt request) simultaneously and enter debug halt state as soon as the currently executing instructions complete.

The cross-thread debug halt request can be masked by setting the Disable Cross-Thread Halt bit in each thread's external debug control register (EDBCR0[DIS_CTH]).

9.9.17 Simultaneous debug event priorities

A priority mechanism is provided to resolve multiple debug event assertions within the same cycle. The priorities affect which debug event(s) are recognized between multiple event occurrences within the same debug event owner (either IDM/IDM or EDM/EDM) or between multiple debug event owners, such as an internal debug agent and an external debug host (IDM/EDM). The priorities are used to define which bits are set within the DBSR (IDM) or EDBSR0 (EDM), the PC value recorded to DSRR0 (IDM) or NIA (EDM), and whether we call a debug exception (IDM) or halt (EDM).

The following rules are used:

- In the event of simultaneous debug events (regardless of whether they are owned by EDM or IDM), the highest priority debug event is recognized and all lower priority debug events are ignored (except for UDE, which results in its DBSR bit being set).
- If there are multiple events at the same priority level (and none at a higher priority):
  - EDM owned debug events are recognized over IDM debug events.
  - In some cases, multiple simultaneous events can set multiple DBSR bits.
    - Multiple IAC and DAC DBSR bits can be set simultaneously.
    - UDE is asynchronous and its DBSR can be set at any time.
    - All other simultaneous events are ignored.
  - In all cases, multiple simultaneous debug halting events set multiple EDBSR0 bits.

This table shows the relative debug event priorities. It should be noted that these priorities are defined within Section 4.8, "Interrupt processing," but are replicated here for this discussion.

Pre

Table 9-27. Debug event priorities

Priority Level (same as exceptions)		Debug Event	Pre or Post Completion[1]
Low <---------- Relative Priority ----------> High	N/A	Debug Halt Request[2]	N/A[3]
	2	CIRPT[4]	N/A[3]
		IRPT[4]	N/A[3]
		UDE[4]	N/A[3]
	8	IAC	Pre
	12	DNI	Pre
		TRAP	Pre
	18	RET	Pre
		CRET	Pre
		BRT	Pre
	19	DAC	Pre
	21	ICMP	Post

1 PC value copied into DSRR0 (exception), or NIA (halt) is current (Pre) or next (Post) instruction.

2 Debug Halt Request - From corex_dbg_halt, dnh instruction, or cross-thread halt.

3 N/A - Not Applicable (i.e. Effectively asynchronous with regards to instruction execution).

4 Asynchronous exceptions cannot be sampled during the post completion of the instruction for which instruction complete (ICMP) is set. This is because the ICMP cannot be separated from the completion of the instruction. Thus, ICMP can appear as a higher priority than this asynchronous interrupt.

9.9.17.1 Simultaneous debug event handing—events within same owner

When two debug events from the same debug owner assert simultaneously, the event of the highest priority is recognized (its corresponding DBSR or EDBSR0 bit is set), and any lower priority events either update the EDBSR0 (for halting events) or are ignored (for interrupt events). The exception to this rule is UDE, which, even when lower priority, sets the DBSR for later interrupt processing.

As an example, if an IAC and DNI are owned by an internal debug agent (IDM) and assert within the same cycle, only the IAC debug event is recognized, and the DNI event is ignored. Thus, the respective IAC bit is set in the DBSR when the debug exception is entered. Once the IAC event condition has been handled and a return is made to normal code execution, the **dni** instruction has a chance to re-execute.

Similarly, if both events are owned by an external debug host (EDM) and occurr in the same cycle, the processor enters debug halt mode with the respective IAC bit set in EDBSR0.

However, if the two debug events of the same priority from the same debug owner assert simultaneously, then each is recognized (corresponding DBSR or EDBSR0 bits set for each debug event of the highest priority).

As an example of this, if TRAP and DNI are owned by an internal debug agent (IDM) and assert within the same cycle, both TRAP and DNI debug events are recognized and their respective bits are set in the DBSR when the debug exception is entered.

UDE is an exception to this and is recorded in DBSR, regardless if a higher priority event occurs simultaneously or not.

9.9.17.2 Simultaneous debug event handing—events of different owners

Because it is possible that an external debugger (EDM) owned resource can produce a debug event in conjunction with a software debug agent (IDM) owned resource producing a different debug event simultaneously, a simple priority ordering mechanism is implemented between the two owners.

When simultaneous events are of a different priority level and from different owners (EDM and IDM), the highest priority debug event is recognized and the lower priority debug event is ignored.

When simultaneous events are of the same priority level but from different owners (EDM and IDM), the EDM debug event is recognized and the IDM debug event is ignored.

9.10 External debug interface

External debug support is supplied through a memory-mapped interface, which allows access to internal processor registers, arrays, and other system state while the processor is halted. EDM provides the ability to enter the halt state when a debug event occurs. This capability can be used to perform single-step operations from the external debug tool.

9.10.1 Processor run states

This section discusses the following processor run states:

- Section 9.10.1.1, “Halt”
- Section 9.10.1.2, “Stop (freeze)”
- Section 9.10.1.3, “Wait”
- Section 9.10.1.4, “Thread disabled”
- Section 9.10.1.5, “Entering/exiting processor run states”

9.10.1.1 Halt

When the e6500 core is in the debug halted state, the clocks are still running, but the processor is not fetching or executing instructions. While in this state, an external debugger can jam instructions into the pipeline, and they are executed. The processor also continues to receive snoops and maintains cache coherency.

Assertion of *core***x***_pm_halt* causes the processor to enter the halted state. PRSR[PM_HALT] is asserted to indicate that *core***x***_pm_halt* has been asserted, and PRSR[HALTED] indicates that the processor is in the halted state. When *core***x***_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 processor resumes immediately.

There are several mechanisms that halt the processor. These are described in the following table.

Table 9-28. Methods for halting the processor

Halt Condition	Classification	Enable	Documentation
Assertion of *core**x**_pm_halt*	Power Management	Always enabled	—
Assertion of *core**x**_dbg_halt*	Debug	Always enabled	—
DNH	Debug	EDBCR0[DNH_EN]	Section 9.9.16.2, "Debugger Notify Halt (dnh) instruction"
DE	Debug	EDBCR0[EDM]	Section 9.9.2, "Internal and external debug modes"

Most external debug operations can only be performed when the processor is halted. Note that if the processor is halted only because *core**x**_pm_halt* is asserted (that is, no other halt requests are active in PRSR), it resumes immediately if *core**x**_pm_halt* is deasserted. Therefore, the processor should always be halted with some other debug mechanism (for example, setting a system debug event halt) before accessing the contents of the processor.

The Processor Run Status (PRSR) register indicates whether or not the processor is halted for debug.

9.10.1.1.1 Watchdog timer during debug halted state

On the e6500 core, when the core is in debug halt mode, the watchdog timer continues to run. However, the watchdog interrupt and watchdog reset are blocked from occurring by holding the TSR[WIS] and TSR[ENW] bits in reset (TSR state 00) while the core is in debug halt mode. When the core exits debug halt mode (to continue software execution), those bits are no longer held in reset, thus, allowing subsequent time-outs to transition the state machine as normal.

9.10.1.2 Stop (freeze)

When the e6500 core 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**x**_stop* causes the processor to enter the stopped state. PRSR[PM_STOP] is asserted to indicate that *core**x**_stop* has been asserted, and PRSR[STOPPED] indicates that the processor is in the stopped state. When *core**x**_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 processor transitions immediately to the appropriate halted or running state.

There are several mechanisms that can stop the processor. These are described in the following table.

Table 9-29. Methods for stopping the processor

Stop Condition	Classification	Enable	Documentation
Assertion of *core**x**_stop*	Power Management	Always enabled	Section 8.2, "Power management signals"

9.10.1.3 Wait

When the processor executes the **wait** instruction, it discontinues fetching and executing instructions and waits for an asynchronous interrupt or the reservation to be cleared. 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**x**_resume* signal is asserted to exit the halted state, the processor 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.

9.10.1.4 Thread disabled

Upon receiving a debug halt request, the thread that receives the debug halt request enters the debug halted state, regardless of whether the thread is enable or disabled. During the debug halted state, IJAM operations work as normal on halted threads. On exiting the debug halted state, a thread, which was disabled prior to debug halt mode entry, resumes to the thread disabled state.

During debug halt mode, the external debugger can IJAM instructions to enable or disable a thread. Thread disabled or enabled status can be determined by reading TENSR (see Section 2.15.1.5, "Thread Enable Status (TENSR) register)."

9.10.1.5 Entering/exiting processor run states

The e6500 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 *core**x**_pm_halt* or *core**x**_stop* is toggled while the processor is under control of a debugger.

Debug operations should not be performed while the processor is halted/stopped only due to power management. If the processor has been halted or stopped only for power management, the debugger should assert *core**x**_dbg_halt* before executing debug operations.

When the processor is running, the SoC should use the following sequence to enter the power management stopped state:

1. Assert *core**x**_pm_halt.*

2. Wait for *core**x**_halted* to be asserted by the processor.
3. Assert *core**x**_stop*.

This ensures that the processor is left in a recoverable state when the clocks are stopped. For the e6500 core, when *core**x**_pm_halt* and *core**x**_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 of the other 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**x**_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**x**_resume*.

Normally, when the processor has been halted for power management by asserting *core**x**_pm_halt*, the processor resumes execution when *core**x**_pm_halt* is deasserted. Similarly, the processor normally exits the power management stopped state whenever *core**x**_stop* is deasserted. However, if the processor has been halted or stopped for a halt or stop condition in the debug class, deassertion of *core**x**_pm_halt* or *core**x**_stop* does not cause the processor to resume until *core**x**_resume* is asserted.

If *core**x**_resume* is asserted while *core**x**_pm_halt* or *core**x**_stop* is asserted, the processor remains in the halted or stopped for power management state.

If any of the debug related halt status bits are set in PRSR indicating whether or not the processor has been halted or stopped for a debug condition, *core**x**_resume* must be asserted before the processor resumes execution.

If the processor has been halted or stopped only by assertion of *core**x**_pm_halt* or *core**x**_stop*, simply releasing *core**x**_pm_halt* or *core**x**_stop* allows the processor to resume execution.

If the processor 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 processor.
4. Stop the processor.

9.10.2 Single-step

An external development tool can single-step 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:

- Executes and completes one instruction, then halts before executing the next instruction.
- Executes one instruction and takes a synchronous interrupt, then halts before executing the first instruction of the interrupt handler.
- Immediately takes an asynchronous interrupt and halts on the first instruction of the interrupt handler.

Therefore, to single-step:

1. Set ICMP, IRPT, and CIRPT.
2. Set EDM.
3. Clear PRSR.
4. Resume.

Note that PRSR must be cleared prior to each resume command.

9.10.3 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):

- 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 processor
  - Direct access to memory

9.10.3.1 Memory-mapped access

Addressing the debug/expert resources through the memory-mapped interface entails driving a base address for the e6500 processor (BASE), a Thread Select (TS), a Functional Group Select (GID), and a register index for a specific register. The TS routes the access to the selected thread's registers within the core. The GID determines what class of resource is to be accessed, while the register index determines which resource within the group to access.

See the SoC reference manual for specifics on accessing the internal memory-mapped resources.

This figure shows the address bit fields used in accessing debug resources.

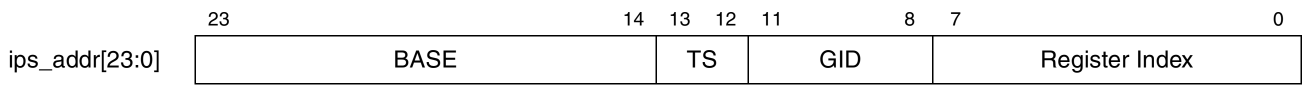


Figure 9-26. Debug resource access

The TS bit field is defined as follows.

Table 9-30. Thread Select (TS)

TS	Thread
2'b00	0
2'b01	1
2'b10	Reserved
2'b11	Reserved

Table 9-31 summarizes the debug resource memory map and is replicated for each thread.

The "Access Restrictions" column in Table 9-31 can be interpreted as follows:

- If there are no restrictions listed, then software and the external debugger can access the corresponding resource through the memory-mapped interface.
- E = External debugger only (not accessible to software as an SPR). Software can read this register via memory-mapped accesses but should not write it.
- H = Access allowed when processor is halted only. Writes to this register while the core is running are ignored and reads are unpredictable.
- O = The register is allocated as a whole within the EDBRAC0 register (when EDM is enabled) to either an internal debug agent (accessible as an SPR) or an external debug host (memory-mapped accessible). The owner of the register has read/write access. Non-owners have read only access and their writes are ignored.
- N= The external debugger always has read and write access to the register. An internal debug agent always has read access, but only has write access when the resource is allocated to the internal debug agent as a whole within the EDBRAC0 register (or when EDM is disabled).
- S = The register is shared between the internal and external debugger and each bit is assigned on a bit-by-bit basis according to the allocations in the EDBRAC0 register (when EDM is enabled). The bit's owner has read/write access to that bit. Non-owners have read only access of that bit. Writes to non-owned bits are ignored.
- T0 = Accessible in Thread 0's map only.
- BT= The register is shared by both threads and accessible by both threads.

Table 9-31. Debug resource address map

Functional Group ID ips_addr[11:8]	Register Index ips_addr[7:0]	Resource	Access Type	Access Restrictions	Service Data Width	Reset Source
0x0 Debug Status	0x00	Processor Run Status (PRSR) register	R/W[1]	E	32	POR
	0x04	Skyblue Status (SBSR1) register	R	T0	32	HRESET
	0x08	Machine State (MSR) register	R/W	E, H	32	HRESET
	0x0c	External Debug Status (EDBSR0) register	R	E	32	HRESET
	0x10	External Debug Status (EDBSR1) register	R	E	32	HRESET
	0x14	External Debug Exception Syndrome (EDESR) register	R	E	32	HRESET
	0x18	Processor Version (PVR) register	R		32	N/A
	0x1c	External Debug Status Mask 0 (EDBSRMSK0) register	R/W	E	32	POR
	0x20 - 0x24	Reserved				
	0x28	Program Counter Capture (PCC) register - Upper 32-bits	R	E	32	POR
	0x2c	Program Counter Capture (PCC) register - Lower 32-bits	R	E	32	POR
	0x30	Perfmon Capture Count 0 (PMCC0) register	R	E	32	POR
	0x34	Perfmon Capture Count 1 (PMCC1) register	R	E	32	POR
	0x38	Perfmon Capture Count 2 (PMCC2) register	R	E	32	POR
	0x3c	Perfmon Capture Count 3 (PMCC3) register	R	E	32	POR
	0x40	Perfmon Capture Count 4 (PMCC4) register	R	E	32	POR
	0x44	Perfmon Capture Count 5 (PMCC5) register	R	E	32	POR
	0x48-0xfc	Reserved				
	0xfc	Processor Debug Information (PDIR) register	R	E	32	POR
0x1 Debug Control	0x00	External Debug Control 0 (EDBCR0) register	R/W	E	32	POR
	0x04 - 0x1c	Reserved				
	0x20	Extended External Debug Control 0 (EEDCR0) register	R/W	E, T0	32	POR
	0x24 - 0x2c	Reserved				
	0x30	External Debug Resource Allocation Control 0 (EDBRAC0) register	R/W	E	32	POR

Table 9-31. Debug resource address map (continued)

Functional Group ID ips_addr[11:8]	Register Index ips_addr[7:0]	Resource	Access Type	Access Restrictions	Service Data Width	Reset Source
0x1 Debug Control	0x34	External Debug Resource Request 0 (EDBRR0) register	R/W	E	32	POR
	0x38	Debug Resource Request 0 (DBRR0) register	R		32	HRESET
	0x3c	Reserved				
	0x40	Debug Control 0 (DBCR0) register	R/W	S	32	HRESET
	0x44-0x4c	Reserved				
	0x50	Perfmon Counter 0 (PMC0) register	R/W	O	32	HRESET
	0x54	Perfmon Counter 1 (PMC1) register	R/W	O	32	HRESET
	0x58	Perfmon Counter 2 (PMC2) register	R/W	O	32	HRESET
	0x5c	Perfmon Counter 3 (PMC3) register	R/W	O	32	HRESET
	0x60	Perfmon Counter 4 (PMC4) register	R/W	O	32	HRESET
	0x64	Perfmon Counter 5 (PMC5) register	R/W	O	32	HRESET
	0x68-0x6c	Reserved				
	0x70	Perfmon Local Control a0 (PMLCa0) register	R/W	O	32	HRESET
	0x74	Perfmon Local Control a1 (PMLCa1) register	R/W	O	32	HRESET
	0x78	Perfmon Local Control a2 (PMLCa2) register	R/W	O	32	HRESET
	0x7c	Perfmon Local Control a3 (PMLCa3) register	R/W	O	32	HRESET
	0x80	Perfmon Local Control a4 (PMLCa4) register	R/W	O	32	HRESET
	0x84	Perfmon Local Control a5 (PMLCa5) register	R/W	O	32	HRESET
	0x88-0x8c	Reserved				
	0x90	Perfmon Local Control b0 (PMLCb0) register	R/W	O	32	HRESET
	0x94	Perfmon Local Control b1 (PMLCb1) register	R/W	O	32	HRESET
	0x98	Perfmon Local Control b2 (PMLCb2) register	R/W	O	32	HRESET
	0x9c	Perfmon Local Control b3 (PMLCb3) register	R/W	O	32	HRESET
	0xa0	Perfmon Local Control b4 (PMLCb4) register	R/W	O	32	HRESET
	0xa4	Perfmon Local Control b5 (PMLCb5) register	R/W	O	32	HRESET
	0xa8-0xac	Reserved				
	0xb0	Perfmon Global Control 0 (PMGC0) register	R/W	O	32	HRESET
	0xb4-0xfc	Reserved				
0x3 Clock Control/Status	0x00 - 0xfc	Reserved				

Table 9-31. Debug resource address map (continued)

Functional Group ID ips_addr[11:8]	Register Index ips_addr[7:0]	Resource	Access Type	Access Restrictions	Service Data Width	Reset Source
0x4 Nexus	0x00 - 0x04	Reserved				
	0x08	Nexus Development Control 1 (DC1) register	R/W	N	32	POR
	0x0c	Nexus Development Control 2 (DC2) register	R/W	N	32	POR
	0x10	Nexus Development Control 3 (DC3) register	R/W	N	32	POR
	0x14	Nexus Development Control 4 (DC4) register	R/W	N	32	POR
	0x18 - 0x28	Reserved				
	0x2c	Watchpoint Trigger 1 (WT1) register	R/W	N	32	POR
	0x30	Watchpoint Trigger 2 (WT2) register	R/W	N	32	POR
	0x34 - 0x54	Reserved				
	0x58	Watchpoint Mask (WMSK) register	R/W	N	32	POR
	0x5c	Nexus Overrun Control (OVCR) register	R/W	N	32	POR
	0x60 - 0xbc	Reserved				
	0xc0	Reload Counter Configuration (RCCR) register	R/W	N	32	POR
	0xc4	Reload Counter Value (RCVR) register	R/W	N	32	POR
	0xc8	Perf Mon Snapshot Configuration (PMSCR) register	R/W	N	32	POR
	0xcc - 0xfc	Reserved				

Table 9-31. Debug resource address map (continued)

Functional Group ID ips_addr[11:8]	Register Index ips_addr[7:0]	Resource	Access Type	Access Restrictions	Service Data Width	Reset Source
0x5 IAC / DAC	0x00	Debug Control 1 (DBCR1) register	R/W	S	32	HRESET
	0x04	Debug Control 2 (DBCR2) register	R/W	O	32	HRESET
	0x08	Reserved				
	0x0c	Debug Control 4 (DBCR4) register	R/W	O	32	HRESET
	0x10	Debug Control 5 (DBCR5) register	R/W	S	32	HRESET
	0x14 - 0x4c	Reserved				
	0x50	Instruction Address Compare 1 (IAC1) register - upper	R/W	O	32	HRESET
	0x54	Instruction Address Compare 1 (IAC1) register - lower	R/W	O	32	HRESET
	0x58	Instruction Address Compare 2 (IAC2) register - upper	R/W	O	32	HRESET
	0x5C	Instruction Address Compare 2 (IAC2) register - lower	R/W	O	32	HRESET
	0x60	Instruction Address Compare 3 (IAC3) register - upper	R/W	O	32	HRESET
	0x64	Instruction Address Compare 3 (IAC3) register - lower	R/W	O	32	HRESET
	0x68	Instruction Address Compare 4 (IAC4) register - upper	R/W	O	32	HRESET
	0x6C	Instruction Address Compare 4 (IAC4) register - lower	R/W	O	32	HRESET
	0x70	Instruction Address Compare 5 (IAC5) register - upper	R/W	O	32	HRESET
	0x74	Instruction Address Compare 5 (IAC5) register - lower	R/W	O	32	HRESET
	0x78	Instruction Address Compare 6 (IAC6) register - upper	R/W	O	32	HRESET
	0x7C	Instruction Address Compare 6 (IAC6) register - lower	R/W	O	32	HRESET
	0x80	Instruction Address Compare 7 (IAC7) register - upper	R/W	O	32	HRESET
	0x84	Instruction Address Compare 7 (IAC7) register - lower	R/W	O	32	HRESET
	0x88	Instruction Address Compare 8 (IAC8) register - upper	R/W	O	32	HRESET
	0x8C	Instruction Address Compare (IAC8) register - lower	R/W	O	32	HRESET
	0x90	Data Address Compare 1 (DAC1) register - upper	R/W	O	32	HRESET
	0x94	Data Address Compare 1 (DAC1) register - lower	R/W	O	32	HRESET
	0x98	Data Address Compare 2 (DAC2) register - upper	R/W	O	32	HRESET
	0x9C	Data Address Compare 2 (DAC2) register - lower	R/W	O	32	HRESET
	0xa0 - 0xfc	Reserved				

Table 9-31. Debug resource address map (continued)

Functional Group ID ips_addr[11:8]	Register Index ips_addr[7:0]	Resource	Access Type	Access Restrictions	Service Data Width	Reset Source
0x6 Instruction Jamming	0x00	Instruction Jamming Data 0 (IJDATA0) register	R/W	E, H	32	HRESET
	0x04	Instruction Jamming Data 1 (IJDATA1) register	R/W	E, H	32	HRESET
	0x08	Instruction Jamming Configuration (IJCFG) register	R/W	E, H	32	POR
	0x0c	Instruction Jamming Instruction (IJIR) register	R/W	E, H	32	HRESET
	0x10	Instruction Jamming Data 2 (IJDATA2) register	R/W	E, H	32	HRESET
	0x14	Instruction Jamming Data 3 (IJDATA3) register	R/W	E, H	32	HRESET
	0x18 - 0xfc	Reserved				
0x7 - 0x9	Reserved	Reserved				
0xc - 0xd	Reserved	Reserved				
0xe - 0xf	Reserved	Reserved for LBIST (not currently implemented)				

[1] Portions of PRSR support write-1-to-clear. All other fields are read-only.

9.10.3.2 Special-purpose register access (Nexus only)

Nexus trace resources can also be accessed through e6500 SPRs—specifically, the Nexus SPR Access Configuration (NSPC) register and the Nexus SPR Access Data (NSPD) register.

NOTE

NSPC and NSPD SPR's are only accessible by the software when EDM is not enabled or when EDM is enabled and EDBRAC0[TRACE] = 1.

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, see Section 2.14.12, "Nexus SPR access registers."

Once the specific Nexus resource has been selected, software can then access NSPD by executing an **mtspr** instruction (for register writes) or an **mfspr** (for register reads).

9.10.4 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 9.10.4.1, “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 9.10.4.5, “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.

9.10.4.1 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 9.10.4.2, “Instruction jamming input,” for a description of the IJAM input.

A debugger wishing to alter the value of a GPR jams a load instruction. The debugger places the desired load data in the IJAM IR register, and it writes 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 jams a store instruction. The debugger writes 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 then reads 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 the following table are supported when IJMODE = 1.

Table 9-32. Load/store IJAM transfers (when IJMODE = 1)

processor Registers	**Action**	**Instruction**	**IJAM Transfer**	
			From	**To**
Floating Point Register (FPR)	Read	**stfd fr**S, 0(0)	**fr**S[0–31]	IJDATA0[0–31]
			frS[32–63]	IJDATA1[0–31]
			32’0[1]	IJDATA2[0–31]
			32’0[1]	IJDATA3[0–31]
	Write	**lfd fr**D, 0(0)	IJDATA0[0–31]	**fr**D[0–31]
			IJDATA1[0–31]	**fr**D[32–63]
	Read	**stfs fr**S, 0(0)	32’0[1]	IJDATA0[0–31]
			frS[32–63]	IJDATA1[0–31]
			32’0[1]	IJDATA2[0–31]
			32’0[1]	IJDATA3[0–31]
	Write	**lfs fr**D, 0(0)	IJDATA1[0–31]	**fr**D[32–63]

Table 9-32. Load/store IJAM transfers (when IJMODE = 1) (continued)

processor Registers	Action	Instruction	IJAM Transfer	
			From	To
General Purpose Register (GPR)	Read	**std** **r**S,0(0)	**r**S[0–31]	IJDATA0[0–31]
			rS[32–63]	IJDATA1[0–31]
			32’0[1]	IJDATA2[0–31]
			32’0[1]	IJDATA3[0–31]
		stw **r**S,0(0)	32’0[1]	IJDATA0[0–31]
			rS[32–63]	IJDATA1[0–31]
			32’0[1]	IJDATA2[0–31]
			32’0[1]	IJDATA3[0–31]
	Write	**ld** **r**D,0(0)	IJDATA0[0–31]	**r**D[0–31]
			IJDATA1[0–31]	**r**D[32–63]
		lwz **r**D,0(0)	IJDATA1[0–31]	**r**D[32–63]
		lhz **r**D,0(0)	IJDATA1[16–31]	**r**D[48–63]
			16’b0	**r**D[32–47]
		lbz **r**D,0(0)	IJDATA1[24–31]	**r**D[56–63]
			24’b0	**r**D[32–55]
Vector Register (VR)	Read	**stvx** **v**S, 0,0	**v**S[0–31]	IJDATA0[0–31]
			vS[32–63]	IJDATA1[0–31]
			vS[64–95]	IJDATA2[0–31]
			vS[96–127]	IJDATA3[0–31]
	Write	**lvx** **v**D, 0,0	IJDATA0[0–31]	**v**D[0–31]
			IJDATA1[0–31]	**v**D[32–63]
			IJDATA2[0–31]	**v**D[64–95]
			IJDATA3[0–31]	**v**D[96–127]

[1] Note that stores to IJDATA result in updates to all IJDATA registers regardless of the size of the access. IJDATA registers, which are not receiving data directly from a GPR or FPR, is set to 0, clearing any stale data previously written to IJDATA.

9.10.4.2 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.

These registers can be accessed through a memory-mapped access individually or through a block transfer. On the e6500 core, incorrect register settings result in **unpredictable** results. IJAM register descriptions can be found in Section 9.6, "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 processors 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 40-bit physical address or page attributes (WIMGE bits) for the load/store instruction. When the processor is operating in 32-bit mode, 8 more address bits are needed to form a 40-bit physical address. These additional 8 bits are supplied by the IJRA field of IJCFG. The 40-bit address is formed by prepending the 8-bit IJRA field to the effective address calculated by the jammed load/store instruction (PA[24:63] = IJRA[0:7] || EA[32:63]). When the processor is operating in 64-bit mode, only the lower 40-bits of the effective address are used to generate the physical address. The upper 24 bits are always 0.

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 that 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 9-33. 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

9.10.4.3 Supported instruction jamming instructions

Table 9-34 lists instructions that 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 = 32'b0.

This table also includes all instructions that are capable of using options in IJCFG. All other instructions are not supported and have **unpredictable** (UNPR) results if jammed. In addition, any instruction jammed with nonzero values in IJCFG, other than those explicitly listed as supporting them, result in **unpredictable** outcomes.

Table 9-34. Implemented IJAM instructions when the processor is halted (with IJMODE=0)

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
icbtls	Instruction Cache Block Touch and Lock Set
lbz	Load Byte and Zero
ld	Load Doubleword
ldbrx	Load Doubleword Byte-Reversed Indexed
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
lvx	Load Vector Indexed
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
mftmr	Move from TMR
mfvscr	Move from VSCR

Table 9-34. Implemented IJAM instructions when the processor is halted (with IJMODE=0) (continued)

Mnemonic	Description
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
mttmr	Move to TMR
mtvscr	Move to VSCR
stb	Store Byte
std	Store Doubleword
stdbrx	Store Doubleword Byte-Reversed Indexed
stfd	Store Floating-Point Double
stfs	Store Floating-Point Single
sth	Store Halfword
sthbrx	Store Halfword Byte-Reversed Indexed
stvx	Store Vector 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. (Only the form with rA=0 is supported)
tlbwe	TLB Write Entry
wait	Wait

9.10.4.4 Instructions supported only during instruction jamming

The following table lists instructions that are only supported when the processor is halted. These instructions produce an illegal instruction exception if attempted when the processor is not halted.

Table 9-35. 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	559	Illegal instruction exception	Executed	Move from SPR, NIA
mtspr NIA		Illegal instruction exception	Executed	Move to SPR, NIA

9.10.4.5 Exception conditions and affected architectural registers

Generally, jammed instructions are allowed to modify any architecture-defined register (such as GPRs, SPRs, MSR, FPRs, and VRs) 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.

This table lists some interesting architectural registers and indicates whether or not they are affected by an exception on a jammed instruction.

Table 9-36. 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	—

Table 9-36. Effect of exceptions on machine state (continued)

Register	Affected by Interrupt	Note
MCAR	Yes	—
DEAR	No	—
MAS registers	No	—
DBSR	No	—

As Table 9-36 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 9-36, the MAS registers and DEAR are not updated by a DTLB miss. Real mode is selected by IJCFG[IJER].

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.

9.10.4.6 Instruction jamming status

The status of instruction jamming operations is captured in the DS 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 the e6500 core). No interrupt is taken, but the IVOR number associated with the interrupt that is normally 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 (ESR) in terms of bit field definitions and provides 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 processor should be halted for debug before jamming instructions. If an IJAM is performed while the processor 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].

9.10.4.7 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 processor, 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. Because 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.

9.10.4.8 Instruction jamming output

Results from instructions that have been jammed into the processor pipeline are retrieved from the IJDATA0/1 registers (and IJDATA2/3 for Altivec 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.
- Store quad-word instructions (Altivec) store the first [upper] word (bits 0–31) into IJDATA0, the second word (bits 32–63) into IJDATA1, the third word (bits 64–95) into IJDATA2, and the fourth [lower] word (bits 96–127) into IJDATA3.

The debugger can then perform register accesses to retrieve the data—it must access all four IJDATA0/1/2/3 registers in the 128-bit data case, IJDATA0/1 registers in the 64-bit data case, and only needs to access one of the data registers (IJDATA1) in the 32-bit case. It is expected that the development tool knows how much result data to expect from an instruction.

9.10.4.9 IJAM procedure

This section provides a summary of the steps to perform various instruction jamming operations.

9.10.4.9.1 IJAM of instructions with input data

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, half-word, or byte.
   — Write IJDATA2 and 3, as well, when storing to Altivec registers.
4. Write IJCFG 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 (half-word) writes, data should always be written to IJDATA1 right-justified (least significant) independent of the specific address accessed.

9.10.4.9.2 IJAM of instructions with output data

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 IJCFG 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, half-word, or byte.
   — Continue reading IJDATA2 and 3 when reading AltiVec registers.

NOTE

For 8-bit (byte) and 16-bit (half-word) reads, data is always read from IJDATA1 right-justified (least significant) independent of the specific address accessed.

9.10.4.9.3 IJAM of instructions with no associated data

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.

9.10.4.9.4 IJAM of instructions to read or write SPRs, PMRs, CR, FPSCR, and MSR

To read or write architecture-defined registers, such as SPRs, PMRs, CR, FPSCR, and MSR, use the following procedures:

Writes:

1. Follow the procedure outlined in Section 9.10.4.9.1, "IJAM of instructions with input data," to copy the data from the IJDATA registers into a GPR.
   - The write data goes into IJDATA0/1.
   - The appropriate load instruction (**ld**, **lwz**, **lhz**, **lbz**) goes into IJIR.
2. Follow the procedure outlined in Section 9.10.4.9.3, "IJAM of instructions with no associated data," to copy the data from the GPR register into the destination SPR, PMR, CR, FPSCR, or MSR.
   - The appropriate "move to" instruction (**mtspr**, **mtpmr**, **mtcrf**, **mtfsf**, **mtmsr**) goes into IJIR.

Reads:

1. Follow the procedure outlined in Section 9.10.4.9.3, "IJAM of instructions with no associated data," to copy the data from a source SPR, PMR, CR, FPSCR, or MSR register to a GPR register.
   - The appropriate "move from" instruction (**mfspr**, **mfpmr**, **mfcr**, **mffs**, **mfmsr**) goes into IJIR.
2. Follow the procedure outlined in Section 9.10.4.9.2, "IJAM of instructions with output data," to copy the data from the GPR into the IJDATA registers.
   - The appropriate store instruction (**std**, **stw**) goes into IJIR.
   - The read data can be read from IJDATA0/1.

9.10.4.10 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 produce 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

9.11 Nexus trace

This specification defines the auxiliary port functions, transfer protocols, and standard development features of the processor Nexus module in compliance with IEEE-ISTO 5001. The development features supported are:

- Program Trace
- Data Trace
- Data Acquisition messaging
- Watchpoint messaging
- Performance Profile messaging
- Timestamp Correlation messaging

- Ownership Trace

The e6500 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.

9.11.1 Nexus features

The e6500 Nexus module is compliant with the IEEE-ISTO 5001 standard. The following features are implemented:

- Program Trace via Branch Trace messaging (BTM). BTM displays program flow discontinuities (direct and indirect branches, exceptions, and so on), allowing the development tool to interpolate what transpires between the discontinuities. Thus, static code may be traced.
- Data Trace via Data Write messaging (DWM). DWM provides the capability for the development tool to trace writes to (selected) internal memory-mapped resources.
- 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 OTM is transmitted when a new process/task is activated, allowing the development tool to trace ownership flow.
- 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
- Data Acquisition messaging (DQM) allows code to be instrumented to export customized information to the Nexus Auxiliary Output Port.
- Performance counter trace via Performance Profile messages
- Timestamp correlation via Timestamp Correlation messages
- Watchpoint Trigger enable of Program Trace messaging
- Filtering of Program Trace messaging based on:
  — Process (indicated by MSR[PMM])
  — Privilege (indicated by MSR[PR])
  — Guest state (indicated by MSR[GS])
- 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
- Registers for Program Trace, Data Trace, Ownership Trace, Data Acquisition, Watchpoint messaging, and Watchpoint Trigger
- All features controllable and configurable via a memory-mapped interface, which is accessible by development tools
- All features controllable and configurable via SPRs, which is accessible by embedded software
- Timestamp capability on all message types

9.11.2 Enabling Nexus operations on the processor

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.

Once the processor Nexus clocks are active, the various features of the Nexus module can be enabled by programming the Nexus registers via the service access.

If the Nexus module is disabled, no trace output is provided, and the Nexus registers are not accessible.

9.11.3 Modes of operation

Nexus modes are described as follows:

- Reset

  The processor 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 processor 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 access.

9.11.4 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 the following table.

Table 9-37. 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 (See Table 9-56 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 (See Table 9-40.)
	8[2]	8[2]	ECODE	fixed	Error code (See Table 9-39.)
	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 e6500 implementations, this field is set to “0”.
	1	64	PC	variable	Full current instruction address[3]
	0	28	TSTAMP	variable	Timestamp (optional)

Table 9-37. 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 (See Table 9-38.)
	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	32	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 (See Table 9-41.)
	1	30	RDATA	variable	Resource data (See Table 9-41.)
	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 (See Table 9-42.)
	1	8	I-CNT	variable	Number of sequential instructions completed since the last predicate instruction, transmitted instruction count, or taken change of flow
	1	64	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)
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 (See Table 9-42.)
	1	8	I-CNT	variable	Number of sequential instructions completed since the last predicate instruction, transmitted instruction count, or taken change of flow.
	1	64	F-ADDR	variable	Full indirect change of flow target address
	1	30	HIST	variable	Direct branch / predicate instruction history information
	0	28	TSTAMP	variable	Timestamp (optional)

Table 9-37. Supported TCODEs (continued)

Message Name	Field Size (bits)[1]		Field Name	Field Type	Field Description
	Minimum	Maximum			
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 (See Table 9-43.)
	1	8	I-CNT	variable	Number os 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)
Performance Profile Message (In-Circuit Trace Msg Format)	6	6	TCODE	fixed	TCODE number = 35 (uses In-Circuit Trace msg format)
	6	6	SRC	fixed	Source processor identifier
	3	3	CKSRC	fixed	Used as an index into the list of possible data to be transmitted. Indicates the first item in the list included in CKDATA (See Table 9-45.)
	2	2	SYNC	fixed	Indicates the reason for the sync condition (See Table 9-47.)
	2	2	CKDF	fixed	Indicates the number of items included in the message (i.e. number of CKDATA fields). See Table 9-46 for the encodings used.
	1	32	CKDATA1	variable	Contains the data being transmitted in uncompressed format. The list of items delivered in CKDATA fields is indicated by the CKSRC table (See Table 9-45.).
	0	32	CKDATA2	variable	Contains additional CKDATA fields when CKDF > 1. The list of items delivered in CKDATA fields is indicated by the CKSRC table. (See Table 9-45.)
	0	28	TSTAMP	variable	Timestamp (optional)
Timestamp Correlation Message	6	6	TCODE	fixed	TCODE number = 56
	6	6	SRC	fixed	Source processor identifier
	4	4	TCORR	fixed	Indicates the timestamp correlation value. Use to correlate timestamps from multiple clients
	6	6	T-TYPE	fixed	Indicates the type of timestamp correlation request (See Table 9-44.)
	1	28	TSTAMP	variable	Timestamp (NOT OPTIONAL)

[1] The number shown in this column indicates the minimum logical number of bits required in the field after any applicable compression is 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: e6500 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 are micro-architected (implementation-specific) amounts 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) are based from this messaged PC value maintaining a coherent trace flow.

Table 9-38. Data Trace Size (DSZ) encodings (TCODE = 13)

DSZ Encoding	Transfer Size	Description
0000	0 bytes[1]	0-bit
0001	1 byte	8-bit
0010	2 bytes	16-bit/halfword
0011	3 bytes	24-bit/string
0100	4 bytes	32-bit/word
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. See Section 9.11.15.3, "Data Trace Size (DSZ) field."

Table 9-39. 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	Performance Profile message (In-Circuit Trace). Applies only to Error Type 0 (ETYPE = 0000).
1xxxxxxx	Reserved

1 Note: e6500 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 9-40. Error Type (ETYPE) encodings (TCODE = 8)

Error Type	Description
0000	Message queue overrun causes one or more messages to be lost.
0001	Contention with higher priority messages causes one or more messages to be lost.
0010–1111	Reserved

Table 9-41. Resource Code (RCODE) encodings (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
0010–0111	Reserved	N/A
1000	Timestamp counter	Maximum timestamp count (0xFF_FFFF)
1001–1111	Reserved	N/A

[1] The e6500 can complete up to two (2) instructions per cycle. The RDATA value transmitted with the value of 0xFD or 0xFE to accurately indicate the maximum instruction count when the RFM is transmitted.

Table 9-42. Branch Type (B-TYPE) encodings (TCODE = 28, 29)

Branch Type Code	Description
00	Branch instruction
01	Interrupt
1*x*	Reserved

Table 9-43. Event Code (EVCODE) encodings (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], MSR[PR], or MSR[GS]. This event applies when DC4 filter settings have been configured.
1010	EVCODE #11	Branch and link occurrence (direct branch function call)
1011–1111	—	Reserved

Table 9-44. Timestamp Correlation Type (T-TYPE) encodings (TCODE = 56)

TCORR Type Code	Description
000000	Periodic
000001	Halt mode exit
000010	Low power mode exit
000011	External Event
000100	Reserved
000101	Reserved
000110	Reserved
000111	Reserved
001000	CLK1
001001	CLK1 / 2
001010	CLK1 / 4
001011	CLK2
001100	CLK2 / 2
001101	CLK2 / 4
001110	CLK3
001111	CLK3 / 2
010000	CLK3 / 4
010001-111110	Reserved
111111	Unknown

Table 9-45. CKSRC encodings (TCODE = 35)

CKSRC	Description
000	Program Counter Capture register (CKDATA1 = PCC[32:63] and CKDATA2 = PCC[0:31])
001	Performance Monitor Counter 0 Capture register (CKDATA1 = PMCC0[32:63]) Performance Monitor Counter 1 Capture register (CKDATA2 = PMCC1[32:63])
010	Performance Monitor Counter 2 Capture register (CKDATA1 = PMCC2[32:63]) Performance Monitor Counter 3 Capture register (CKDATA2 = PMCC3[32:63])
011	Performance Monitor Counter 4 Capture register (CKDATA1 = PMCC4[32:63]) Performance Monitor Counter 5 Capture register (CKDATA2 = PMCC5[32:63])
100	Reserved
101	Reserved
110	Reserved
111	Reserved

Table 9-46. CKDF encodings (TCODE = 35)

CKDF	Description
00	1 CKDATA field included
01	2 CKDATA fields included
10	3 CKDATA fields included
11	4 CKDATA fields included

Table 9-47. SYNC encodings (TCODE = 35)

SYNC	Description
00	Previous message successful
01	Previous message not sent due to message contention
10	Previous snapshot not completed (due to another snapshot occurrence)
11	Reserved

9.11.5 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.

9.11.5.1 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.

9.11.5.2 Source ID field (SRC)

Each Nexus module in a device is identified by a unique client source identification number. The processor implements a 6-bit fixed length source ID consisting of fields to indicate that it is a core client, a unique cluster identifier, a unique core identifier, and a thread identifier.

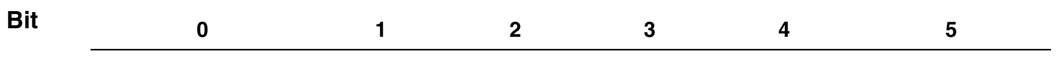


Figure 9-27. Source ID field structure

Subfield Usage	Thread	Core	Cluster		Core/SoC Client Indicator
Thread Client Specific	0 = Thread 0 1 = Thread 1 (Driven by the client to indicate thread)	00=Core 0 01=Core 1 10=Core 2 11=Core 3 (Driven by the client to indicate core)	corex_ext_src_id[0]	corex_ext_src_id[1]	Always 0 (Driven by the client to indicate a core client)

Figure 9-27. Source ID field structure (continued)

9.11.5.3 Relative Address field (U-ADDR)

The non-sync forms of the Program Trace messages include addresses that are relative to the address that was transmitted in the previous 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.

9.11.5.4 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.

9.11.5.5 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.

MSB	LSB
Correction count (4 bits)	Time stamp count (24 bits)

Figure 9-28. Timestamp 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 Resource Full messages caused by a timestamp do not pend. Clearing DC1[TSEN] disables the timestamp counter, preventing Resource Full messages from being generated due to timestamp overflow.

NOTE

The timestamp counter stops counting (freezes) when the processor enters debug halt mode or when the core enters a low power mode where the core clock is disabled. The timestamp counter resumes counting when debug halt mode is exited or when the clocks are turned back on as the low power mode is exited.

9.11.6 Nexus message queues

The e6500 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. See Section 9.11.7, “Nexus message priority,” for more information on message priorities.

The Overrun Control (OVCR) register 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 processor behaviors, which, in general, are not detailed here. An overrun can also occur if the Extended EA Nexus buffer is full. The Extended EA Nexus buffer contains four entries used to store the upper 32 bits of the effective address when a Program Trace message causes a change in the upper 32 bits (that is, a 4 GB page change). The entries remain allocated until the associated Program Trace message is sent. If all four entries are allocated and a new Program Trace message needs to allocate an entry due to a 4 GB page change, an overrun occurs.

9.11.6.1 Message queue overrun

In this mode, the message queue stops accepting messages when an overrun condition is detected. The contents of the queues is allowed to drain until empty. Incoming messages are discarded until the queue is emptied. At that point, an overrun error message is enqueued, which contains information about the types of messages that were discarded due to the overrun condition.

9.11.6.2 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 to drop one threshold level below the level that 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. See Section 9.5.8, “Nexus Overrun Control (OVCR) register,” for programming options.

9.11.6.3 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 avoid an overrun condition. If an overrun condition occurs despite this message suppression, the queue responds according to the behavior described in Section 9.11.6.1, “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.

9.11.7 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 type that is already pending a message for retry due to contention with other 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 9-48 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 9-48 lists the various message types and their relative priority from highest to lowest. Note that Program Trace is 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 the following table.

Table 9-48. 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
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 and history buffer	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	Timestamp Correlation	Timestamp Correlation message (TCM)	Y	None
6	Performance Profile	Performance Profile message (PPM)	Y	Sync encoding in next message indicates reason for message drop (except in case of queue overrun or message contention)
7 (lowest)	Data Trace	Data Trace Write message (DTM)	Y	None

9.11.7.1 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.

9.11.7.2 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.

9.11.7.3 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 or history buffer, 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.

9.11.8 Timestamp Correlation Message priority loss response and retry

If a Timestamp Correlation message (TCM) loses arbitration due to contention with higher priority messages, the TCM pends and retries on the subsequent cycle. If a new timestamp correlation request occurs while a TCM is pending, the current TCM is discarded and a new TCM is created. This means that even a TCM generated due to a frequency change could be lost.

9.11.9 Performance Profile Message priority loss response and retry

If a Performance Profile message (PPM) loses arbitration due to contention with higher priority messages, the PPM pends and retries on the subsequent cycle.

If a new performance monitor snapshot event occurs while a PPM is pending, the current PPM is discarded and PPMs are created for the new snapshot.

9.11.10 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.

9.11.11 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, wait, halted, and stopped states constitutes a processor state change for the purpose of generating debug status messages.

9.11.12 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 that are generated if any other messages are discarded due to contention have the highest priority. 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.

9.11.13 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 e6500 core is capable of generating Resource Full messages. The information from these messages is added or concatenated with information from subsequent messages to interpret the full picture of what has transpired. For the e6500 core, Resource Full messages are generated upon overflow of any one of three resources:

- Instruction counter
- History buffer
- 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 disables 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 disables 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 that counts cycles at the e6500 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] disables 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 9-41, "Resource Code (RCODE) encodings (TCODE = 27).”

9.11.14 Program Trace

This section details the Program Trace mechanisms supported by the Nexus module included in the processor. 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 that have completed because 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.

9.11.14.1 Program Trace—enable and disable

Program Trace can be enabled in one of two ways:

- Setting the DC1[TM] Program Trace enable bit (DC1[61]).
- Programming WT1[PTS] to enable Program Trace on the occurrence of a watchpoint condition.

NOTE

For a disabled thread, Program Trace remains disabled until the thread is enabled.

Once Program Trace is enabled, Lite Program Trace mode can be turned on in one of two ways:

- Setting the DC1[LPTE] Lite Program Trace enable bit (DC1[38]).
- Programming WT2[LPTS] to enable Program Trace on the occurrence of a watchpoint condition.

Lite Program Trace mode may be turned off in one of two ways:

- Clearing the DC1[LPTE] Lite Program Trace enable bit (DC1[38]).
- Programming WT2[LPTE] to disable Lite Program Trace mode on the occurrence of a watchpoint condition.

Program Trace may be disabled by the following:

- Clearing the DC1[TM] Program Trace enable bit (DC1[61]). Note that resetting the Nexus module clears all Nexus registers, disabling Program Trace as a side effect.
- Disabling the thread effectively disables Program Trace:
  - Forces a flush of the history buffer when the thread is disabled (PCM w/EVCODE = #5).
  - Does not disable Program Trace entirely and re-starts when the thread is re-enabled.
- Programming WT1[PTE] to disable Program Trace on the occurrence of a watchpoint condition.
- Program Trace can be filtered out (effectively disabled):
  - For performance monitor mark (MSR[PMM]) by programming DC4[PTFPMM].
  - For privilege level (MSR[PR]) by programming DC4[PTFPR].
  - For guest state (MSR[GS]) by programming DC4[PTFGS].

Program Trace is effectively suppressed whenever the processor is in the debug halted or debug stopped state. Instruction jamming operations do not produce any Program Trace messages. Whenever the processor leaves the debug halted state, Program Trace enable state reverts to the status of DC1[61].

9.11.14.2 Lite Program Trace mode

To reduce the number of Indirect Branch History messages and help alleviate bandwidth issues, Lite Program Trace mode may be used. It is designed to reduce the number of IBHM's when code is well behaved by using a return stack, or Nexus Link Stack, that is maintained both in hardware and the Program Trace reconstruction tool. The e6500 core currently implements three entries in the Nexus Link Stack.

Well-behaved code in this context is defined as code that calls functions using branch and link class instructions with returns from those instructions being branch to link class instructions. Normally, a branch to link register instruction would cause an IBHM message to be generated. However, by saving off the expected return target address when the function is called, it is possible to predict that return address. If the prediction is correct, it eliminates the need for that IBHM and converts this indirect branch taken to a direct branch taken.

9.11.14.2.1 Lite Program Trace mode—enabling

Program Trace must be enabled (in DC1[TM]) in order to use Lite Program Trace mode. Once Program Trace is enabled, Lite Program Trace mode may be turned on (or off) by setting (or clearing) the Lite Program Trace Enable bit (DC1[LPTE]). Alternatively, Lite Program Trace mode can be turned on and off via watchpoint triggers configured in WT2 (LPTS/LPTE).

9.11.14.2.2 Lite Program Trace mode—how it works

When enabled, Program Trace checks the target of the taken branch to the link register branch and compares it against the top of the Nexus Link Stack. If the top entry of the link stack matches the target address of the branch to link register branch, it is considered as a taken direct branch and a history bit is added to the history buffer rather than causing an IBHM to be sent.

The Nexus Link Stack is the key to the operation of Lite Program Trace. It keeps track of return addresses when function calls are made using a branch and link instruction. Thus, when Lite Program Trace is enabled, the link stack pushes a return address for every branch and link that is taken. The Program Trace decoder software must mirror this operation when reconstructing the program.

When a branch to link register taken is seen, an entry is popped from the link stack and compared against the target of that branch. If they match, the branch to link register taken is treated as a direct branch because the target can be implied by the Nexus Link Stack tracked by the Program Trace decoder. However, if they do not match, all entries of the link stack are invalidated and the normal IBHM is sent.

This table describes the Nexus Link Stack operations.

Table 9-49. Nexus Link Stack operations

Scenario	Action
Branch and link taken	Push branch and link instruction's address plus 4 onto the Nexus Link Stack. **Note:** If the Nexus Link Stack is full, the oldest entry is overwritten. If that is the case, the overwritten entry are lost and the Nexus Link Stack optimization for that entry are not realized.
Branch to link taken	Pop Nexus Link Stack and compare this address to the target address of the branch to link taken instruction. One of the following then occurs. 1.) If there is a match and Lite Program Trace is enabled, convert this branch from a branch indirect taken to a direct branch taken. This causes this entry to be entered into the branch history buffer and no IBHM is sent out. 2.) If the entry does not match the branch to link target address, then the entire Nexus Link Stack is invalidated and the IBHM is sent out. 3.) If the entry does match, but Lite Program Trace is not enabled, the stack is not invalidated, but the IBHM is still sent out.
Program Trace Synchronization message (Hard Sync / Soft Sync)	Resets the Nexus Link Stack.

In order for Lite Program Trace to work, the Program Trace reconstruction tool must maintain its own link stack, as well as use the rules in the following table.

Table 9-50. Trace reconstructor link stack operation

Scenario Seen in Program Trace Stream	Action
Branch history buffer from decoded IBHM (or PCM) contains a branch and link instruction taken	Push branch and link instruction's address plus 4 onto the Trace Reconstructor Link Stack
Branch history buffer from decoded IBHM (or PCM) hits a branch to link instruction taken	This indicates that the Nexus Link successfully predicted the target address of the branch to link register. In Program Trace reconstruction, the tool should then use the top entry of the Trace Reconstructor Link Stack it maintains as the target address.
After decoding the branch history using the preceding rules, it is found that the IBHM is sent out due to a branch to link instruction taken.	This indicates that the Nexus Link did not successfully predict a branch to link target address. The Nexus Link Stack was invalidated and now the Trace Reconstructor Link Stack should also be invalidated.
Program Trace Synchronization message (Hard Sync) decoded	Resets the Reconstructor Link Stack

9.11.14.2.3 Lite Program Trace mode—example

Consider the following program segments:

```
main()
{
   for(i=0; i<5; i++)
       hot();
}
```

```
hot()
{
   melt();
}
```

```
melt()
{
   return;
}
```

Figure 9-29. Lite Program Trace mode example

Referring to the above code segments, each function return generates an Indirect Branch History message (IBHM) when Lite Program Trace mode is turned off:

- Program Trace messages generated with Lite Program Trace mode <u>turned off</u>

```
Program Trace Synchronization Message
Indirect Branch History Message (return from melt)
Indirect Branch History Message (return from hot)
Indirect Branch History Message (return from melt)
Indirect Branch History Message (return from hot)
Indirect Branch History Message (return from melt)
Indirect Branch History Message (return from hot)
Indirect Branch History Message (return from melt)
Indirect Branch History Message (return from hot)
Indirect Branch History Message (return from melt)
Indirect Branch History Message (return from hot)
Program Correlation Message (end of main, contains history since return of hot)
```

However, if a program is well behaved (meaning it does not alter the return address on the stack), these function returns that generate IBHMs can be predicted and logged as a direct branch in the history buffer instead.

In order to consider these function returns as direct branches, the target address must be known. In other words, as long as the return address is not altered after it is pushed on the stack, we can record the return as a direct branch. To do this, the Nexus module builds a link stack for tracking return addresses. When a function call is executed, the corresponding return address is pushed onto the Nexus Link Stack. Upon reaching a function return, the top entry of the Nexus Link Stack is popped off the stack and compared to the actual return address. If they match, instead of generating a IBHM as normal, the function return is added to the branch history buffer as a taken branch. If they do not match, the IBHM is generated and the Nexus Link Stack is invalidated.

Thus, returning to the same code segment with Lite Program Trace mode turned on:

- Program Trace messages generated with Lite Program Trace mode <u>turned on</u>

```
Program Trace Synchronization Message
Program Correlation Message (end of main, contains history of entire program)
```

9.11.14.3 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 that produce branch/predicate history information are not included in this count. Taken direct branches (BL/BCL) that generate PCMs are also 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.

9.11.14.4 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 taken indirect branches when the target address matches the address being tracked on the link stack) 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 e6500 core 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.

This table describes the branch and predicate history events.

Table 9-51. 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 the 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.
Taken indirect branches	1	**bclr**, **bclrl**	When Lite Program Trace mode is turned on, taken **bclr**, **bclrl** instructions that are well behaved (see Section 9.11.14.2, “Lite Program Trace mode”), are logged in the history buffer. When the source code is not well behaved, or when Lite Program Trace is disabled, these instructions are not logged in the history buffer but generate an IBHM instead.
Predicated instructions	1	**isel**, **fsel**	—

9.11.14.5 Indirect Branch message events

An Indirect Branch event is a change of flow in which a 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.

This table describes the Indirect Branch message events.

Table 9-52. Indirect Branch message events

Indirect Branch Message Event	BTYPE	Relevant Instructions
Taken register indirect branches	00	**bcctr, bcctrl, bclr**[1], **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

[1] When Lite Program Trace is enabled, taken **bclr** instructions that are well behaved (see Section 9.11.14.2, “Lite Program Trace mode”), are logged in the history buffer. When **bclr** instructions are not well behaved, or when Lite Program Trace is disabled, these instructions are not logged in the history buffer but generate a IBHM instead.

9.11.14.6 Resource Full events

Program Trace can produce two types of Resource Full messages (TCODE 27): instruction counter and history buffer.

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.

9.11.14.7 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 4 (DC4) register.

The e6500 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] filter configured by DC4[PTFPMM].
- Program Trace becomes masked due to MSR[PR] filter configured by DC4[PTFPR].
- Program Trace becomes masked due to MSR[GS] filter configured by DC4[PTFGS].
- Branch and Link instructions (direct branch function call, **bl/bcl**).

9.11.14.8 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 bases their sequential instruction count (I-CNT) and branch history (HIST) values starting from the program counter (PC) value transmitted within these messages.

This table outlines hard sync cases.

Table 9-53. Hard synchronization conditions

Hard SYNC Condition	Description
$\overline{\text{EVTI}}$0 Assertion	The e6500 $\overline{\text{EVTI}}$0 pin is 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. This includes when Program Trace is re-enabled due to filtering (masking) because MSR[PMM|PR|GS] filters can disable Program Trace temporarily. NOTE: The hard sync message is suppressed in the case where Program Trace is being re-enabled (via filtering) at the same time as Program Trace is being disabled (via a stop trigger).
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.

Table 9-53. Hard synchronization conditions (continued)

Hard SYNC Condition	Description
Message Contention	One or more messages is 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 9.11.7, “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 that 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).

This table outlines soft sync cases.

Table 9-54. Soft synchronization conditions

Soft SYNC Condition	Description
$\overline{\text{EVTI}}$1 Assertion	The e6500 $\overline{\text{EVTI}}$1 pin is 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 ensures 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.

9.11.15 Data Trace

The e6500 core 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 KB on exact match. (See Section 2.14.7, “Debug Control 4 (DBCR4) register,” 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.

9.11.15.1 Data Trace—enable and disable

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] and DBCR0[EDM] are don't cares. They do not have to be set for the DACs to be used to trace data.
- EDBRAC0[DAC1, DAC2] and EDBRAC0[TRACE] should be allocated to the same owner for data tracing. (**Note:** This is not absolutely necessary for Data Trace to work).
- DAC1 and DAC2 should be programmed with the desired addresses for data tracing.
- DBCR0[DAC1, DAC2] should be programmed to disable the DAC debug conditions. If DBCR0[DAC1, DAC2] are enabled, a DAC condition may prevent the store operation from completing by causing entry into debug halted state or generating a debug interrupt.
- If DBCR0[DAC1, DAC2] are programmed to disable the DAC conditions, DBCR4[DAC1CFG, DAC2CFG] should be programmed to enable the DAC 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:

- Set the appropriate DC1[TM] bit (DC1[62]).
- Program WT1[DTS] to enable Data Trace on the occurrence of a watchpoint condition.

Similarly, Data Trace may be disabled by one of the following:

- Disable the DAC conditions for store-type accesses.
- Clear the appropriate DC1[TM] bit (DC1[62]). Note that resetting the Nexus module clears all Nexus registers, disabling Data Trace as a side effect.
- Program WT1[DTE] to disable Data Trace on the occurrence of a watchpoint condition.

NOTE

The latter two mechanisms defined above disable additional stores from entering the e6500 store queue, but accesses that have already entered the queue (that is, accesses in flight) messaged out before the DTMs are actually disabled.

Data Trace is effectively suppressed whenever the processor is in the debug halted or debug stopped state. Instruction jamming operations do not produce any Data Trace messages. Whenever the processor leaves the debug halted state, Data Trace enable state reverts to the status of DC1[62].

9.11.15.2 Data Trace range control

The Data Trace address range is limited to two 4 KB ranges. These ranges are controlled by setting the effective addresses in DAC1 and DAC2 and the DAC configuration in DBCR4, as follows:

- DBCR4[DAC1CFG] and DBCR4[DAC2CFG] 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 KB.

9.11.15.3 Data Trace Size (DSZ) field

For normal data transfers, DSZ indicates the size (in bytes) of the store that is being traced, but there are two special cases that 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 always include a DSZ value indicating 64-bytes (4’b1011). See Section 3.4.11.1, “User-level cache instructions,” for more detail on cache management instructions.

The e6500 core 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) indicates a zero byte data transfer. See the integrated device reference manual for more details on the **dsn** instruction.

9.11.15.4 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:

MSB	LSB
DAC tag	ADDR[52:63]

Figure 9-30. 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[0:51] or DAC2[0:51]	ADDR[52:63]

Figure 9-31. 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 KB results in address aliasing, which makes precise reconstruction of the full effective address impossible (without other implied restrictions or information that can remove the ambiguity).

9.11.15.5 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.

9.11.15.6 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 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 9-55. Data Trace message events

Data Trace Message Event Source	Relevant Instructions
Cache management instructions that are treated as store-type data storage accesses[1]	**dcba, dcbal, dcbz, dcbzep, dcbzl, dcbzlep**
External PID store instructions that produce data storage write accesses	**stbepx**, **stfdepx**, **sthepx**, **stvepx[l]**[2], **stwepx**, **stdepx**
Integer store instructions that produce data storage write accesses	**stb**[**u**][**x**], **std**[**u**][**x**], **sth**[**u**][**x**], **stw**[**u**][**x**], **stdbrx**, **sthbrx**, **stwbrx**, **stmw**[2]
Floating-point store instructions that produce data storage write accesses	**stfiwx**, **stfd**[**u**][**x**], **stfdepx**, **stfs**[**u**][**x**]
Altivec store instructions that produce data storage write accesses	**stvebx**, **stve[x]hx**, **stve[x]wx**, **stve[x]bx**, **stvflx[l]**[2], **stvfrx[l]**[2], **stvswx[l]**[2], **stvx[l]**[2]
Decorated store instructions that produce data storage write accesses	**stbdx**, **sthdx**, **stwdx**, **stddx**, **stfddx**, **dsn**
Conditional store instructions[3]	**stwcx.**, **stdcx.**

1 These instructions are treated like data storage writes with write data value of zero.
2 A separate Data Trace message is generated for each word stored that meets the trace criteria.
3 These instructions only generate Data Trace messages if the associated store is successful (that is, the condition evaluates to true).

9.11.16 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 operating system process ID changes that are reflected in the Nexus Process ID (NPIDR) register or the PID register generate Ownership Trace messages (OTMs). Changes in the Logical Partition ID (LPIDR) register also generate OTMs.

9.11.16.1 Ownership Trace—enable and disable

Ownership Trace can be enabled by setting the appropriate DC1[TM] bit (DC1[63]).

Similarly, Ownership Trace may be disabled by one of the following:

- Clearing the appropriate DC1[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 is in the debug halted or debug stopped state. Instruction jamming operations do not produce any OTMs. Whenever the processor leaves the debug halted state, Ownership Trace enable state reverts to the status of DC1[63].

9.11.16.2 Ownership Trace Process field

The process field of an Ownership Trace message (OTM) 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]. The process field also consists of an index to identify which process ID values are being reported for a particular message. See Section 9.11.16.3, “Standard Ownership Trace message events,” and Section 9.11.16.4, ““Sync” Ownership Trace message events,” for detail on Ownership Trace events.

Table 9-56. OTM Process field components

Configuration DC1[OTS]	Process Field				Total PROCESS Field Width
	PID Value		PID Index		
	Description	Size	Encoding	Description	
0	PID0[50:63]	14-bits	0000	OS PID	18-bits
1	NPIDR[32:63]	32-bits			36-bits
x	LPIDR[58:63] (Logical Partition ID)	6-bits	0001	Hypervisor PID	10-bits
0	{LPIDR[58:63], MSR[GS], PID0[50:63], MSR[PR]}	22-bits	0010	“Sync” PID	26-bits
1	{LPIDR[58:63], MSR[GS], NPIDR[32:63], MSR[PR]}	40-bits			44-bits
x	N/A	0-bits	0011–1111	Reserved	0-bits

9.11.16.3 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 processID 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.

9.11.16.4 “Sync” Ownership Trace message events

The following events generate “sync” OTMs when Ownership Trace is enabled:

- Upon OTM enable (DC1[TM]), an OTM is generated with the same information as in the “sync” OTM case (PID index = 0010).
- 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 is generated with the same information as in the “sync” OTM case (PID index = 0010).
- Upon a change in instruction address space (MSR[IS]), an OTM is generated with the same information as in the “sync” OTM case (PID index = 0010).
- Periodically—once every 256 messages—an OTM is also generated with the same information as in the “sync” OTM case (PID index = 0010). These periodic Ownership Trace message events can be disabled by writing DC1[POTD] = 1.
- After flush of the Nexus buffers due to a FIFO Overrun Error, an OTM “sync” message is generated. If ownership changes 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.

9.11.17 Data Acquisition Trace

This section details the data acquisition mechanisms supported by the Nexus module included in a processor. 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.

9.11.17.1 Data Acquisition Trace—enable and disable

Enabling and disabling Data Acquisition Trace messaging is done as follows:

- Enable by setting the appropriate DC1[TM] bit (DC1[58]).
- Disable by clearing the appropriate DC1[TM] bit (DC1[58]).

Note that resetting the Nexus module clears all Nexus registers, which disables Data Acquisition Trace as a side effect.

Data Acquisition Trace is effectively suppressed whenever the processor is in the debug halted or debug stopped state. Instruction jamming operations do not produce any Data Acquisition Trace messages. Whenever the processor leaves the debug halted state, the Data Acquisition Trace enable state reverts to the status of DC1[58].

9.11.17.2 Data Acquisition ID Tag field

The Data Acquisition ID Tag field (IDTAG) is an 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 usage of the IDTAG is left to the discretion of the development tool to be used in whatever manner is deemed appropriate for the application.

9.11.17.3 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].

9.11.17.4 Data Acquisition Trace event

For DQM, a dedicated SPR is 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. See Section 9.9.2, “Internal and external debug modes,” for more information regarding internal/external debugger contention of debug resources. Reads from the data acquisition channel do not generate a data acquisition event and return zeros for the read data.

9.11.18 Watchpoint Trace

This section details the Watchpoint Trace mechanisms supported by the Nexus module included in the processor. Watchpoint Trace is implemented using Watchpoint Trace messaging in accordance with IEEE-ISTO 5001 definitions.

Watchpoint Trace facilitates monitoring program execution for specific event occurrences.

9.11.18.1 Watchpoint events

Table 9-57 lists all of the watchpoint events supported by the e6500 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 9-57. Processor debug watchpoint mappings

Processor Watchpoints	Type	Event Description
Watchpoint #1	IAC1	Instruction Address Compare 1 debug event watchpoint—Asserted whenever an IAC1 compare occurs, regardless of being enabled to set the DBSR status.
Watchpoint #2	IAC2	Instruction Address Compare 2 debug event watchpoint—Asserted whenever an IAC2 compare occurs, regardless of being enabled to set the DBSR status.
Watchpoint #3	IAC3	Instruction Address Compare 3 debug event watchpoint—Asserted whenever an IAC3 compare occurs, regardless of being enabled to set the DBSR status.
Watchpoint #4	IAC4	Instruction Address Compare 4 debug event watchpoint—Asserted whenever an IAC4 compare occurs, regardless of being enabled to set the DBSR status.
Watchpoint #5	DAC1	Data Address Compare 1 debug event watchpoint—Asserted whenever an DAC1 compare occurs, regardless of being enabled to set the DBSR status.
Watchpoint #6	DAC2	Data Address Compare 2 debug event watchpoint—Asserted whenever an DAC2 compare occurs, regardless of being enabled to set the DBSR status.
Watchpoint #7	DEVNT4	Data Acquisition Event 4 (DEVNT4)
Watchpoint #8	DEVNT5	Data Acquisition Event 5 (DEVNT5)
Watchpoint #9	IAC5	Instruction Address Compare 5 debug event watchpoint—Asserted whenever an IAC5 compare occurs, regardless of being enabled to set the DBSR status.
Watchpoint #10	IAC6	Instruction Address Compare 6 debug event watchpoint—Asserted whenever an IAC6 compare occurs, regardless of being enabled to set the DBSR status.
Watchpoint #11	EVTI0[1]	Event In 0 (EVTI0)
Watchpoint #12	EVTI1	Event In 1 (EVTI1)
Watchpoint #13	DEVNT0	Data Acquisition Event 0 (DEVNT0)
Watchpoint #14	DEVNT1	Data Acquisition Event 1 (DEVNT1)
Watchpoint #15	IAC7	Instruction Address Compare 7 debug event watchpoint
Watchpoint #16	IAC8	Instruction Address Compare 8 debug event watchpoint
Watchpoint #17	PMW0[2]	Performance Monitor Watchpoint 0 (PMW0)
Watchpoint #18	PMW1[2]	Performance Monitor Watchpoint 1 (PMW1)
Watchpoint #19	PMW2[2]	Performance Monitor Watchpoint 2 (PMW2)
Watchpoint #20	PMW3[2]	Performance Monitor Watchpoint 3 (PMW3)
Watchpoint #21	DEVNT2	Data Acquisition Event 2 (DEVNT2)
Watchpoint #22	DEVNT3	Data Acquisition Event 3 (DEVNT3)
Watchpoint #23	PMW4[2]	Performance Monitor Watchpoint 4 (PMW4)
Watchpoint #24	PMW5[2]	Performance Monitor Watchpoint 5 (PMW5)
Watchpoint #25	PMEVENT	Performance Monitor Event
Watchpoint #26	PID	Process ID Update (PID) (Generated on updates to both PID and LPIDR registers)

Table 9-57. Processor debug watchpoint mappings (continued)

Processor Watchpoints	Type	Event Description
Watchpoint #27	RELOAD CNTR	Reloadable Counter Event
Watchpoint #28	IRPT	Interrupt Taken debug event watchpoint
Watchpoint #29	RET	Interrupt Return debug event watchpoint
Watchpoint #30	DNI	Debug Notify Interrupt instruction debug event—Generated when the **dni** instruction executes and bit 15 of the instruction is set.
Watchpoint #31	TRAP	TRAP instruction debug event—Generated for both debug (IVOR15) and program (IVOR6) TRAP exceptions.
Watchpoint #32	DNH	Debug Notify Halt instruction debug event—Generated when the **dnh** instruction executes and bit 15 of the instruction is set.

[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 PMLCbs. See Section 2.16.3, "Local control b registers (PMLCb0–PMLCb5/UPMLCb0–UPMLCb5)."

When the debug resource is allocated to the external debugger (EDM), IACs, DACs, return from interrupt, and return from critical interrupt, debug conditions cause bits to be set in EDBSR0 (if not masked within EDBSRMSK0) and the processor to halt instead of taking a debug interrupt. In this case, the watchpoint for these respective event triggers on the update to EDBSR0.

9.11.18.2 Watchpoint Trace–enable and disable

Watchpoint Trace messaging can be enabled by setting the appropriate DC1[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 set up the individual watchpoint conditions. These additional configuration controls are specific to the event type.

Similarly, Watchpoint Trace may be disabled by the following:

- Clear the appropriate DC1[TM] bit (DC1[60]). Note that resetting the Nexus module clears all Nexus registers, which disables Program Trace as a side effect.
- Clear the WMSK register such that no watchpoint events are enabled to produce a Watchpoint Trace message.

Watchpoint Trace is effectively suppressed whenever the processor is in the debug halted or debug stopped state. Instruction jamming operations do not produce any Watchpoint Trace messages. Whenever the processor leaves the debug halted state, the Watchpoint Trace enable state reverts to the status of DC1[60].

9.11.18.3 Watchpoint Hit field

The Watchpoint Hit field consists of 32 bits with one bit per watchpoint event. Whenever a Watchpoint Trace message is generated, the Watchpoint Hit field of the message includes a one (1) for each watchpoint event that occurred at that time and a zero (0) 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

WP32	WP31	WP30	WP29	WP28	WP27	WP26	WP25	WP24	WP23	WP22	WP21	WP20	WP19	WP18	WP17	WP16	WP15	WP14	WP13	WP12	WP11	WP10	WP9	WP8	WP7	WP6	WP5	WP4	WP3	WP2	WP1

Figure 9-32. Watchpoint Hit field

9.11.18.4 Watchpoint Trace message events

A Watchpoint Trace message is generated whenever Watchpoint Trace is enabled (DC1[60] = 1) and a watchpoint event that is enabled to produce a Watchpoint Trace message occurs (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.

9.11.19 Timestamp Correlation messages

Timestamp Correlation messages are used by an external trace tool to correlate timestamps from multiple Nexus clients. Timestamp Correlation messages are enabled whenever any Nexus trace modes are enabled and timestamps are enabled (both DC1[TM] and DC1[TSEN] are non-zero). A timestamp correlation message is generated in response to a system-level request to correlate timestamps. The request may be periodic or may be due to an event where a client’s timestamp clock was briefly halted and needs to be re-correlated.

9.11.20 Performance Profile messages

Performance Profile messages are used to transmit snapshots of the NIA and performance monitor counters (PMC0-5) to an external tool for performance profiling analysis.

Performance Profile messages use the In-Circuit Trace (ICT) TCODE and format.

9.11.20.1 Performance Profile messages—enable and disable

Performance Profile messages can be enabled in one of two ways:

- Set the DC1[TM] profile message enable bit (DC1[59]).
- Program WT2[ITS] to enable profile messages on the occurrence of a watchpoint condition.

Performance Profile messages can be disabled in one of two ways:

- Clear the DC1[TM] profile message enable bit (DC1[59]). Note that resetting the Nexus module clears all Nexus registers, disabling Profile messages as a side effect.
- Program WT2[ITE] to disable profile messages on the occurrence of a watchpoint condition.

9.11.20.2 Performance Profile message events

When enabled, Performance Profile messages are generated when a snapshot event occurs. Snapshot events are configured within the Reloadable Counter Configuration (RCCR) register.

9.11.20.3 Performance Profile message configuration

The profiling message uses the In-Circuit Trace (TCODE 35) format, which allows for flexible messaging.

The Performance Profile Configuration bits (DC1[PPC]) configure the data transmitted at each snapshot event. The data is transmitted using the order defined by the PPC bits. If a bit is unselected, the corresponding data is not transmitted but is skipped. Each bit selected in PPC corresponds to a separate message that is generated. In other words, a single snapshot event can result in multiple messages being created.

The timestamp value is captured when a snapshot event occurs and is included in the first message sent. No timestamp is transmitted on additional messages generated by a snapshot event.

Example 1: If DC1, PPC = 4’b1111, the following messages are sent:

```
TCODE=35,SRC,CKSRC=4’b0000,SYNC=2’b00,CKDF=2’b01,CKDATA1=PCC[0:29],CKDATA2=PCC[30:61],TSTAMP
TCODE=35,SRC,CKSRC=4’b0001,SYNC=2’b00,CKDF=2’b01,CKDATA1=PMCC0,CKDATA2=PMCC1
TCODE=35,SRC,CKSRC=4’b0011,SYNC=2’b00,CKDF=2’b01,CKDATA1=PMCC2,CKDATA2=PMCC3
TCODE=35,SRC,CKSRC=4’b0101,SYNC=2’b00,CKDF=2’b01,CKDATA1=PMCC4,CKDATA2=PMCC5
```

Example 2: If DC1, PPC = 4’b0101, the following messages are sent:

```
TCODE=35,SRC,CKSRC=4’b0001,SYNC=2’b00,CKDF=2’b01,CKDATA1=PMCC0,CKDATA2=PMCC1,TSTAMP
TCODE=35,SRC,CKSRC=4’b0101,SYNC=2’b00,CKDF=2’b01,CKDATA1=PMCC4,CKDATA2=PMCC5
```

9.11.20.4 Performance Profile Sync field

Performance Profile messages do not use compression on the CKDATA fields. However, the Sync field is used within the Performance Profile message to indicate the success or failure of previous message submission to the FIFO, and also when another snapshot is taken before the current snapshot was completed. See Table 9-47 for Sync field encodings.

9.12 Performance monitor

This section describes the performance monitor, which is defined by the architecture and described in *EREF*. The primary function of the performance monitor is to count events pertaining to the performance of the processor (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.

9.12.1 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], can be used to turn the counters on/off for marked processes.
- Privilege level filtering can be applied so the counters only increment during the privilege level of interest.
- The move to/from performance monitor (PMR) register instructions, **mtpmr** and **mfpmr**, can be used to access performance monitor configuration and counter registers.
- The external input, *pm_event* can be used to allow events external to the e6500 core to be counted.
- The watchpoints can be used as a trigger for counter snapshots.
- The reloadable counter can be used to generate snapshots periodically.
- The performance monitor counter values can be automatically sent externally via Nexus messages at each snapshot.

This figure shows a detailed view of one of the PMC counters available within the processor performance monitor. The sections in blue are special triggering controls that are available for the e6500 core.

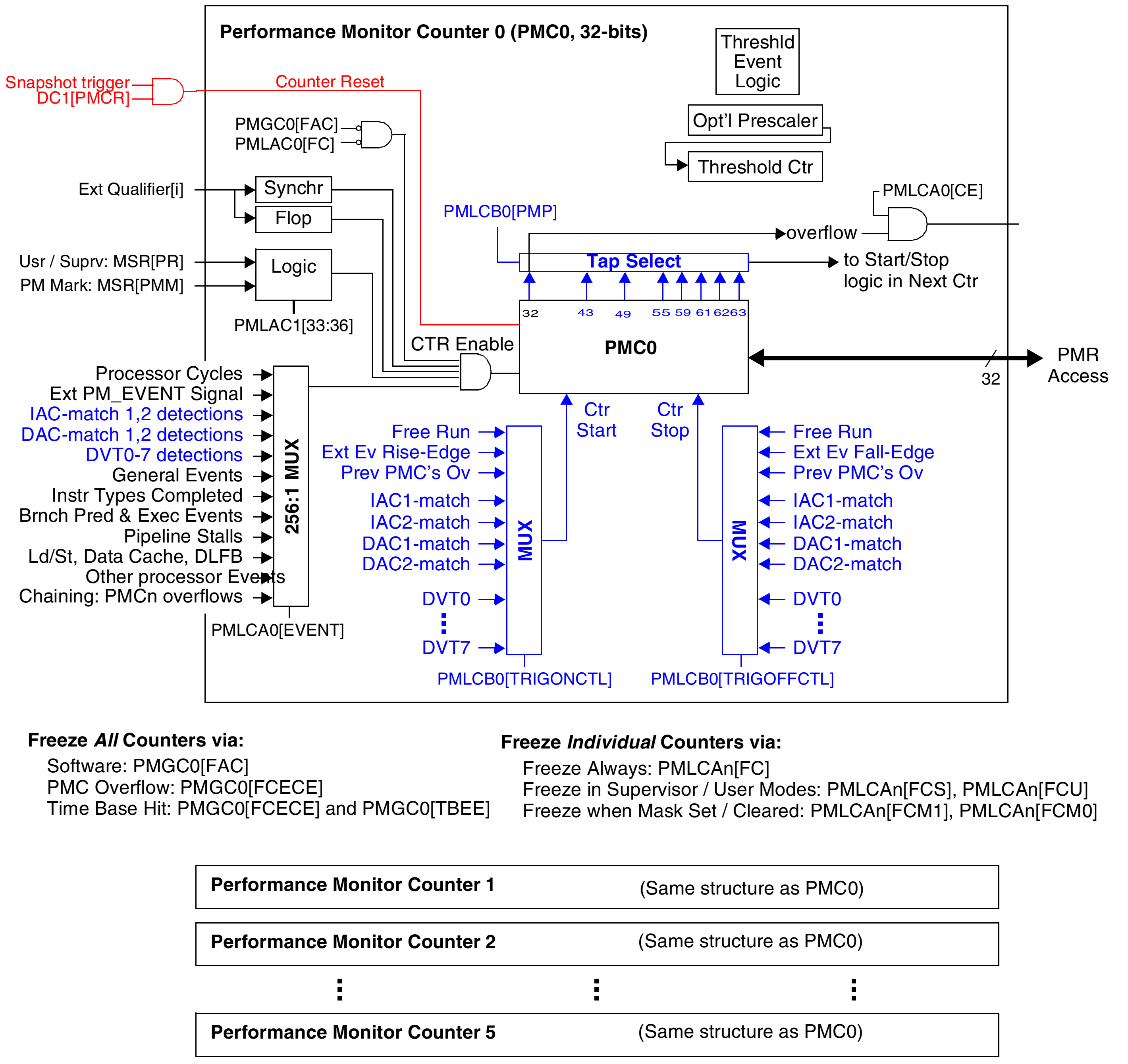


Figure 9-33. Detailed view: processor performance monitor counters 0 through 3

- PMRs:
  — The performance monitor counter registers (PMC0–PMC5) are 32-bit counters used to count software-selectable events. Each counter counts up to 512 events. UPMC0–UPMC5 provide

user-level read access to these registers. Reference events are those that should be applicable to most microprocessor microarchitectures and be of general value. They are identified in Table 9-61.

— The Performance Monitor Global Control (PMGC0) register 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–PMLCa5 and PMLCb0–PMLCb5) control each individual performance monitor counter. Each counter has a corresponding PMLCa and PMLCb register. UPMLCa0–UPMLCa5 and UPMLCb0–UPMLCb5 provide user-level read access to PMLCa0–PMLCa5, PMLCb0–PMLCb5).

The performance monitor interrupt follows the architecture-defined interrupt model and is briefly described in Section 4.9.19, "Performance monitor interrupt—IVOR35/GIVOR35."

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.

Performance monitor activity is suspended in PH15, PH20, PH30, and PW20 Power Management activity states. See Section 8.3, "Core power management states," for more details on Power Management activity states. The performance monitor interrupt is not an exit condition from the PW20 Power Management activity state.

9.12.2 Performance monitor instructions

Instructions for reading and writing the PMRs are shown in the following table. These are described in detail in *EREF*.

Table 9-58. Performance monitor instructions

Name	Mnemonic	Syntax
Move from performance monitor register	**mfpmr**	**r**D,PMRN
Move to performance monitor register	**mtpmr**	PMRN,**r**S

9.12.3 Performance monitor interrupt

The performance monitor interrupt is triggered by an enabled condition or event. The only enabled condition or event defined for the e6500 core 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] = 1.
  — The counter indicates an overflow; PMC*n*[OV] = 1.

If PMGC0[PMIE] = 1, an enabled condition or event triggers the signaling of a performance monitor exception. If PMGC0[FCECE] = 1, an enabled condition or event also triggers all performance monitor counters to freeze.

Even if the performance monitor exception condition occurs, the performance monitor interrupt does not occur unless the interrupt is enabled. The performance monitor interrupt is enabled if one of the following is true:

- The interrupt is directed to the guest state, base class interrupts are enabled, and the processor is in the guest state (EPCR[PMGS] = 1, MSR[EE] = 1, and MSR[GS] = 1).
- The interrupt is directed to the hypervisor state, base class interrupts are enabled, or the processor is in the guest state ((EPCR[PMGS] = 0) and (MSR[EE] = 1 | MSR[GS] = 1)).

Although the performance monitor exception condition could occur when the interrupt is not enabled, the interrupt cannot be taken until the enabling conditions are met. If PMC*n* overflows, signals an exception (PMLCa*n*[CE] and PMGC0[PMIE] = 1) while the interrupt is not enabled, and freezing of the counters is not enabled (PMGC0[FCECE] = 0), PMC*n* can wrap around to all zeros again without the performance monitor interrupt being taken.

9.12.4 Event counting

This section describes configurability and specific unconditional counting modes.

9.12.4.1 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*.

This table describes the processor states and the settings of the FCS, FCU, FCM1, FCM0, FCGS0, and FCGS1 fields in PMLCa*n* necessary to enable monitoring of each processor state.

Table 9-59. Processor states and PMLCa*n* bit settings

FCS	FCU	FCM1	FCM0	FCGS0	FCGS1	Processor State
0	0	0	1	0	0	Marked
0	0	1	0	0	0	Not marked
0	1	0	0	0	0	Supervisor
1	0	0	0	0	0	User
0	1	0	0	1	0	Guest Supervisor
0	1	0	0	0	1	Hypervisor
0	1	0	1	0	0	Marked and supervisor

Table 9-59. Processor states and PMLCa*n* bit settings (continued)

FCS	FCU	FCM1	FCM0	FCGS0	FCGS1	Processor State
1	0	0	1	0	0	Marked and user
0	1	1	0	0	0	Not marked and supervisor
1	0	1	0	0	0	Not mark and user
0	0	0	0	0	0	All
X	X	1	1	1	1	None
1	1	X	X	X	X	None

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.

9.12.4.2 Processor performance monitor and program counter capture function

For real-time debug, a capture function is available for the processor performance counters, PMC0–PMC5, as well as the micro-architected program counter (PC). Whenever the snapshot trigger signal is asserted, the PMC values of each counter and the current PC are captured in registers. The capture registers are readable through the e6500 memory-mapped interface. They can also be used to generate Nexus performance profile messages automatically at each snapshot.

The snapshot trigger is configured in the Performance Monitor Snapshot Configuration register. Any combination of all 32 watchpoints can be used to generate the snapshot trigger. See Section 9.5.11, "Performance Monitor Snapshot Configuration (PMSCR) register," for details on how to configure the snapshot trigger. The following are some examples of e6500 watchpoint sources:

- Reloadable counter
- 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 e6500 core.

A high-level block diagram of the capture functionality is shown in Figure 9-34. The capture registers are written only from the PMCs on the snapshot trigger signal and are readable only through the e6500 memory-mapped interface. The location of these registers in the memory map are outlined in Table 9-31.

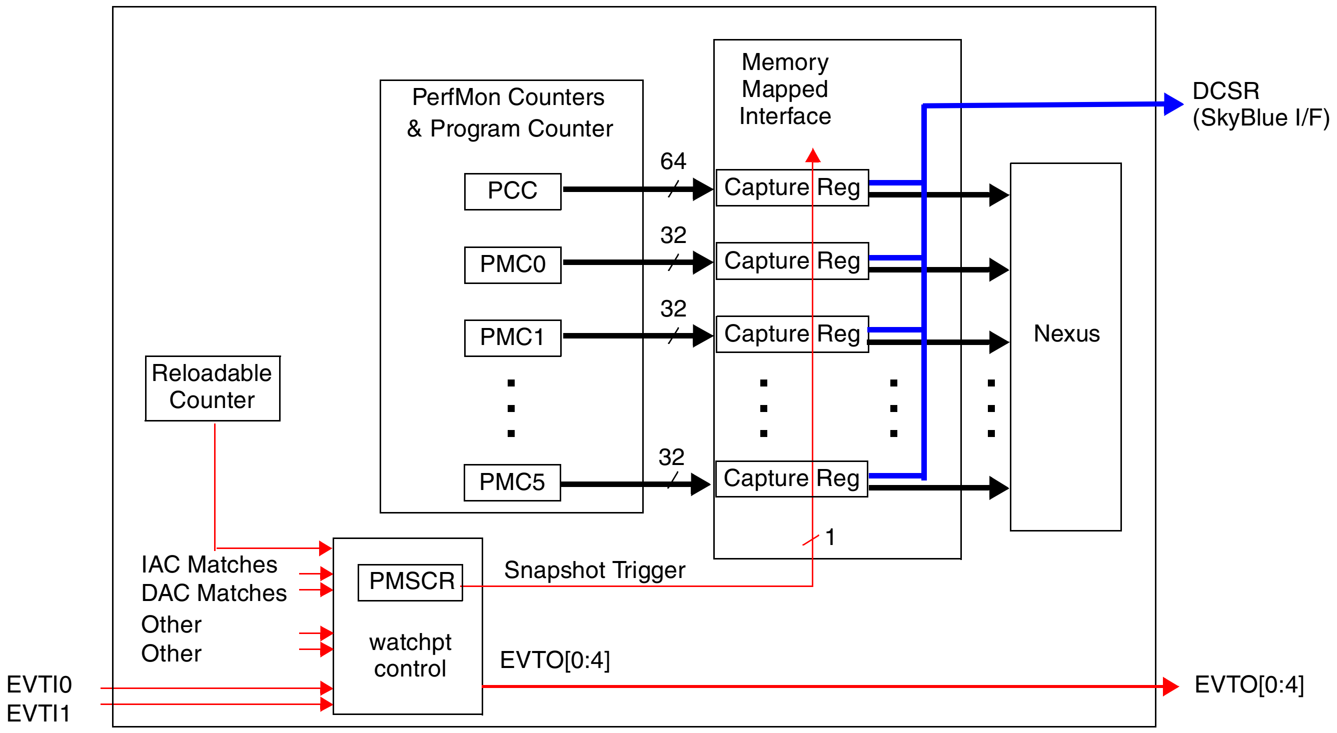


Figure 9-34. Processor performance monitor capture capability

NOTE

The EVTI signal, provided from the SoC, can be used to capture not only PMC counter and PC values from a single processor, but from all processors (or a subset of processors) on the SoC, as well as the SoC-level performance counters located in the event processing unit (EPU).

9.12.5 Examples

The following sections provide examples of how to use the performance monitor facility.

9.12.5.1 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, which produces an inaccurate value. A sequence similar to the following sequence 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.

9.12.6 Event selection

Event selection is specified through the PMLCa*n* registers described in Section 2.16.2, "Local control A registers (PMLCa0–PMLCa5/UPMLCa0–UPMLCa5)." The event-select fields in PMLCa*n*[EVENT] are described in Table 9-61, which lists encodings for the selectable events to be monitored. Table 9-61 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 double-word 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 no-op 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 double-word 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 no-op 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 in Table 9-61 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 are later flushed.
- Nonspeculative counts do not include speculative operations, which are flushed. Table 9-60 describes how event types are indicated in Table 9-61.

Table 9-60. Event types

Event Type	Label	Description
Reference	Ref:#	Shared across counters PMC0—PMC5. Applicable to most microprocessors.
Common	Com:#	Shared across counters PMC0–PMC5. Fairly specific to e500 microarchitectures.
Counter-specific	C[0–5]:#	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:#.

This table lists performance monitor events arranged by category.

Table 9-61. Performance monitor event selection (by category)

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 Counts 0, 1, or 2 per cycle
Com:3	Micro-ops completed	Nonspec	Completed micro-ops
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			

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:8	Branch instructions completed	Nonspec	Completed branch instructions
Com:67	Branches taken	Nonspec	Completed branch instructions that were taken
Com:68	**blr** taken	Nonspec	Completed **blr** instructions that were taken
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)
Com:9	Load micro-ops completed	Nonspec	Completed load micro-ops
Com:10	Store micro-ops completed	Nonspec	Completed store micro-ops
Com:181	LSU micro-ops completed	Nonspec	Completed Load Store Unit micro-ops—every micro-op that goes down the LSU pipe, which includes: • GPR loads / GPR stores • FPR loads / FPR stores • VR loads / VR stores • Cache ops • Memory barriers • Other LSU ops (**dsn**, **msgsnd**, **mvidsplt**, **mviwsplt**, **tlbilx**, **tlbivax**, **tlbsync**)
Com:182	GPR loads completed	Nonspec	GPR load micro-ops completed. This event only counts once for misaligns. Note that **lmw** that causes a fault may end up double-counting micro-ops—once for first pass, once for second pass.
Com:183	GPR stores completed	Nonspec	GPR store micro-ops completed. This event only counts once for misaligns. Note that **stmw** that causes a fault may end up double-counting micro-ops—once for first pass, once for second pass.
Com:184	Cache ops completed	Nonspec	Cache ops completed, which includes: • **dcba** / **dcbal** • **dcbf** / **dcbfep** • dcbi • **dcblc** / **dcblq.** • **dcbst** / **dcbstep** • **dcbt** / **dcbtep** / **dcbtls** • **dcbtst** / **dcbtstep** / **dcbtstls** • **dcbz** / **dcbzep** / **dcbzl** / **dcbzlep** • **icbi** / **icbiep** • **icblc** / **icblq.** • **icbt** / **icbtls**
Com:185	Memory barriers completed	Nonspec	Memory barriers completed, which includes: • msync (sync, lwsync, elemental barriers) • mbar (eieio) • miso
Com:186	SFX micro-ops completed	Nonspec	SFX micro-ops completed
Com:187	SFX single-cycle micro-ops completed	Nonspec	SFX single-cycle micro-ops completed

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:188	SFX double-cycle micro-ops completed	Nonspec	SFX double-cycle micro-ops completed
Com:190	CFX instructions completed	Nonspec	CFX instructions completed
Com:191	SFX or CFX instructions completed	Nonspec	SFX or CFX instructions completed
Com:192	FPU instructions completed	Nonspec	FPU instructions completed (non-LSU)
Com:193	FPR loads completed	Nonspec	FPR load micro-ops completed
Com:194	FPR stores completed	Nonspec	FPR store micro-ops completed
Com:195	FPR loads and stores completed	Nonspec	FPR load and store micro-ops completed
Com:196	FPR single-precision loads and stores completed	Nonspec	FPR single-precision load and store micro-ops completed
Com:197	FPR double-precision loads and stores completed	Nonspec	FPR double-precision load and store micro-ops completed
Com:198	AltiVec instructions completed	Nonspec	AltiVec instructions completed (non-LSU)
Com:199	AltiVec VSFX instructions completed	Nonspec	AltiVec VSFX instructions completed
Com:200	AltiVec VCFX instructions completed	Nonspec	AltiVec VCFX instructions completed
Com:201	AltiVec VPU instructions completed	Nonspec	AltiVec VPU instructions completed
Com:202	AltiVec VFPU instructions completed	Nonspec	AltiVec VFPU instructions completed
Com:203	VR loads completed	Nonspec	VR load micro-ops completed
Com:204	VR stores completed	Nonspec	VR store micro-ops completed
Com:205	VSCR[SAT] set	Nonspec	Number of times the saturate bit flips from 0 to 1
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
Com:69	Target mispredict (BTB)	Spec	Number of target mispredicts (BTB)
Com:70	Target blr mispredict (link stack)	Spec	Number of link stack mispredicts (LS)

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:71	BTB miss, but taken (BTB allocates)	Spec	Number of BTB misses, but taken (BTB allocates)
Com:72	BTB hit with phantom branch	Spec	Number of BTB hits with phantom branch
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 SFX/CFX issue stalled	Spec	Cycles the SFX/CFX issue queue is not empty but 0 instructions issued
Com:110	Cycles LSU issue stalled	Spec	Cycles the LSU issue queue is not empty but 0 instructions issued
Com:20	Cycles Branch issue stalled	Spec	Cycles the Branch issue queue is not empty but 0 instructions issued
Com:111	Cycles FPU issue stalled	Spec	Cycles the FPU issue queue is not empty but 0 instructions issued
Com:112	Cycles AltiVec issue stalled	Spec	Cycles the AltiVec issue queue 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 CFX schedule stalled	Spec	Cycles CFX is not empty but 0 instructions scheduled
Com:24	Cycles LSU schedule stalled	Spec	Cycles LSU is not empty but 0 instructions scheduled
Com:25	Cycles BU schedule stalled	Spec	Cycles BU is not empty but 0 instructions scheduled
Com:113	Cycles FPU schedule stalled	Spec	Cycles FPU is not empty but 0 instructions scheduled
Com:114	Cycles VPERM schedule stalled	Spec	Cycles VPERM is not empty but 0 instructions scheduled
Com:115	Cycles VGEN schedule stalled	Spec	Cycles VGEN is not empty but 0 instructions scheduled
Com:116	Cycles VPU instruction waits for operands	Spec	Cycles VPU instruction waits for operands
Com:117	Cycles VFPU instruction waits for operands	Spec	Cycles VFPU instruction waits for operands
Com:118	Cycles VSFX instruction waits for operands	Spec	Cycles VSFX instruction waits for operands
Com:119	Cycles VCFX instruction waits for operands	Spec	Cycles VCFX instruction waits for operands
Com:122	Cycles IB empty	Spec	Number of cycles the Instruction Buffer is empty
Com:123	Cycles IB full or close to full	Spec	Number of cycles the Instruction Buffer is full enough such that fetch stops fetching
Com:124	Cycles CB empty	Spec	Number of cycles the Completion Buffer is empty

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:125	Cycles CB full or close to full	Spec	Number of cycles the Completion Buffer is full enough such that decode stops
Com:127	Cycles 0 instructions completed	Spec	Increments if 0 instructions (micro-ops) completed
Com:128	Cycles 1 instruction completed	Spec	Increments if 1 instruction (micro-op) completed
Com:129	Cycles 2 instructions completed	Spec	Increments if 2 instructions (micro-ops) completed
Execution Unit Idle Events			
Com:210	Cycles SFX0 idle	Spec	Cycles Simple Fixed Point Unit 0 is idle
Com:211	Cycles SFX1 idle	Spec	Cycles Simple Fixed Point Unit 1 is idle
Com:212	Cycles CFX idle	Spec	Cycles Complex Fixed Point Unit is idle
Com:213	Cycles LSU idle	Spec	Cycles Load Store Unit is idle
Com:214	Cycles BU idle	Spec	Cycles Branch Unit is idle
Com:215	Cycles FPU idle	Spec	Cycles Floating Point Unit is idle
Com:216	Cycles VPU idle	Spec	Cycles AltiVec Permute Unit is idle
Com:217	Cycles VFPU idle	Spec	Cycles AltiVec Floating Point Unit is idle
Com:218	Cycles VSFX idle	Spec	Cycles AltiVec Simple Fixed Point Unit is idle
Com:219	Cycles VCFX idle	Spec	Cycles AltiVec Complex Fixed Point Unit is idle
Load/Store and Data Cache Events			
Com:26	Total translated	Spec	Total LSU micro-ops that reach the second stage of the LSU.[1] **Note:** If instruction is replayed or misaligned, it is still counted just once.
Com:27	Loads translated	Spec	Cacheable load micro-ops translated[1] (does not include WT.)
Com:28	Stores translated	Spec	Cacheable store micro-ops translated[1] (does not include WT.)
Com:29	Touches translated	Spec	Cacheable touch instructions translated, which includes: • dcbt / dcbtep • **dcbtst** / **dcbtstep** • icbt ct=2 (Does not include touches that are converted to no-ops.) (Does not include **dcbtls** / **dcbtstls** / **icbtls**.)
Com:30	Cache ops translated	Spec	Cache op instructions translated, which includes: • **dcba** / **dcbal** • **dcbf** / **dcbfep** • dcbi • **dcbst** / **dcbstep** • **dcbz** / **dcbzep** / **dcbzl** / **dcbzlep**
Com:31	Cache-inhibited accesses translated	Spec	Cache inhibited load and store accesses translated
Com:32	Guarded loads translated	Spec	Guarded loads and decorated CI loads translated

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
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:221	Data L1 cache misses	Spec	Data L1 cache misses (includes load, store, cache ops)
Com:222	Data L1 cache load misses	Spec	Data L1 cache load misses
Com:223	Data L1 cache store misses	Spec	Data L1 cache store misses
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:224	Loads that allocate into LMQ	Spec	Loads that allocate into Load Miss Queue. (Data L1 cache misses, but may not be to different cache lines).
Com:225	Load thread miss collision	Spec	Number of times that this thread’s load hits a line that is valid for the other thread but not this thread
Com:226	Inter-thread status array collision	Spec	Number of times that two threads collide on status array access
Com:227	SGB allocates	Spec	Number of Store Gather Buffer allocates
Com:228	SGB gathers	Spec	Number of Store Gather Buffer gathers
Com:229	SGB overflows	Spec	Number of Store Gather Buffer overflows. (Causes SGB full condition when additional store request is made)
Com:230	SGB promotions	Spec	Number of Store Gather Buffer promotions
Com:231	SGB in-order promotions	Spec	Number of Store Gather Buffer in-order promotions (also includes oldest-entry timeout condition)
Com:232	SGB out-of-order promotions	Spec	Number of Store Gather Buffer out-of-order promotions
Com:233	SGB high-priority promotions	Spec	Number of Store Gather Buffer high-priority promotions (load hits on pending store)
Com:234	SGB miso promotions	Spec	Number of Store Gather Buffer miso promotions
Com:235	SGB watermark promotions	Spec	Number of Store Gather Buffer watermark promotions
Com:236	SGB overflow promotions	Spec	Number of Store Gather Buffer overflow promotions
Com:237	DLAQ full cycles	Spec	Number of cycles the DLink Age Queue is full
Com:238	DLAQ full times	Spec	Number of times the DLink Age Queue is full
Com:239	LRSAQ full cycles	Spec	Number of cycles the Load Reservation Set Age Queue is full
Com:240	LRSAQ full times	Spec	Number of times the Load Reservation Set Age Queue is full
Com:241	FWDAQ full cycles	Spec	Number of cycles the Forward Age Queue is full
Com:242	FWDAQ full times	Spec	Number of times the Forward Age Queue is full

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:44	Load miss with load queue full	Spec	Counts number of stalls; Com:52 counts cycles stalled. which includes: • cacheable loads • CI loads • loadec • load and reserve • touches • memory barriers
Com:45	Load guarded miss when the load is not yet at the bottom of the CB	Spec	Counts number of stalls; Com:53 counts cycles stalled, which includes: • guarded loads • decorated CI loads
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 (load on store)	Spec	Counts number of stalls; Com:55 counts cycles stalled.
Com:243	STQ collision forwardable (times)	Spec	Number of times a Store Queue collision is forwardable The following cases are not forwardable: • store address + size does not contain the load • cache-inhibited store • denormalized floating point store • store conditional. • guarded load
Com:244	STQ collision forwardable (times data ready)	Spec	Number of times a Store Queue collision is forwardable and is ready with data to forward
Com:245	STQ collision forwardable (times data not ready)	Spec	Number of times a Store Queue collision is forwardable but is not ready with data to forward
Com:246	STQ collision not-forwardable (times not forwardable)	Spec	Number of times a Store Queue collision is not forwardable and must wait until the store leaves the Store Queue
Com:247	STQ collision forwardable (cycles)	Spec	Number of cycles a Store Queue collision is forwardable (Number of cycles from the detection of a forwardable Store Queue entry until the load is replayed in stg1)
Com:248	STQ collision forwardable (cycles data ready)	Spec	Number of cycles a Store Queue collision is forwardable and is ready with data to forward (Number of cycles from the detection of a forwardable Store Queue entry with valid data until the load is replayed in stg1)
Com:249	STQ collision forwardable (cycles data not ready)	Spec	Number of cycles a Store Queue collision is forwardable but is not ready with data to forward (Number of cycles from the detection of a forwardable Store Queue entry without valid data until the load is replayed in stg1)
Com:250	STQ collision non-forwardable (cycles not forwardable)	Spec	Number of cycles a Store Queue collision is not forwardable and has to wait until the store leaves the Store Queue (Number of cycles from the detection of a non-forwardable Store Queue entry until the load is replayed in stg1)

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:251	False EA (load-on store) collisions (times)	Spec	Number of times the lower 12-bits of EA matched but the upper bits did not, leading to a false load-on-store replay. Cycle penalty is 4x the number of times.
Com:48	DTLB miss times	Spec	Counts number of stalls; Com:56 counts cycles stalled.
Com:49	DTLB busy times	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.
Com:52	Load miss with load queue full	Spec	Counts cycles stalled; Com:44 counts number of stalls, which includes: • cacheable loads • CI loads • loadec • load and reserve • touches • memory barriers
Com:53	Load guarded miss when the load is not yet at the bottom of the CB	Spec	Counts cycles stalled; Com:45 counts number of stalls which includes: • guarded loads • decorated CI loads
Com:252	LS0 result bus collisions	Spec	Number of LS0 result bus collisions. Cycle penalty is 3x this measurement.
Com:253	Inter-thread doubleword bank collisions	Spec	Number of inter-thread double-word bank collisions. Measures when both threads attempt to access the same double-word bank. Cycle penalty is 3x this measurement.
Com:54	Translate a store when the store queue is full cycles.	Spec	Counts cycles stalled; Com:46 counts number of stalls.
Com:55	Address collision cycles (load on store)	Spec	Counts cycles stalled; Com:47 counts number of stalls.
Com:56	DTLB miss cycles	Spec	Counts cycles stalled; Com:48 counts number of stalls.
Com:57	DTLB busy cycles	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 and Instruction Cache Events			
Com:254	Instruction L1 cache misses	Spec	Instruction L1 cache demand fetch misses. Includes **icbtls**. Does not include prefetch.
Com:60	Instruction L1 cache reloads from fetch	Spec	Instruction L1 cache reloads due to demand fetch. Includes **icbtls**. Does not include prefetch. 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 Instruction Buffer

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:35	Fetch 2x4 hits	Spec	Each fetch retrieves up to 8 instructions, but only the first 4 are required. This event increments if at least one instruction of the second 4 are actually used.
Com:36	Fetch hits on prefetches	Spec	Fetch hits on instruction prefetch when the data is still in the ILFB
Com:37	Fetch prefetches generated	Spec	Number of prefetches generated
Instruction MMU, Data MMU and L2 MMU Events			
Com:62	IMMU TLB-4K reloads	Spec	Counts reloads in the level 1 Instruction MMU TLB-4K.þ A reload in the level 2 MMU TLB-4K is not counted.
Com:63	IMMU VSP reloads	Spec	Counts reloads in the level 1 Instruction MMU VSP.þ A reload in the level 2 MMU VSP is not counted.
Com:256	IMMU misses	Spec	Counts misses in the level 1 Instruction MMU
Com:257	IMMU TLB-4K hits	Spec	Counts hits in the level 1 Instruction MMU TLB-4K
Com:258	IMMU VSP hits	Spec	Counts hits in the level 1 Instruction MMU VSP
Com:259	IMMU cycles spent in hardware tablewalk	Spec	Counts IMMU cycles spent in hardware tablewalk. This represents the cycles from the point where the L2 MMU miss occurs to when the page table walk completes with a valid translation or exception.
Com:64	DMMU TLB-4K reloads	Spec	Counts reloads in the level 1 Data MMU TLB-4K. þA reload in the level 2 MMU TLB-4K is not counted.
Com:65	DMMU VSP reloads	Spec	Counts reloads in the level 1 Data MMU VSP. A reload in the level 2 MMU VSP is not counted.
Com:260	DMMU misses	Spec	Counts misses in the level 1 Data MMU. (Does not count replayed operations).
Com:261	DMMU TLB-4K hits	Spec	Counts hits in the level 1 Data MMU TLB-4K. (Does not count replayed operations).
Com:262	DMMU VSP hits	Spec	Counts hits in the level 1 Data MMU VSP. (Does not count replayed operations).
Com:263	DMMU cycles spent in hardware tablewalk	Spec	Counts DMMU cycles spent in hardware tablewalk. This represents the cycles from the point where the L2 MMU miss occurs to when the page table walk completes with a valid translation or exception.
Com:264	L2MMU misses	Spec	Counts level 2 MMU misses. (Does not count misses that occur due to **dcbt** / **dcbtst** / **dcba** / **dcbal** instructions that fail translation and are no-oped. Does not count misses in L2MMU-VSP when looking up an indirect entry).
Com:265	L2MMU hits in L2MMU-4K	Spec	Counts level 2 MMU hits in L2MMU-4K.
Com:266	L2MMU hits in L2MMU-VSP	Spec	Counts level 2 MMU hits in L2MMU-VSP. (Does not count indirect lookups)
Com:66	L2MMU error misses	Nonspec	Counts instruction TLB / data TLB error interrupts. This represents L2MMU misses that occur during translation.

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:267	L2MMU indirect misses	Spec	Counts level 2 MMU indirect misses. This represents indirect entry lookups that do not have a matching indirect entry.
Com:268	L2MMU indirect valid misses	Spec	Counts level 2 MMU indirect valid misses. This occurs when the indirect entry is valid, but the corresponding PTE[V] = 0 or the permissions in the PTE are not sufficient for the requested access.
Com:269	LRAT misses	Spec	Counts Logical to Real Address Translation misses. This includes LRAT misses from tlbwe instructions or from page table translations.
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.
Com:91	PMC4 overflow	N/A	PMC4[32] transitioned from 1 to 0.
Com:92	PMC5 overflow	N/A	PMC5[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]
L1 Stashing Events			
Com:97	Stash hit to L1 Data Cache	N/A	Stash hits in L1 Data Cache
Com:99	Stash requests to L1 Data Cache	N/A	Stash requests to L1 Data Cache
Thread Events			
Com:100	Number of times LSU thread priority switched	N/A	Number of times the Load Store Unit thread priority switched based on resource collisions (doubleword bank, DL1 status array, and so on).
Com:101	Number of cycles both threads had FPU requests and one was denied	N/A	Number of cycles both threads had Floating Point Unit requests and one was denied
Com:102	Number of cycles both threads had VPERM requests and one was denied	N/A	Number of cycles both threads had Altivec Permute requests and one was denied
Com:103	Number of cycles both threads had VGEN requests and one was denied	N/A	Number of cycles both threads had Altivec General requests and one was denied

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:104	Number of cycles both threads had CFX requests and one was denied	N/A	Number of cycles both threads had Complex Fixed-Point Unit requests and one was denied
Com:105	Number of cycles both threads had Fetch requests and one was denied	N/A	Number of cycles both threads both threads made a Fetch request to the L1 Instruction Cache and one thread wins arbitration
IAC and DAC Events			
Com:140	IAC1s detected	Nonspec	Every valid IAC1 detection
Com:141	IAC2s detected	Nonspec	Every valid IAC2 detection
Com:142	IAC3s detected	Nonspec	Every valid IAC3 detection
Com:143	IAC4s detected	Nonspec	Every valid IAC4 detection
Com:136	IAC5s detected	Nonspec	Every valid IAC5 detection
Com:137	IAC6s detected	Nonspec	Every valid IAC6 detection
Com:138	IAC7s detected	Nonspec	Every valid IAC7 detection
Com:139	IAC8s detected	Nonspec	Every valid IAC8 detection
Com:144	DAC1s detected	Nonspec	Every valid DAC1 detection
Com:145	DAC2s detected	Nonspec	Every valid DAC2 detection
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)	Spec	Number of completion cycles stalled due to Nexus FIFO full
FPU Events			
Com:161	FPU finish	Spec	FPU finish.
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 denorm output	Spec	FPU denorm output

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:165	FPU FPSCR full stall	Spec	FPU FPSCR stall
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 cannot issue. 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.
Com:168	FPU instruction generates flags	Spec	FPU instruction sets FPSCR[FEX].
		Power Management Events	
Com:172	PW20 count	N/A	Number of times the core enters the PW20 power management state
		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:179	**stbcx.**, **sthcx.**, **stwcx.**, or **stdcx.** successful	Nonspec	Number of successful **stbcx.**, **sthcx.**, **stwcx.**, or **stdcx.** instructions
Com:180	**stbcx.**, **sthcx.**, **stwcx.**, or **stdcx.** unsuccessful	Nonspec	Number of unsuccessful **stbcx.**, **sthcx.**, **stwcx.**, or **stdcx.** instructions
Com:272	Cycles LMQ loses DLINK arbitration due to SGB	Spec	Cycles the Load Miss Queue loses DLINK arbitration due to the Store Gather Buffer
Com:273	Cycles SGB loses DLINK arbitration due to LMQ	Spec	Cycles the Store Gather Buffer loses DLINK arbitration due to the Load Miss Queue
Com:274	Cycles thread loses DLINK arbitration due to other thread	Spec	Cycles thread loses DLINK arbitration due to other thread
		eLink Events	
Com:443	DLINK request	N/A	Number of DLINK requests made from core to Shared L2
Com:444	ILINK request	N/A	Number of ILINK requests made from core to Shared L2. (Includes instruction fetches and L2MMU hardware tablewalk requests)
Com:445	RLINK request	N/A	Number of RLINK requests made from Shared L2 to core. (reload data)
Com:446	BLINK request	N/A	Number of BLINK requests made from Shared L2 to core. (back invalidates, stashes, barriers)
Com:447	CLINK request	N/A	Number of CLINK requests made from Shared L2 to core. (CoreNet data forwarding)
		Shared L2 Events	
Com:456	L2 hits	N/A	Number of L2 Cache hits Counts 0, 1, 2, 3, or 4 per cycle

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:457	L2 misses	N/A	Number of L2 Cache misses Counts 0, 1, 2, 3, or 4 per cycle
Com:458	L2 demand accesses	N/A	Number of L2 Cache demand accesses Counts 0, 1, 2, 3, or 4 per cycle
Com:459	L2 accesses	N/A	Number of L2 Cache accesses from all sources (demand, reload, snoop, and so on) Counts 0, 1, 2, 3, or 4 per cycle
Com:460	L2 store allocates	N/A	Number of L2 Cache store allocates Counts 0, 1, 2, 3, or 4 per cycle
Com:461	L2 instruction accesses	N/A	Number of L2 Cache instruction accesses Counts 0, 1, 2, 3, or 4 per cycle
Com:462	L2 data accesses	N/A	Number of L2 Cache data accesses Counts 0, 1, 2, 3, or 4 per cycle
Com:463	L2 instruction misses	N/A	Number of L2 Cache instruction misses Counts 0, 1, 2, 3, or 4 per cycle
Com:464	L2 data misses	N/A	Number of L2 Cache data misses Counts 0, 1, 2, 3, or 4 per cycle
Com:465	L2 hits per thread	N/A	Number of times this core/thread hits in the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:466	L2 misses per thread	N/A	Number of times this core/thread misses in the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:467	L2 demand accesses per thread	N/A	Number of times this core/thread makes a demand access to the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:468	L2 store allocates per thread	N/A	Number of times a store from this core/thread allocates in the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:469	L2 instruction accesses per thread	N/A	Number of times an instruction from this core/thread accesses the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:470	L2 data accesses per thread	N/A	Number of times a data operation from this core/thread accesses the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:471	L2 instruction misses per thread	N/A	Number of times an instruction from this core/thread misses in the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:472	L2 data misses per thread	N/A	Number of times a data operation from this core/thread misses in the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:473	L2 reloads from CoreNet	N/A	Number of L2 Cache reloads from CoreNet Counts 0, 1, 2, 3, or 4 per cycle

Table 9-61. Performance monitor event selection (by category) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:474	L2 stash requests	N/A	Number of incoming L2 Cache stash requests Counts 0, 1, 2, 3, or 4 per cycle
Com:475	L2 stash requests downgraded to snoops	N/A	Number of incoming L2 Cache stash requests downgraded to snoops Counts 0, 1, 2, 3, or 4 per cycle
Com:476	L2 snoop hits	N/A	Number of L2 Cache snoop hits Counts 0, 1, 2, 3, or 4 per cycle
Com:477	L2 snoops causing MINT	N/A	Number of L2 Cache snoops causing MINT
Com:478	L2 snoops causing SINT	N/A	Number of L2 Cache snoops causing SINT
Com:479	L2 snoop pushes	N/A	Number of L2 Cache snoop pushes
Com:480	Stall for BIB cycles	N/A	Stall for Back Invalidate Buffer entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:482	Stall for RLT cycles	N/A	Stall for Reload Table entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:484	Stall for RLFQ cycles	N/A	Stall for Reload Fold Queue entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:486	Stall for DTQ cycles	N/A	Stall for Data Transaction Queue entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:488	Stall for COB cycles	N/A	Stall for Castout Buffer entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:490	Stall for WDB cycles	N/A	Stall for Write Data Buffer entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:492	Stall for RLDB cycles	N/A	Stall for Reload Data Buffer entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:494	Stall for SNPQ cycles	N/A	Stall for Snoop Queue entry (cycles)
BIU Events			
Com:506	BIU master requests	N/A	Master transaction starts. (Number of AOut sent to CoreNet)
Com:507	BIU master global requests	N/A	Master transaction starts that are global. (Number of AOut with M=1 sent to CoreNet)
Com:508	BIU master data-side requests	N/A	Master data-side transaction starts. (Number of D-side AOut sent to CoreNet)
Com:509	BIU master instruction-side requests	N/A	Master instruction-side transaction starts. (Number of I-side AOut sent to CoreNet)
Com:510	Stash requests	N/A	Stash request on AIn matches stash IDs for core or L2
Com:511	Snoop requests	N/A	Externally generated snoop requests. (Number of AIn from CoreNet not from self)

[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.

This table lists performance monitor events arranged by number.

Table 9-62. Performance monitor event selection (by number)

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 Counts 0, 1, or 2 per cycle
Com:3	Micro-ops completed	Nonspec	Completed micro-ops
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
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**, **rfgi**, **rfci**, **rfdi**, **rfmci**, **isync**, and interrupts)
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
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.
Com:18	Cycles decode stalled	Spec	Cycles the IQ is not empty but 0 instructions decoded
Com:19	Cycles SFX/CFX issue stalled	Spec	Cycles the SFX/CFX issue queue is not empty but 0 instructions issued
Com:20	Cycles Branch issue stalled	Spec	Cycles the Branch issue queue is not empty but 0 instructions issued

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
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 CFX schedule stalled	Spec	Cycles CFX is not empty but 0 instructions scheduled
Com:24	Cycles LSU schedule stalled	Spec	Cycles LSU is not empty but 0 instructions scheduled
Com:25	Cycles BU schedule stalled	Spec	Cycles BU is not empty but 0 instructions scheduled
Com:26	Total translated	Spec	Total LSU micro-ops that reach the second stage of the LSU[1] **Note:** If instruction is replayed or misaligned, it is still counted just once.
Com:27	Loads translated	Spec	Cacheable load micro-ops translated[1] (does not include WT)
Com:28	Stores translated	Spec	Cacheable store micro-ops translated[1] (does not include WT)
Com:29	Touches translated	Spec	Cacheable touch instructions translated, which includes: • dcbt / dcbtep • **dcbtst** / **dcbtstep** • icbt ct=2 (Does not include touches that are converted to no-ops) (Does not include **dcbtls** / **dcbtstls** / **icbtls**)
Com:30	Cache ops translated	Spec	Cache op instructions translated, which includes: • **dcba** / **dcbal** • **dcbf** / **dcbfep** • dcbi • **dcbst** / **dcbstep** • **dcbz** / **dcbzep** / **dcbzl** / **dcbzlep**
Com:31	Cache-inhibited accesses translated	Spec	Cache inhibited load and store accesses translated
Com:32	Guarded loads translated	Spec	Guarded loads and decorated CI 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	Fetch 2x4 hits	Spec	Each fetch retrieves up to 8 instructions, but only the first 4 are required. This event increments if at least one instruction of the second 4 are actually used.
Com:36	Fetch hits on prefetches	Spec	Fetch hits on instruction prefetch when the data is still in the ILFB
Com:37	Fetch prefetches generated	Spec	Number of prefetches generated
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).

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:44	Load miss with load queue full	Spec	Counts number of stalls; Com:52 counts cycles stalled, which ncludes: • cacheable loads • CI loads • loadec • load and reserve • touches • memory barriers
Com:45	Load guarded miss when the load is not yet at the bottom of the CB	Spec	Counts number of stalls; Com:53 counts cycles stalled, which includes: • guarded loads • decorated CI loads
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 (load on store)	Spec	Counts number of stalls; Com:55 counts cycles stalled.
Com:48	DTLB miss times	Spec	Counts number of stalls; Com:56 counts cycles stalled.
Com:49	DTLB busy times	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.
Com:52	Load miss with load queue full	Spec	Counts cycles stalled; Com:44 counts number of stalls, which includes: • cacheable loads • CI loads • loadec • load and reserve • touches • memory barriers
Com:53	Load guarded miss when the load is not yet at the bottom of the CB	Spec	Counts cycles stalled; Com:45 counts number of stalls, which includes: • guarded loads • decorated CI loads
Com:54	Translate a store when the store queue is full cycles.	Spec	Counts cycles stalled; Com:46 counts number of stalls.
Com:55	Address collision cycles (load on store)	Spec	Counts cycles stalled; Com:47 counts number of stalls.
Com:56	DTLB miss cycles	Spec	Counts cycles stalled; Com:48 counts number of stalls.
Com:57	DTLB busy cycles	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.
Com:60	Instruction L1 cache reloads from fetch	Spec	Instruction L1 cache reloads due to demand fetch. (Includes **icbtls**. Does not include prefetch.) Typically used to determine instruction cache miss rate along with instructions completed.

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:61	Number of fetches	Spec	Counts fetches that write at least one instruction to the Instruction Buffer
Com:62	IMMU TLB-4K reloads	Spec	Counts reloads in the level 1 Instruction MMU TLB-4K.þ A reload in the level 2 MMU TLB-4Kis not counted
Com:63	IMMU VSP reloads	Spec	Counts reloads in the level 1 Instruction MMU VSP.þ A reload in the level 2 MMU VSP is not counted
Com:64	DMMU TLB-4K reloads	Spec	Counts reloads in the level 1 Data MMU TLB-4K. þA reload in the level 2 MMU TLB-4K is not counted
Com:65	DMMU 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 error misses	Nonspec	Counts instruction TLB / data TLB error interrupts. This represents L2MMU misses that occur during translation
Com:67	Branches taken	Nonspec	Completed branch instructions that were taken
Com:68	**blr** taken	Nonspec	Completed **blr** instructions that were taken
Com:69	Target mispredict (BTB)	Spec	Number of target mispredicts (BTB)
Com:70	Target blr mispredict (link stack)	Spec	Number of link stack mispredicts (LS)
Com:71	BTB miss, but taken (BTB allocates)	Spec	Number of BTB misses, but taken (BTB allocates)
Com:72	BTB hit with phantom branch	Spec	Number of BTB hits with phantom branch
Com:82	PMC0 overflow	N/A	PMC0[32] transitions from 1 to 0.[2]
Com:83	PMC1 overflow	N/A	PMC1[32] transitions from 1 to 0.[2]
Com:84	PMC2 overflow	N/A	PMC2[32] transitions from 1 to 0.[2]
Com:85	PMC3 overflow	N/A	PMC3[32] transitioned from 1 to 0.[2]
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	—
Com:90	Transitions of TBL bit selected by PMGC0[TBSEL].	Nonspec	Counts transitions of the TBL bit selected by PMGC0[TBSEL]
Com:91	PMC4 overflow	N/A	PMC4[32] transitions from 1 to 0.
Com:92	PMC5 overflow	N/A	PMC5[32] transitions from 1 to 0.
Com:97	Stash hit to L1 Data Cache	N/A	Stash hits in L1 Data Cache
Com:99	Stash requests to L1 Data Cache	N/A	Stash requests to L1 Data Cache
Com:100	Number of times LSU thread priority switched	N/A	Number of times the Load Store Unit thread priority switched based on resource collisions (doubleword bank, DL1 status array, and so on).

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:101	Number of cycles both threads had FPU requests and one was denied	N/A	Number of cycles both threads had Floating Point Unit requests and one was denied.
Com:102	Number of cycles both threads had VPERM requests and one was denied	N/A	Number of cycles both threads had Altivec Permute requests and one was denied.
Com:103	Number of cycles both threads had VGEN requests and one was denied	N/A	Number of cycles both threads had Altivec General requests and one was denied.
Com:104	Number of cycles both threads had CFX requests and one was denied	N/A	Number of cycles both threads had Complex Fixed-Point Unit requests and one was denied.
Com:105	Number of cycles both threads had Fetch requests and one was denied	N/A	Number of cycles both threads both threads made a Fetch request to the L1 Instruction Cache and one thread wins arbitration.
Com:110	Cycles LSU issue stalled	Spec	Cycles the LSU issue queue is not empty but 0 instructions issued
Com:111	Cycles FPU issue stalled	Spec	Cycles the FPU issue queue is not empty but 0 instructions issued
Com:112	Cycles AltiVec issue stalled	Spec	Cycles the AltiVec issue queue is not empty but 0 instructions issued
Com:113	Cycles FPU schedule stalled	Spec	Cycles FPU is not empty but 0 instructions scheduled
Com:114	Cycles VPERM schedule stalled	Spec	Cycles VPERM is not empty but 0 instructions scheduled
Com:115	Cycles VGEN schedule stalled	Spec	Cycles VGEN is not empty but 0 instructions scheduled
Com:116	Cycles VPU instruction waits for operands	Spec	Cycles VPU instruction waits for operands
Com:117	Cycles VFPU instruction waits for operands	Spec	Cycles VFPU instruction waits for operands
Com:118	Cycles VSFX instruction waits for operands	Spec	Cycles VSFX instruction waits for operands
Com:119	Cycles VCFX instruction waits for operands	Spec	Cycles VCFX instruction waits for operands
Com:122	Cycles IB empty	Spec	Number of cycles the Instruction Buffer is empty
Com:123	Cycles IB full or close to full	Spec	Number of cycles the Instruction Buffer is full enough such that fetch stops fetching
Com:124	Cycles CB empty	Spec	Number of cycles the Completion Buffer is empty
Com:125	Cycles CB full or close to full	Spec	Number of cycles the Completion Buffer is full enough such that decode stops
Com:126	Cycles a pre-sync serialized instruction holds in IB	Spec	Number of cycles a pre-sync serialized instruction holds in the Instruction Buffer and is not decoded
Com:127	Cycles 0 instructions completed	Spec	Increments if 0 instructions (micro-ops) completed
Com:128	Cycles 1 instruction completed	Spec	Increments if 1 instruction (micro-op) completed

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:129	Cycles 2 instructions completed	Spec	Increments if 2 instructions (micro-ops) completed
Com:136	IAC5s detected	Nonspec	Every valid IAC5 detection
Com:137	IAC6s detected	Nonspec	Every valid IAC6 detection
Com:138	IAC7s detected	Nonspec	Every valid IAC7 detection
Com:139	IAC8s detected	Nonspec	Every valid IAC8 detection
Com:140	IAC1s detected	Nonspec	Every valid IAC1 detection
Com:141	IAC2s detected	Nonspec	Every valid IAC2 detection
Com:142	IAC3s detected	Nonspec	Every valid IAC3 detection
Com:143	IAC4s detected	Nonspec	Every valid IAC4 detection
Com:144	DAC1s detected	Nonspec	Every valid DAC1 detection
Com:145	DAC2s detected	Nonspec	Every valid DAC2 detection
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)	Spec	Number of completion cycles stalled due to Nexus FIFO full
Com:161	FPU finish	Spec	FPU finish.
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 denorm output	Spec	FPU denorm output
Com:165	FPU FPSCR full stall	Spec	FPU FPSCR stall
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 cannot issue. 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.

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:168	FPU instruction generates flags	Spec	FPU instruction sets FPSCR[FEX].
Com:172	PW20 count	N/A	Number of times the core enters the PW20 power management state
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:179	**stbcx.**, **sthcx.**, **stwcx.**, or **stdcx.** successful	Nonspec	Number of successful **stbcx.**, **sthcx.**, **stwcx.**, or **stdcx.** instructions
Com:180	**stbcx.**, **sthcx.**, **stwcx.**, or **stdcx.** unsuccessful	Nonspec	Number of unsuccessful **stbcx.**, **sthcx.**, **stwcx.**, or **stdcx.** instructions
Com:181	LSU micro-ops completed	Nonspec	Completed Load Store Unit micro-ops. Every micro-op that goes down the LSU pipe, which includes: • GPR loads / GPR stores • FPR loads / FPR stores • VR loads / VR stores • Cache ops • Memory barriers Other LSU ops (**dsn**, **msgsnd**, **mvidsplt**, **mviwsplt**, **tlbilx**, **tlbivax**, **tlbsync**)
Com:182	GPR loads completed	Nonspec	GPR load micro-ops completed. This event only counts once for misaligns. Note that lmw that causes a fault may end up double-counting micro-ops -- once for first pass, once for second pass.
Com:183	GPR stores completed	Nonspec	GPR store micro-ops completed. This event only counts once for misaligns. Note that stmw that causes a fault may end up double-counting micro-ops -- once for first pass, once for second pass.
Com:184	Cache ops completed	Nonspec	Cache ops completed, which ncludes: • **dcba** / **dcbal** • **dcbf** / **dcbfep** • dcbi • **dcblc** / **dcblq.** • **dcbst** / **dcbstep** • **dcbt** / **dcbtep** / **dcbtls** • **dcbtst** / **dcbtstep** / **dcbtstls** • **dcbz** / **dcbzep** / **dcbzl** / **dcbzlep** • **icbi** / **icbiep** • **icblc** / **icblq.** • **icbt** / **icbtls**
Com:185	Memory barriers completed	Nonspec	Memory barriers completed, which includes: • msync (sync, lwsync, elemental barriers) • mbar (eieio) • miso
Com:186	SFX micro-ops completed	Nonspec	SFX micro-ops completed
Com:187	SFX single-cycle micro-ops completed	Nonspec	SFX single-cycle micro-ops completed

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:188	SFX double-cycle micro-ops completed	Nonspec	SFX double-cycle micro-ops completed
Com:190	CFX instructions completed	Nonspec	CFX instructions completed
Com:191	SFX or CFX instructions completed	Nonspec	SFX or CFX instructions completed
Com:192	FPU instructions completed	Nonspec	FPU instructions completed (non-LSU)
Com:193	FPR loads completed	Nonspec	FPR load micro-ops completed
Com:194	FPR stores completed	Nonspec	FPR store micro-ops completed
Com:195	FPR loads and stores completed	Nonspec	FPR load and store micro-ops completed
Com:196	FPR single-precision loads and stores completed	Nonspec	FPR single-precision load and store micro-ops completed
Com:197	FPR double-precision loads and stores completed	Nonspec	FPR double-precision load and store micro-ops completed
Com:198	AltiVec instructions completed	Nonspec	AltiVec instructions completed (non-LSU)
Com:199	AltiVec VSFX instructions completed	Nonspec	AltiVec VSFX instructions completed
Com:200	AltiVec VCFX instructions completed	Nonspec	AltiVec VCFX instructions completed
Com:201	AltiVec VPU instructions completed	Nonspec	AltiVec VPU instructions completed
Com:202	AltiVec VFPU instructions completed	Nonspec	AltiVec VFPU instructions completed
Com:203	VR loads completed	Nonspec	VR load micro-ops completed
Com:204	VR stores completed	Nonspec	VR store micro-ops completed
Com:205	VSCR[SAT] set	Nonspec	Number of times the saturate bit flips from 0 to 1
Com:210	Cycles SFX0 idle	Spec	Cycles Simple Fixed Point Unit 0 is idle
Com:211	Cycles SFX1 idle	Spec	Cycles Simple Fixed Point Unit 1 is idle
Com:212	Cycles CFX idle	Spec	Cycles Complex Fixed Point Unit is idle
Com:213	Cycles LSU idle	Spec	Cycles Load Store Unit is idle
Com:214	Cycles BU idle	Spec	Cycles Branch Unit is idle
Com:215	Cycles FPU idle	Spec	Cycles Floating Point Unit is idle
Com:216	Cycles VPU idle	Spec	Cycles AltiVec Permute Unit is idle
Com:217	Cycles VFPU idle	Spec	Cycles AltiVec Floating Point Unit is idle
Com:218	Cycles VSFX idle	Spec	Cycles AltiVec Simple Fixed Point Unit is idle
Com:219	Cycles VCFX idle	Spec	Cycles AltiVec Complex Fixed Point Unit is idle
Com:221	Data L1 cache misses	Spec	Data L1 cache misses. (Includes load, store, cache ops)

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:222	Data L1 cache load misses	Spec	Data L1 cache load misses
Com:223	Data L1 cache store misses	Spec	Data L1 cache store misses
Com:224	Loads that allocate into LMQ	Spec	Loads that allocate into Load Miss Queue (data L1 cache misses, but may not be to different cache lines)
Com:225	Load thread miss collision	Spec	Number of times that this thread's load hits a line that is valid for the other thread but not this thread
Com:226	inter-thread status array collision	Spec	Number of times that two threads collide on status array access
Com:227	SGB allocates	Spec	Number of Store Gather Buffer allocates
Com:228	SGB gathers	Spec	Number of Store Gather Buffer gathers
Com:229	SGB overflows	Spec	Number of Store Gather Buffer overflows (causes SGB full condition when additional store request is made)
Com:230	SGB promotions	Spec	Number of Store Gather Buffer promotions
Com:231	SGB in-order promotions	Spec	Number of Store Gather Buffer in-order promotions (also includes oldest-entry timeout condition)
Com:232	SGB out-of-order promotions	Spec	Number of Store Gather Buffer out-of-order promotions
Com:233	SGB high-priority promotions	Spec	Number of Store Gather Buffer high-priority promotions (load hits on pending store)
Com:234	SGB miso promotions	Spec	Number of Store Gather Buffer miso promotions
Com:235	SGB watermark promotions	Spec	Number of Store Gather Buffer watermark promotions
Com:236	SGB overflow promotions	Spec	Number of Store Gather Buffer overflow promotions
Com:237	DLAQ full cycles	Spec	Number of cycles the DLink Age Queue is full
Com:238	DLAQ full times	Spec	Number of times the DLink Age Queue is full
Com:239	LRSAQ full cycles	Spec	Number of cycles the Load Reservation Set Age Queue is full
Com:240	LRSAQ full times	Spec	Number of times the Load Reservation Set Age Queue is full
Com:241	FWDAQ full cycles	Spec	Number of cycles the Forward Age Queue is full
Com:242	FWDAQ full times	Spec	Number of times the Forward Age Queue is full
Com:243	STQ collision forwardable (times)	Spec	Number of times a Store Queue collision is forwardable The following cases are not forwardable: • store address + size does not contain the load • cache-inhibited store • denormalized floating point store • store conditional • guarded load
Com:244	STQ collision forwardable (times data ready)	Spec	Number of times a Store Queue collision is forwardable and is ready with data to forward
Com:245	STQ collision forwardable (times data not ready)	Spec	Number of times a Store Queue collision is forwardable but is not ready with data to forward

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:246	STQ collision not-forwardable (times not forwardable)	Spec	Number of times a Store Queue collision is not forwardable and must wait until the store leaves the Store Queue
Com:247	STQ collision forwardable (cycles)	Spec	Number of cycles a Store Queue collision is forwardable (Number of cycles from the detection of a forwardable Store Queue entry until the load is replayed in stg1)
Com:248	STQ collision forwardable (cycles data ready)	Spec	Number of cycles a Store Queue collision is forwardable and is ready with data to forward (Number of cycles from the detection of a forwardable Store Queue entry with valid data until the load is replayed in stg1)
Com:249	STQ collision forwardable (cycles data not ready)	Spec	Number of cycles a Store Queue collision is forwardable but is not ready with data to forward (Number of cycles from the detection of a forwardable Store Queue entry without valid data until the load is replayed in stg1)
Com:250	STQ collision non-forwardable (cycles not forwardable)	Spec	Number of cycles a Store Queue collision is not forwardable and has to wait until the store leaves the Store Queue (Number of cycles from the detection of a non-forwardable Store Queue entry until the load is replayed in stg1)
Com:251	False EA (load-on store) collisions (times)	Spec	Number of times the lower 12-bits of EA matched but the upper bits did not, leading to a false load-on-store replay. Cycle penalty is 4x the number of times.
Com:252	LS0 result bus collisions	Spec	Number of LS0 result bus collisions. Cycle penalty is 3x this measurement.
Com:253	Inter-thread doubleword bank collisions	Spec	Number of inter-thread double-word bank collisions. Measures when both threads attempt to access the same double-word bank. Cycle penalty is 3x this measurement.
Com:254	Instruction L1 cache misses	Spec	Instruction L1 cache demand fetch misses (includes **icbtls**. Does not include prefetch)
Com:256	IMMU misses	Spec	Counts misses in the level 1 Instruction MMU
Com:257	IMMU TLB-4K hits	Spec	Counts hits in the level 1 Instruction MMU TLB-4K
Com:258	IMMU VSP hits	Spec	Counts hits in the level 1 Instruction MMU VSP
Com:259	IMMU cycles spent in hardware tablewalk	Spec	Counts IMMU cycles spent in hardware tablewalk. This represents the cycles from the point where the L2 MMU miss occurs to when the page table walk completes with a valid translation or exception.
Com:260	DMMU misses	Spec	Counts misses in the level 1 Data MMU (does not count replayed operations)
Com:261	DMMU TLB-4K hits	Spec	Counts hits in the level 1 Data MMU TLB-4K (does not count replayed operations)
Com:262	DMMU VSP hits	Spec	Counts hits in the level 1 Data MMU VSP (does not count replayed operations)
Com:263	DMMU cycles spent in hardware tablewalk	Spec	Counts DMMU cycles spent in hardware tablewalk. This represents the cycles from the point where the L2 MMU miss occurs to when the page table walk completes with a valid translation or exception.

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:264	L2MMU misses	Spec	Counts level 2 MMU misses. (Does not count misses that occur due to **dcbt** / **dcbtst** / **dcba** / **dcbal** instructions that fail translation and are no-oped. Does not count misses in L2MMU-VSP when looking up an indirect entry).
Com:265	L2MMU hits in L2MMU-4K	Spec	Counts level 2 MMU hits in L2MMU-4K
Com:266	L2MMU hits in L2MMU-VSP	Spec	Counts level 2 MMU hits in L2MMU-VSP (does not count indirect lookups)
Com:267	L2MMU indirect misses	Spec	Counts level 2 MMU indirect misses. This represents indirect entry lookups that do not have a matching indirect entry.
Com:268	L2MMU indirect valid misses	Spec	Counts level 2 MMU indirect valid misses. This occurs when the indirect entry is valid, but the corresponding PTE[V] = 0 or the permissions in the PTE are not sufficient for the requested access.
Com:269	LRAT misses	Spec	Counts Logical to Real Address Translation misses. This includes LRAT misses from tlbwe instructions or from page table translations.
Com:272	Cycles LMQ loses DLINK arbitration due to SGB	Spec	Cycles the Load Miss Queue loses DLINK arbitration due to the Store Gather Buffer
Com:273	Cycles SGB loses DLINK arbitration due to LMQ	Spec	Cycles the Store Gather Buffer loses DLINK arbitration due to the Load Miss Queue
Com:274	Cycles thread loses DLINK arbitration due to other thread	Spec	Cycles thread loses DLINK arbitration due to other thread
Com:278	Decode mask/value event	Nonspec	One mask/value pair that allows instructions to be counted in Decode
Com:443	DLINK request	N/A	Number of DLINK requests made from core to Shared L2
Com:444	ILINK request	N/A	Number of ILINK requests made from core to Shared L2 (includes instruction fetches and L2MMU hardware tablewalk requests)
Com:445	RLINK request	N/A	Number of RLINK requests made from Shared L2 to core (reload data)
Com:446	BLINK request	N/A	Number of BLINK requests made from Shared L2 to core (back invalidates, stashes, barriers)
Com:447	CLINK request	N/A	Number of CLINK requests made from Shared L2 to core (CoreNet data forwarding)
Com:456	L2 hits	N/A	Number of L2 Cache hits Counts 0, 1, 2, 3, or 4 per cycle
Com:457	L2 misses	N/A	Number of L2 Cache misses Counts 0, 1, 2, 3, or 4 per cycle
Com:458	L2 demand accesses	N/A	Number of L2 Cache demand accesses Counts 0, 1, 2, 3, or 4 per cycle
Com:459	L2 accesses	N/A	Number of L2 Cache accesses from all sources (demand, reload, snoop, and so on) Counts 0, 1, 2, 3, or 4 per cycle

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:460	L2 store allocates	N/A	Number of L2 Cache store allocates Counts 0, 1, 2, 3, or 4 per cycle
Com:461	L2 instruction accesses	N/A	Number of L2 Cache instruction accesses Counts 0, 1, 2, 3, or 4 per cycle
Com:462	L2 data accesses	N/A	Number of L2 Cache data accesses Counts 0, 1, 2, 3, or 4 per cycle
Com:463	L2 instruction misses	N/A	Number of L2 Cache instruction misses Counts 0, 1, 2, 3, or 4 per cycle
Com:464	L2 data misses	N/A	Number of L2 Cache data misses. Counts 0, 1, 2, 3, or 4 per cycle
Com:465	L2 hits per thread	N/A	Number of times this core/thread hits in the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:466	L2 misses per thread	N/A	Number of times this core/thread misses in the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:467	L2 demand accesses per thread	N/A	Number of times this core/thread makes a demand access to the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:468	L2 store allocates per thread	N/A	Number of times a store from this core/thread allocates in the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:469	L2 instruction accesses per thread	N/A	Number of times an instruction from this core/thread accesses the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:470	L2 data accesses per thread	N/A	Number of times a data operation from this core/thread accesses the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:471	L2 instruction misses per thread	N/A	Number of times an instruction from this core/thread misses in the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:472	L2 data misses per thread	N/A	Number of times a data operation from this core/thread misses in the L2 Cache Counts 0, 1, 2, 3, or 4 per cycle
Com:473	L2 reloads from CoreNet	N/A	Number of L2 Cache reloads from CoreNet Counts 0, 1, 2, 3, or 4 per cycle
Com:474	L2 stash requests	N/A	Number of incoming L2 Cache stash requests Counts 0, 1, 2, 3, or 4 per cycle
Com:475	L2 stash requests downgraded to snoops	N/A	Number of incoming L2 Cache stash requests downgraded to snoops Counts 0, 1, 2, 3, or 4 per cycle
Com:476	L2 snoop hits	N/A	Number of L2 Cache snoop hits Counts 0, 1, 2, 3, or 4 per cycle

Table 9-62. Performance monitor event selection (by number) (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:477	L2 snoops causing MINT	N/A	Number of L2 Cache snoops causing MINT
Com:478	L2 snoops causing SINT	N/A	Number of L2 Cache snoops causing SINT
Com:479	L2 snoop pushes	N/A	Number of L2 Cache snoop pushes.
Com:480	Stall for BIB cycles	N/A	Stall for Back Invalidate Buffer entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:482	Stall for RLT cycles	N/A	Stall for Reload Table entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:484	Stall for RLFQ cycles	N/A	Stall for Reload Fold Queue entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:486	Stall for DTQ cycles	N/A	Stall for Data Transaction Queue entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:488	Stall for COB cycles	N/A	Stall for Castout Buffer entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:490	Stall for WDB cycles	N/A	Stall for Write Data Buffer entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:492	Stall for RLDB cycles	N/A	Stall for Reload Data Buffer entry (cycles) Counts 0, 1, 2, 3, or 4 per cycle
Com:494	Stall for SNPQ cycles	N/A	Stall for Snoop Queue entry (cycles)
Com:506	BIU master requests	N/A	Master transaction starts (number of AOut sent to CoreNet)
Com:507	BIU master global requests	N/A	Master transaction starts that are global (number of AOut with M=1 sent to CoreNet)
Com:508	BIU master data-side requests	N/A	Master data-side transaction starts (number of D-side AOut sent to CoreNet)
Com:509	BIU master instruction-side requests	N/A	Master instruction-side transaction starts (number of I-side AOut sent to CoreNet)
Com:510	Stash requests	N/A	Stash request on AIn matches stash IDs for core or L2
Com:511	Snoop requests	N/A	Externally generated snoop requests (number of AIn from CoreNet not from self)

[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 e6500 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 glossary of terms used in this chapter. These definitions offer a review of commonly used terms and point out specific ways these terms are used in this chapter.

NOTE

Please read this glossary carefully. Some definitions differ slightly from those used to describe previous processors, 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 e6500 core does not use the architecture-defined hint bits in the BO operand for static prediction. Writing BUCSR[BPEN] = 0 disables dynamic branch prediction; in this case, the e6500 core predicts every branch as not taken.
- Branch resolution—the determination of whether a branch prediction is correct. If a prediction is correct, 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—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—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.
- Thread—depending on the context, denotes either a software thread of execution or one of several virtual processors within a single processor core. The e6500 core has two virtual processors that share certain resources and can simultaneously execute two software threads.
- 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—(in the context of instruction handling) occurs when a result is written into the architecture-defined registers (typically the GPRs). On the e6500 core, 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 e6500 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 e6500 core implements pipelining, superscalar instruction issue, multiple execution units that operate independently and in parallel, and two simultaneously operating thread processors.

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 non-cacheable 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.

Some instructions (including integer multiplies and divides, all floating-point and AltiVec, and loads and stores) may experience greater latencies if both threads are enabled because the resources that execute these instructions are shared by the threads and, therefore, may be a source of contention between the threads. Unlike some simultaneous multi-threading processors, the execution timing of many instructions is not affected by the simultaneous execution of two threads because each thread of the e6500 core has dedicated branch and ALU execution and completion resources.

Each thread of the e6500 core can complete as many as two instructions on each clock cycle.

The instruction pipeline stages of a thread processor in the e6500 core 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) of the thread for consideration by the decoder and dispatcher.

  The fetcher tries to initiate a fetch for all of the enabled threads in every cycle in which it is guaranteed that the IQ of each enabled thread 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 instruction cache or an L2 cache. Those factors increase when it is necessary to fetch instructions from an L3 cache or 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 10.3.2.1, "L1 and L2 TLB access times."
- 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**, **dni**, **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 of each thread.

  Each thread has five issue queues, BIQ, GIQ, LSIQ, VIQ, and FIQ. The BIQ can accept one instruction per cycle. The other issue queues can accept up to two instructions 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 both the CQ and the target issue queue for an instruction to decode and dispatch.

  In this chapter, dispatch is treated as an event at the end of the decode stage.
- 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 decode and dispatch unit per cycle. SFX0, SFX1, and CFX instructions of a thread are dispatched to and issued from the GIQ of the thread, as shown in the following figure.

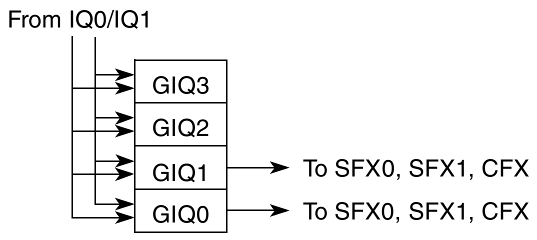


Figure 10-1. GPR Issue Queue (GIQ) (per thread)

The GIQ of each thread can hold up to four instructions.

Instructions can be issued out-of-order from GIQ1–GIQ0 to either SFX0, SFX1, and CFX.

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 either SFX need not wait for an CFX instruction in GIQ0 that is stalled behind some long-latency CFX operation.

— The LSIQ of each thread accepts as many as two instructions from the dispatch unit per cycle. LSU instructions are dispatched to the LSIQ.

  Each LSIQ can hold up to four instructions.

  Instructions are issued in-order from the LSIQ to the LSU and can issue at a rate of one per thread per cycle.

— The FIQ of each thread accepts as many as two instructions from the dispatch unit per cycle. FPU instructions are dispatched to the FIQ.

  Each 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 thread per cycle.

— The VIQ of each thread accepts as many as two instructions from the dispatch unit per cycle. AltiVec instructions are dispatched to the VIQ.

  Each VIQ can hold up to four instructions.

  Instructions can be issued out-of-order from the VIQ to the vector units from slots VIQ0 and VIQ1. Only one instruction for the units VSFX, VCFX, and VFPU can be issued per thread per cycle because these units share a reservation station. Only one instruction for the VPERM unit and its reservation station can issue per thread per cycle.

— The BIQ of each thread accepts as many as one instruction from the dispatch unit per cycle. BU instructions are dispatched to the BIQ.

  Each BIQ can hold up to three instructions.

  Instructions are issued in-order from the BIQ to the BU and can issue at a rate of one per thread per cycle.

- Execute stage—comprised of individual non-blocking execution units implemented in parallel. The CFX, floating-point, and AltiVec execution units are shared by all threads. Each execution unit has a reservation station per thread that must be available for an instruction issue for that thread to occur. Instructions are issued to the appropriate reservation station to receive their operands. If the execution unit is shared between threads, instructions in the per-thread reservation stations of that execution unit that are ready to execute arbitrate for execution.

  The e6500 core has the following execution units:

  — Branch unit (BU)—executes branches and CR logical operations. There is one independent BU per thread.

  — Two simple units (SFX0 and SFX1)—execute logical instructions and all integer computational instructions except multiply and divide instructions. Each thread has its own independent SFX0 and SFX1.

  - SFX0 executes all integer simple unit instructions (that is, all that can be dispatched to simple units) and a few common move to/from special purpose register (SPR) instructions.
  - SFX1 executes most, but not all, of the instructions that can be executed in SFX0.

  Most SFX instructions execute in one cycle. However, some instructions can take longer than one cycle.
- Complex unit (CFX)—executes integer multiplication and division instructions, as well as most move to/from special purpose register (SPR) and all move to/from thread management register (TMR) instructions. A single CFX is shared by all threads.
- Floating-point unit (FPU)—executes FPR-based floating point computational instructions. Floating-point load and store instructions execute in the LSU. A single FPU is shared by all threads.
- 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.
- Vector unit, consisting of 4 sub-units (VSFX, VCFX, VFPU, and VPERM)—executes AltiVec instructions. VPERM executes permute instructions; VFPU executes floating-point instructions; VCFX executes multiply instructions; VSFX executes other integer arithmetic and logical instructions. A single vector unit is shared by all threads.

An 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 one-cycle latency, so results of these instructions are available one clock cycle after an instruction enters the execution unit. SFX0, LSU, FPU, VSFX, VCFX, VFPU, VPERM and CFX are fully pipelined for continuous multi-cycle execution.

- Complete and Write-Back stages—maintain the correct architectural machine state and commit results to the architecture-defined registers in the proper order for each thread. If completion logic detects an instruction containing an exception status or a mispredicted branch, all following instructions of that thread are cancelled, their execution results in rename registers are discarded, and the correct instruction stream for the thread is fetched.

  The Complete stage ends when the instruction is retired. If no dependencies or exceptions exist, as many as two instructions per thread are retired in program order per clock cycle. The Write-Back stage occurs in the clock cycle after the instruction is retired and updates the architectural state.

10.3 General timing considerations

As many as four instructions can be fetched to the IQ of one thread during each clock cycle. These instructions are known as a *fetch group*. The starting quadword-bounded address of this fetch group is the *fetch group address* (FGA). Two instructions per thread per clock cycle can be dispatched to the issue queues. Two instructions from the GIQ and one instruction from the BIQ can issue per thread per clock cycle to the appropriate execution units. One instruction from the LSIQs of both threads can issue to the LSU per cycle. One instruction from one thread's FIQ can issue to the FPU per clock cycle. A total of two instructions from the VIQs of either thread can issue to the AltiVec execution units per cycle. Thus, the total two-thread maximum issue width per clock cycle of the e6500 core is eleven instructions. Two instructions can retire and two can write back per thread per cycle.

The e6500 core 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, VR, 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 history table (BHT) and the branch target buffer (BTB). Predictions are resolved in the branch unit (BU). Incorrect predictions of branch direction or target address are handled as follows:

1. Fetch is redirected to the correct path, and instructions on the mispredicted path 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.
4. The branch predictor is updated as appropriate.

After an instruction executes, results are made available to subsequent instructions in the appropriate rename registers. 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, FPR, or VR during the write-back stage. Branch instructions that update either the LR or CTR write back their results in a similar fashion.

Section 10.3.1, "General instruction flow," describes this process.

10.3.1 General instruction flow

The e6500 core ignores static branch prediction hints: *a* and *t* bits in the BO field in branch encodings are ignored.

Dynamic branch prediction for a thread is enabled by writing 1 to BUCSR[BPEN]. Writing 0 to a thread's BUCSR[BPEN] disables dynamic branch prediction for that thread only, in which case the e6500 core predicts every branch as not taken for that thread.

To resolve branch instructions and improve the accuracy of branch predictions, each thread of the e6500 implements a dynamic branch prediction mechanism using a 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 also holds a 2-bit branch local history whose value depends on whether the branch was taken or not in its two most recent occurrences. The local history bits from the BTB are used to select among four 2-bit global history saturating counters obtained from a 512-entry pattern history table (PHT) that is indexed by an XOR of a global taken/not-taken history and the previous fetch address. The PHT bits can take four values:

- Strongly taken
- Weakly taken

- Weakly not taken
- Strongly not taken.

This prediction mechanism is described in Section 10.4.1.2, “Branch prediction and resolution.”

Each thread of the e6500 core also predicts the upper 32 bits of the branch address using a Segment Target Address Cache (STAC) and a Segment Target Index Cache (STIC). These are used in concert with the BTB to construct 64-bit branch targets and predict long branches (branches that cross 4 GB segments). The prediction of long branches is enabled by setting BUCSR[STAC_EN]. The STAC and STIC structures are more fully described in Section 10.4.1.2.3, “Segment Target Address Cache (STAC), Segment Target Index Cache (STIC), and link stack.”

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 an instruction in IQ0.

10.3.2 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 10.3.2.1, “L1 and L2 TLB access times.”
- If a page translation is not in a TLB, the hardware may perform a tablewalk operation to obtain the correct page table entry or an instruction TLB miss interrupt may be taken. Section 10.3.2.2, “Interrupts associated with instruction fetching,” describes other conditions that cause an instruction fetch to take an interrupt.
- If an L1 instruction cache miss occurs, an L2 cache transaction is performed in which fetch latency is affected by traffic from other e6500 cores that share the same L2 cache and other system traffic. If 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 10.3.2.3, “Cache-related latency.”

10.3.2.1 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 six clocks; for L1 data address translation misses, the L2 TLB latency is at least six clocks. These access times may be longer, depending on arbitration between L2 TLB accesses for simultaneous instruction L1 TLB misses and/or data L1 TLB misses from either thread, the execution of TLB instructions or a hardware tablewalk operation by either thread, and TLB snoop operations (snooping of TLB invalidate operations from **tlbivax** instructions on CoreNet).

If the page translation is in neither TLB, a hardware tablewalk is initiated to obtain the appropriate page table entry. If the hardware tablewalk fails, an instruction TLB error interrupt occurs, as described in Section 4.9.15, "Instruction TLB error interrupt—IVOR14/GIVOR14."

The L2 TLB is shared by all threads in the e6500 core. If one thread misses in both its L1 TLB and the shared L2 TLB, the other threads can continue to access both their own L1 TLBs and the shared L2 TLB. Handling of L2 TLB misses from multiple threads via the hardware tablewalk mechanism are not pipelined.

When a TLB invalidate operation is detected, the L2 MMU becomes 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.

TLBs are described in detail in Chapter 6, "Memory Management Units (MMUs)."

10.3.2.2 Interrupts associated with instruction fetching

An instruction fetch can generate the following interrupts:

- An instruction TLB error interrupt occurs when both the effective address translation for a fetch is not found in the TLBs and the hardware table walk to obtain the translation fails for any reason. 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 MSR[PR] and occurs regardless of whether the thread is in guest state.
    - 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 MSR[PR] and occurs regardless of whether the thread is in guest state.

  When an instruction storage interrupt occurs, the thread suppresses execution of the instruction causing the exception. For more information, see Section 4.9.5, "Instruction storage interrupt (ISI)—IVOR3/GIVOR3."

10.3.2.3 Cache-related latency

Instruction fetch latencies depend on various instruction cache related factors:

- If the fetch hits in the instruction cache or the Instruction Line Fill Buffer (ILFB) or the thread fetch buffer, as many as four instructions will be added to the IQ of the thread two clock cycles after the cache request is made.
- The instruction 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. If a cache miss is discarded by a misprediction and a subsequent fetch hits,

the cache provides instructions to the IQ of the requesting thread. As many as four instructions can be fetched by all threads every cycle from their private fetch buffers, from the the ILFB, or from the instruction cache.

- 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. For example, if a thread cannot fetch from either its fetch buffer or the ILFB, but a cache reload is being performed in the same cycle it would otherwise be granted access to the instruction cache, the fetch of that thread will be blocked for 1 cycle.
- If an instruction fetch misses in the instruction cache, it is requested from the shared L2 cache. If it misses in the shared backside L2 cache, the L2 cache initiates a CoreNet transaction and allocates the line when it is returned from the off-core memory system.

The architecture defines WIM (of WIMGE) bits that define caching characteristics for the corresponding memory page. Fetching an instruction as caching-inhibited (I = 1) produce the following actions:

- The ILFB may hit, and the instructions returned from the ILFB are used, even if the ILFB entry was established by an earlier cacheable access.
- The instruction cache performs an access and may hit, and, if a hit occurs, the instructions are used.
- The L2 cache does 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 returns and uses the entire fetch group that was requested. Therefore, fetching with caching-inhibited accesses does not produce a bus transaction for each instruction; it produces one bus transaction for each fetch group or cache line.

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 caching-inhibited addresses.

10.3.3 Dispatch, issue, and completion considerations

All threads can simultaneously dispatch as many as two instructions per cycle, depending on the mix of instructions and on the availability of issue queues and CQ entries. As many as two instructions per thread can be dispatched in parallel, but an instruction in IQ1 cannot be dispatched ahead of an instruction in IQ0 of one thread.

Instructions can issue out-of-order from GIQ0 and GIQ1 to either the SFX0 or SFX1 associated with each thread or the CFX that is shared by all threads. If an instruction stalls in GIQ0 (reservation station busy), an instruction in GIQ1 can issue if the reservation station of its target execution unit is available.

Instructions can simultaneously issue in-order from the LSIQ of every thread.

One instruction can issue in-order from the FIQ of one selected thread per cycle.

Instructions can issue out-of-order from the VIQ of each thread. The VIQ of each thread can issue to any of the VSFX, VCFX, or VFPU at one per cycle and can also issue to VPERM at one per cycle.

Issue queues and reservation stations allow e6500 threads to dispatch instructions even if their private or shared 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 on which the instruction is dependent.

The CQ of each thread maintains program order after instructions are dispatched, guaranteeing per-thread in-order completion and a precise exception model. Thread instruction state and other information required for completion are kept in this 16-entry FIFO. All instructions in a thread complete in order; none can retire ahead of a previous instruction. In-order completion ensures the correct architectural state when an e6500 thread must recover from a mispredicted branch or exception.

Each thread can simultaneously retire as many as two instructions, never out of order. Note the following:

- Instructions must be non-speculative in order to complete.
- As many as two rename registers per thread 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 instruction is assigned CR and GPR renames at dispatch.
- Some instructions have retirement restrictions, such as retiring only out of CQ0. See Section 10.3.3.1, "Instruction serialization."

Program-related exceptions are signaled when the instruction causing the exception reaches CQ0. Previous instructions in the thread are allowed to complete before the exception is taken, which ensures that any exceptions those instructions may cause are taken. One thread taking an exception generally does not affect instruction processing by other threads.

10.3.3.1 Instruction serialization

Although each e6500 thread can dispatch and complete two instructions per cycle, some serializing instructions limit dispatch and completion to one per thread per cycle. There are seven basic types of instruction serialization:

- Presync serialization—Presync-serialized instructions are held in the instruction queue of the thread until all prior instructions of the thread have completed before they are allowed to be decoded and executed. For example, an **mfspr** instruction that reads a non-renamed status register is marked as presync-serialized.
- Postsync serialization—Postsync-serialized instructions, such as **mtspr**[XER], prevent subsequent instructions in a thread from decoding until the serialized instruction completes. For example, instructions that modify the thread state in a way that affects the handling of future instruction execution by the thread 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 of the thread have completed. If the instruction is currently in the reservation station and is the oldest instruction in the thread, 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 thread architectural state until all older instructions that could affect the architectural state of the thread 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 in the thread. 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 in a thread after the completion of the instruction. Refetch serialization is used when an instruction has changed or may change a particular context needed by subsequent instructions in the thread. Examples include **isync**, **sc**, **rfi**, **rfci**, **rfmci**, **rfdi**, **rfgi**, **wait**, and any instruction that causes the summary-overflow (SO) bit to change state.
- Store serialization (applicable to stores and some LSU instructions that access the data cache)—Store-serialized instructions are dispatched and held in the finished store queue of the thread. 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 thread 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 block the reservation station for the associated execution unit. This prevents other instructions from issuing to the reservation station while the unit-serialized instruction executes. Normally, such instructions modify the architectural state of a renamed register, and the serialization ensures that no other instruction in the thread accesses the renamed register when the unit-serialized instruction executes.

10.3.4 Memory synchronization timing considerations

This section describes the behavior of the **sync** and **mbar** instructions as they are implemented on the e6500 core.

10.3.4.1 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, **sync** also provides an ordering boundary for pre- and post-**sync** storage transactions.

On the e6500 core, **sync** waits for all preceding data memory accesses of the thread 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 in its thread complete entirely in cache memories and externally on the bus (address and data complete on the bus, excluding instruction fetches). No subsequent instructions and associated storage transactions are initiated by the thread until such completion.

Execution of **sync** by a thread 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 e6500 core complete in their queues or at their destinations before completing the SYNC command on the CoreNet interface.

10.3.4.2 mbar instruction timing considerations

The **mbar** instruction provides an ordering boundary for storage operations within a thread. Its architectural intent is to guarantee that storage operations resulting from previous instructions of the thread occur before any subsequent storage operations occur, thereby, ensuring an order between pre- and post-**mbar** memory operations. The **mbar** instruction 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 e6500 core supports two classes for system flexibility.

The e6500 threads implement the following two variations of **mbar**:

- When MO = 0, **mbar** behaves as defined by the Power ISA, which, on the e6500 core (for all practical purposes), produces the same memory barrier as **sync**.
- When MO = 1, **mbar** is a weaker, faster memory barrier; the e6500 core 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 of a thread complete before any data accesses caused by subsequent instructions of the same thread. 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 on the e6500 core.

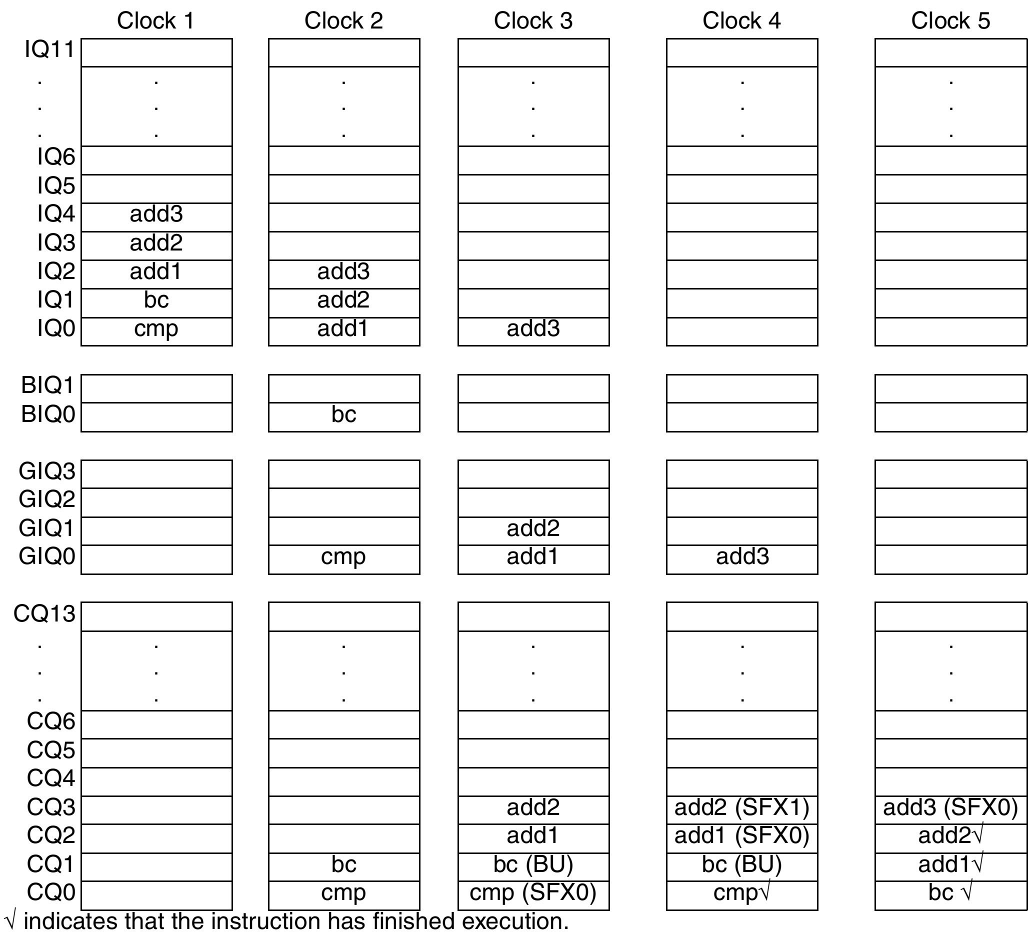
10.4.1 Branch execution unit

When branch or trap instructions change the program flow of a thread, its IQ must be reloaded with the target instruction stream. Previously issued instructions of the thread 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 e6500 core minimizes the penalties associated with flow control operations by employing dynamic alloyed global and local history-based branch prediction, speculative link and counter registers, and non-blocking caches.

10.4.1.1 Branch instructions and completion

Branch instructions are not folded on an e6500 thread. 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 of the thread and are assigned a slot in the CQ of the thread, as shown in the following figure.



√ indicates that the instruction has finished execution.

Figure 10-2. Branch completion (LR/CTR write-back) for one thread

In this example, the **bc** depends on **cmp** and is predicted as not taken. At the end of clock 1, **cmp** and **bc** are dispatched to the GIQ and BIQ of the thread, respectively, and are issued to SFX0 and the BU at the end of clock 2.

In clock 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 clock 4, the **bc** resolves as correctly predicted; add1 and add2 are issued to the SFX0 and SFX1 and are marked as nonspeculative, and add3 is dispatched to the GIQ. The **cmp** is retired from the CQ at the end of clock 4.

In clock 5, **bc**, add1, and add2 finish execution, and **bc** and add1 retire.

10.4.1.2 Branch prediction and resolution

Each e6500 thread has an independent dynamic branch prediction mechanism that monitors and records branch instruction behavior within the thread, from which the outcome of the next occurrence of a branch instruction is predicted. If branch prediction for a thread is disabled, all branches for that thread are predicted as not taken. The e6500 core does not support static branch prediction—the BO static prediction field in branch instructions is ignored. The branch prediction mechanism for each thread includes a branch target buffer (BTB), pattern history table (PHT), segment target address cache (STAC), segment target index cache (STIC), and a link stack.

The e6500 branch prediction mechanism uses the current fetch group address to detect whether the fetch group includes any branches in the BTB. If a branch is in the BTB, the branch outcome and target fetch group address are predicted. Local history or an unconditional flag in the BTB entry for the branch is combined with global history from the PHT to generate the taken or not taken prediction. If the branch is predicted as taken, information in the BTB selects a target address from that in the BTB entry or on the top of the link stack. If the link stack is empty, the target address is always taken from the BTB entry. If indicated by the BTB entry, the upper 32 bits of the target address are generated by the STIC and STAC. Also, if indicated by the BTB entry, the address of the instruction after the branch is pushed onto the link stack.

When a branch instruction first enters the instruction pipeline, it is not allocated in the BTB. By default, the branch instruction is predicted as not taken. If the branch is resolved as not taken, nothing is allocated in the BTB. If the branch is resolved as taken, the misprediction allocates both a BTB entry for this branch with an initial local history of taken/taken and sets the taken/taken field of the appropriate PHT entry to strongly-taken. Note that unconditional branches are allocated in the BTB the first time they are encountered.

Instructions after an unresolved branch within a thread can execute speculatively, but per thread in-order completion ensures that mispredicted speculative instructions do not complete. If either the outcome or target address of a branch are incorrectly predicted, instructions in a thread dispatched after a mispredicted branch instruction are flushed from the thread CQ, and any speculative results are flushed from the rename registers. Instructions in the thread that preceded the predicted branch are not flushed and proceed normally through the pipeline. After a misprediction, instruction fetching is redirected to the correct path, and the branch predictor contents are revised by either invalidating a phantom branch entry or updating the PHT, local history and overwriting an incorrect target address of the existing branch predictor entry.

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 thread pipeline (the branch execute unit, the BIQ, and the IQ). The presence of speculative branches allocated in the BTB slightly reduces speculation depth.

10.4.1.2.1 Branch predictor structure and operation

The BTB for each thread is a 512 entry, four-way associative structure that contains the types, local taken/not-taken histories, and target addresses of previously taken branch instructions. Each BTB entry also has a valid bit that is set to zero (invalid) upon reset of the core. A valid BTB entry is selected based on the low-order 32 bits of the current fetch group address. The BTB is indexed by bits 53-59 from the fetch group address, and the tag is compared against bits 32-52 concatenated with bits 60-61 of the current fetch group address.

The BTB allocates an entry when both a branch has been resolved as taken and no BTB entry already exists for the fetch group address in which the branch instruction was encountered. Note that it is possible for the same branch instruction to have multiple entries in the BTB if the branch is encountered by different fetch group addresses. When a new entry is allocated, the following information is stored in a BTB entry to describe the branch:

- target—30 bits, set to bits 32:61 of the target address to where the branch goes.
- history—2 bits, set to 0b11 to indicate taken/taken, used to select among PHT entries.
- iab—3 bits, set to indicate where in the fetch group the instruction following the branch is located (that is, the instruction to be executed if the branch is not taken).
- long—1 bit, set if the branch is a 'long' branch (see Section 10.4.1.2.3, "Segment Target Address Cache (STAC), Segment Target Index Cache (STIC), and link stack") Note: this bit was used as a lock bit for e500v1 and e500v2.
- ls_push—1 bit, set to indicate the current contents of the LR should be pushed onto the link stack.
- ls_pop—1 bit, set to indicate the branch target address should be taken from the top of the link stack.
- valid—1 bit, set to 1 to indicate that the entry is valid.
- always—1 bit, set to 1 to indicate an unconditional branch.

When allocating an entry in the BTB, a 6-bit LRU value (lru) keeps a relative use between the ways encoded. The meaning of the lru bits is as follows:

- lru[0]—If set, indicates that way 0 is more recently used than way 1.
- lru[1]—If set, indicates that way 0 is more recently used than way 2.
- lru[2]—If set, indicates that way 0 is more recently used than way 3.
- lru[3]—If set, indicates that way 1 is more recently used than way 2.
- lru[4]—If set, indicates that way 1 is more recently used than way 3.
- lru[5]—If set, indicates that way 2 is more recently used than way 3.

In addition to the BTB entry, the STIC and STAC structures are updated if the newly allocated branch is a 'long' branch:

- A fully associative search is performed on the STAC to determine if the upper 32 bits of the branch target address is already present. If found, the STIC is updated to point to the STAC entry that was found. If no matching entry in the STAC is found:
  - A STAC entry is replaced using a FIFO method to select an entry for replacement.
  - The STAC entry is set to the upper 32 bits of the target address.

— The STIC is updated to point to the STAC entry.

- The updated STIC entry is indexed by bits 53-59 of the fetch group address.

The allocation of a function call or return branch causes the link stack to be updated. A branch is considered a function call if it sets the LR of the thread as a result of the branch but is neither a **bclrl**, which is treated as a return, nor a specific branch form that is used to get the current instruction address for position independent code (for example, **bcl** 20,31,$+4). A branch is considered a function return if it branches to the contents of the LR.

A function call branch causes a link stack entry to be "pushed". The following occurs:

- ls_push is set in the BTB entry to indicate that this branch is a function call.
- The address of the branch instruction + 4 (that is, where the function call will return to) is pushed onto the top of the link stack. If the link stack overflows, its oldest entry is discarded.

When the branch is a function return, the link stack has an entry "popped". The following occurs:

- ls_pop is set in the BTB entry to indicate that this branch is a function call return.
- If the link stack is not empty, the top of the link stack is "popped" (that is, removed from the link stack), and that address is used as bits 0-61 of the branch target. (Bits 62-63 of any instruction address are always 0.)

10.4.1.2.2 Global History (GHR) register and Pattern History Table (PHT)

The direction (taken or not-taken) of the nine most recent branches is stored in the GHR. In the GHR, a taken branch is represented as binary 1. The GHR shifts from the right or least significant bit positions to the left so that the direction (taken or not-taken) of the most recent branch is indicated by the rightmost or least significant bit.

The PHT is indexed by the GHR XORed with bits 53-61 of current FGA to select one of 512 entries. Each entry contains four local-history two-bit saturating counters, one each for the four possible local history values contained in the selected BTB entry. The most significant bit of each counter indicates whether the branch should be taken or not. The least significant bit of the counter indicates the strength of this prediction. A counter is incremented when the branch is resolved as taken or decremented if the branch is resolved as not-taken. To speed branch predictor learning, upon invalidation of the PHT, the default values of the counters in all entries are:

— Local not-taken/not-taken: weakly not-taken

— Local not-taken/taken: weakly taken

— Local taken/not-taken: weakly taken

— Local taken/taken: strongly taken

10.4.1.2.3 Segment Target Address Cache (STAC), Segment Target Index Cache (STIC), and link stack

The BTB contains a bit for each entry that indicates whether the branch is a 'long' branch or not. A long branch is a branch that occurs to an address for which the upper 32 bits of the branch target differ from the upper 32 bits of the address of the branch instruction. The STAC and STIC structures are used to predict the upper 32 bits of the target address.

The STAC structure is an eight-entry array in which each entry contains the upper 32 bits of the target address. Entries are allocated in a FIFO manner, but a new entry is only allocated if the unique 32-bit value does not already exist in the array. When a branch is taken and considered long, a fully associative lookup of the STAC structure is performed to determine if the upper 32 bit target already exists. If it does, the STIC entry associated with the branch is updated to point to the found STAC entry.

The STIC is a 128-entry array for which each 3-bit entry points to a STAC entry. The STIC entry is indexed by bits 55-61 of the fetch group address.

There are no valid bits for either the STIC or STAC.

The link stack is used to predict function call and function call return branches. The link stack is an eight-entry structure that contains the entire 62-bit branch target address (bits 0-61). Note that bits 62-63 are always zero because branch targets on the e6500 core are always word aligned. In addition, the ls_push and ls_pop bits in the BTB entry are used to denote whether the target of the branch was due to a function call (ls_push) or a function return (ls_pop).

On a taken branch for which ls_push is set, NIA+4 (the instruction sequentially after the taken branch) is pushed onto the link stack (this will be the target of the function return). On a taken branch for which ls_pop is set and the link stack is not empty, the top of the linked stack is popped, and that address is used as the predicted target address.

On a taken branch for which ls_pop is set and the link stack is empty, the target address is taken from the BTB entry, essentially providing an additional link stack entry and handling deeply recursive function call paths.

10.4.1.2.4 Branch predictor operations controlled by BUCSR

The following branch predictor operations are controlled through BUCSR:

- Branch prediction disabling—BUCSR[BPEN] is used to enable or disable the branch predictor of a thread. The branch predictor is enabled when the bit is set and disabled when it is cleared.When it is disabled, the branch predictor is not used to predict branch outcomes or targets, and the branch predictor is not updated as a result of executing branch. All branch prediction is disabled, including predictions of the upper 32 bits of the targets address and any function calls and returns.
- BTB and PHT invalidation—Flash invalidation of the BTB and PHT for a thread are accomplished by writing BUCSR[BBFI] with a 0 and then a 1 using the **mtspr** instructions. Invalidation of the PHT causes the default values of the counters in all entries to be set as follows:
  — Local not-taken/not-taken: weakly not-taken
  — Local not-taken/taken: weakly taken
  — Local taken/not-taken: weakly taken
  — Local taken/taken: strongly taken
- Prediction of upper 32 bits of branch target—If BUCSR[STAC_EN] of a thread is set, then the upper 32 bits of the branch target address are predicted. STIC and STAC structures are used to predict the upper 32 bits of the target address. If BUCSR[BPEN] is not 1, then BUCSR[STAC_EN] is ignored and all branch prediction is disabled.

- Function call and return branches—If BUCSR[LS_EN] of a thread is set, then function call and return branches are predicted using the link stack. If BUCSR[BPEN] is not 1, then BUCSR[LS_EN] is ignored and all branch prediction is disabled.

10.4.1.2.5 Branch prediction special cases: multiple matches and phantom branches

The e6500 branch prediction hardware for each thread prevents multiple matches for the same fetch address by ensuring that an entry is unique when it is allocated.

Because the BTB and link stack hold effective addresses (not real or physical addresses) associated with branch instructions, a phantom branch can occur when a process context switch brings an MMU translation that maps a non-branch instruction in the current context to the same effective address as a branch instruction for which the BTB has a valid entry. If the phantom branch is predicted taken, instruction fetching is redirected to the predicted target address. Later, during execution of the instruction, the hardware detects the prediction error and invalidates the BTB entry.

10.4.1.3 Changing LR and CTR in branch instructions

When a branch instruction that sets the LR and decrements the CTR (for example, **bdnzl**) is executed, the instruction is processed as two micro-ops. One micro-op writes the LR, and one micro-op decrements and writes the CTR. The micro-op that decrements the CTR is sent to either of the SFX units, and the rest of the branch execution and LR update is executed by the branch unit.

10.4.2 Complex and simple unit execution

The e6500 core has one complex unit (CFX) shared by all threads, and each thread has two simple units (SFX0, SFX1).

The CFX executes multiplies, divides, and move to and from special registers (including PMRs and TMRs) other than LR and CTR.

SFX0 and SFX1 execute most logical and integer computational instructions except multiplies and divides. SFX0 also executes certain integer computational instructions (such as, **popcnt***x*, **bpermd, cntlz***x*) and executes move to and from special registers LR and CTR.

10.4.2.1 CFX divide execution

Divide latency depends upon the operand data and the size (32- or 64-bit) of the divide and ranges from 4 to 26 cycles, as shown in the following table.

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

	For all other cases, the latency is a function of the number of predicted significant result bits. The number of predicted significant result bits (PBITS) is computed using the number of significant dividend bits (NUMBITS) and the number of significant divisor bits (DENOMBITS). The number of significant bits is the operand size minus a count of the leading zero bits of the absolute value of the operand.
	
	Let LZB32(*x*) = number of leading zero bits of *x* starting at bit 32; ABS(*x*) = absolute value of register *x* contents
	
	NUMBITS = 32 - LZB32(ABS(**r**A))
	DENOMBITS = 32 - LZB32(ABS(**r**B))
	PBITS = NUMBITS - DENOMBITS + 1
		5 + ((PBITS + 1) / 3)
		
		<u>Example:</u>
		rA is 0x0f000 (16 significant bits),
		rB is 0x07 (3 significant bits),
		PBITS = 16 - 3 + 1 = 14
		cycles = 5 + ((14 + 1) / 3) = 10
		
		Thus:
		0x0f000 divided by 0x07 takes 10 cycles
		
		<u>Max latency:</u>
		rA has 32 significant bits (unsigned divide),
		rB has 1 significant bit,
		PBITS = 32 -1 + 1 = 32
		cycles = 5 + ((32 + 1) / 3) = 16

divd*x*, **divdu***x*	**r**A or **r**B is 0, or **r**A < **r**B	4

	For all other cases, the latency is a function of the number of predicted significant result bits. The number of predicted significant result bits (PBITS) is computed using the number of significant dividend bits (NUMBITS) and the number of significant divisor bits (DENOMBITS). The number of significant bits is the operand size minus a count of the leading zero bits of the absolute value of the operand.
	
	Let LZB(*x*) = number of leading zero bits of *x* starting at bit 0; ABS(*x*) = absolute value of register *x* contents
	
	NUMBITS = 64 - LZB(ABS(**r**A))
	DENOMBITS = 64 - LZB(ABS(**r**B))
	PBITS = NUMBITS - DENOMBITS + 1
		5 + ((PBITS + 1) / 3)
		
		<u>Example:</u>
		rA is 0x0FF_0000_F000 (38 significant bits),
		rB is 0x07 (3 significant bits),
		PBITS = 38 - 3 + 1 = 36
		cycles = 5 + ((36 + 1) / 3) = 17
		
		Thus:
		0x0FF_0000_F000 divided by 0x07 takes 17 cycles
		
		<u>Max latency:</u>
		rA has 64 significant bits (unsigned divide),
		rB has 1 significant bit,
		PBITS = 64 -1 + 1 = 64
		cycles = 5 + ((64 + 1) / 3) = 26

10.4.2.2 CFX multiply execution

Multiply latency depends on the operand values being multiplied and the type of operation (word or doubleword). The minimum latency for a multiply is four cycles and the maximum latency is seven cycles.

Any multiply instruction that uses 32-bit operands, such as **mullw***x* and **mulhw***x*, is fully pipelined and has a four-cycle latency with a repeat rate of one. These instructions execute with this latency and repeat rate, regardless of the operand values.

Multiply instructions that use 64-bit operands, such as **mulld***x*, **mulli**, and **mulhd***x*, depend on the operand values to determine latency and repeat rate.

This table 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*	For all multiply instruction that use 32-bit operands, the latency is not a function of the values of the operands.	4 cycles, repeat rate of 1
mulli	For multiply instructions that use at least one 64-bit operand, the latency is a function of whether the operand has 31 or less significant bits and the sign of the operand. The number of significant bits of an operand is the operand size minus a count of the leading zero bits of the absolute value of the operand. Let LZB(*x*) = number of leading zero bits of *x* starting at bit 0; A_POS_32 = LZB(**r**A) > 32 (**r**A is a positive value with 31 or less significant bits); A_NEG_32 = LZB(~**r**A) > 32 (**r**A is a negative value with 31 or less significant bits); A_64 = ~A_POS_32 & ~A_NEG_32	When **r**A is larger than 31 significant digits (A_64 = 1): 5 cycles, repeat rate of 2 When **r**A is 31 significant digits or less and **r**A is positive (A_POS_32 = 1): 4 cycles, repeat rate of 2 When **r**A is 31 significant digits or less and **r**A is negative (A_NEG_32 = 1): 5 cycles, repeat rate of 2
mulld*x*, **mulhd***x*	For multiply instructions that use 64-bit operands, the latency is a function of whether the operand has 31 or less significant bits and the sign of the operand. The number of significant bits of an operand is the operand size minus a count of the leading zero bits of the absolute value of the operand. Let LZB(*x*) = number of leading zero bits of *x* starting at bit 0; A_POS_32 = LZB(**r**A) > 32 (**r**A is a positive value with 31 or less significant bits); A_NEG_32 = LZB(~**r**A) > 32 (**r**A is a negative value with 31 or less significant bits); A_64 = ~A_POS_32 & ~A_NEG_32 B_POS_32 = LZB(**r**B) > 32 (**r**B is a positive value with 31 or less significant bits); B_NEG_32 = LZB(~**r**B) > 32 (**r**B is a negative value with 31 or less significant bits); B_64 = ~B_POS_32 & ~B_NEG_32	When rB is larger than 31 significant digits (B_64 = 1): 7 cycles, repeat rate of 4 When **r**A is larger than 31 significant digits (A_64 = 1) and rB is 31 significant digits or less (B_64 = 0): 5 cycles, repeat rate of 2 When **r**A is larger than 31 significant digits (A_64 = 1) and rB is 31 significant digits or less (B_64 = 0): 5 cycles, repeat rate of 2 When **r**A is 31 significant digits or less and **r**A is positive (A_POS_32 = 1) and rB is 31 significant digits or less (B_64 = 0): 4 cycles, repeat rate of 2 When **r**A is 31 significant digits or less and **r**A is negative (A_NEG_32 = 1) and rB is 31 significant digits or less (B_64 = 0): 5 cycles, repeat rate of 2

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 multiply (or divide) instruction cannot start execution if another multiply is executing and is three cycles or less from finishing execution. This affects the repeat rate of multiply instructions that use doubleword operands. This is because the currently executing multiply instruction is using the reservation station operands until it is 3 cycles or less from finishing execution.

A new divide instruction cannot start execution if another divide is executing.

A new multiply may be stalled for one cycle from starting execution if a divide is executing and the divide is within the window of when the divide result and the multiply result could possibly arrive at the same cycle. This scheduling helps avoid the condition where the results of both a multiply and divide are available in the same cycle, in which case the divide is stalled.

For the e6500 core, the CFX is also used to execute other instructions. These instructions are prevented from beginning execution if a multiply or divide instruction requires the result bus on the same cycle as the other instruction to write its results. These instructions are the set of all other instructions that are executed by CFX, except for multiply and divide instructions. These instructions include move to/from SPRs, **tlbre**, **tlbwe**, **tlbsx**, **mtpmr**, **mfpmr**, **mtmsr**, **mfmsr**, and others.

10.4.3 AltiVec (vector) execution

AltiVec execution units are shared by all threads and operate on inputs from vector registers (VRs) and produce VR outputs. Four execution units comprise the AltiVec execution complex, each unit performing different classes of vector execution. The units are:

- VSFX—used for performing simple, one-cycle operations, mostly involving integer vector arithmetic or logical operations. CR results for record form of compare instructions (“.” forms) take two cycles of latency for the CR results to be visible at the branch unit of the respective thread.
- VFPU—used for performing vector single-precision floating-point operations, usually in six cycles fully pipelined.
- VCFX—used for performing more complex integer vector operations involving datapaths that span vector element data paths (such as “sum-across” type operations). These generally execute in four cycles.
- VPERM—used for performing vector permute, merge, pack, and unpack operations, which also span vector element datapaths. These generally execute in two cycles.

AltiVec instructions in each thread are dispatched to a Vector Issue Queue (VIQ), up to two per cycle. Instructions can be issued out of order from the bottom two slots of each thread VIQ to one of two reservation stations for AltiVec instructions. The corresponding reservation stations of all threads arbitrate for their respective execution units. For a given thread the VSFX, VFPU, and VCFX units share a reservation station, and the VPERM unit has its own reservation station. Thus, AltiVec instructions can issue an instruction from VIQ slot 0 to either VPERM reservation station or VSFX, VFPU, VCFX reservation station and from VIQ slot 1 to the other reservation station (not issued from slot 0).

10.4.4 Load/store execution

Each thread has an LSU that executes instructions that move data between one of the register files (for example, GPR, FPR, VPR) and the memory unit of the core composed of the L1 data cache, which is shared by all threads and the interfaces to the L2 cache and bus interface units, which are shared by multiple cores.

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 and bank collisions between threads in the L1 cache, execute in a speculative manner with a maximum throughput of one instruction per thread per cycle with a three-cycle latency. Data returned from the cache is held in a rename register until the completion logic of the thread commits the value to the thread state.

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.

AltiVec and floating-point load instructions take a fourth cycle for data reordering.

Stores cannot be executed speculatively and must be held in the store queue until they are known to be non-speculative and can be committed, at which point the data cache array is updated.

10.4.4.1 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 *EREF*. AltiVec loads and stores are always naturally aligned when accessing the memory subsystem.

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.

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

Table 10-3. Performance effects of operand placement in memory (continued)

Operand		Boundary Crossing[1]		
1 byte	1	Optimal	—	—
lmw, **stmw**	4 <4	Good Poor	Good Poor	Good Poor

[1] “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.

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 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 because 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. Please note the following:

- 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 (for example, XER[OV] or the CR result of a “.” instruction), may take one 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 that is dispatched to the unit can begin execution. For example, an instruction with a latency of three and a repeat rate of one means that, even though it takes three 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 10.3.3.1, “Instruction serialization”.

Table 10-4. e6500 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	—	2	2	
addco.	SFX0, SFX1	—	2	2	
adde	SFX0, SFX1	—	1	1	
adde.	SFX0, SFX1	—	1	1	
addeo	SFX0, SFX1	—	2	2	
addeo.	SFX0, SFX1	—	2	2	
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	—	2	2	
addmeo.	SFX0, SFX1	—	2	2	
addo	SFX0, SFX1	—	2	2	
addo.	SFX0, SFX1	—	2	2	
addze	SFX0, SFX1	—	1	1	
addze.	SFX0, SFX1	—	1	1	
addzeo	SFX0, SFX1	—	2	2	
addzeo.	SFX0, SFX1	—	2	2	
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	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
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	
bpermd	SFX0	—	1	1	
cmp	SFX0, SFX1	—	1	1 or 2	EQ bit is 1 cycle to branch unit; other results are 2 cycles.
cmpb	SFX0, SFX1	—	1	1	
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.
cntlzd	SFX0	—	1	1	
cntlzd.	SFX0	—	1	1	
cntlzw	SFX0	—	1	1	
cntlzw.	SFX0	—	1	1	
crand	BU	—	1	1	
crandc	BU	—	1	1	
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	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
dcbfep	LSU	Store	1	3	
dcbi	LSU	Store	1	3	
dcblc	LSU	Store	1	3	
dcblq.	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	
divd	CFX	—	4 to 26	4 to 26	See Section 10.4.2.1, "CFX divide execution."
divd.	CFX	—	4 to 26	4 to 26	See Section 10.4.2.1, "CFX divide execution."
divdo	CFX	—	4 to 26	4 to 26	See Section 10.4.2.1, "CFX divide execution."
divdo.	CFX	—	4 to 26	4 to 26	See Section 10.4.2.1, "CFX divide execution."
divdu	CFX	—	4 to 26	4 to 26	See Section 10.4.2.1, "CFX divide execution."
divdu.	CFX	—	4 to 26	4 to 26	See Section 10.4.2.1, "CFX divide execution."
divduo	CFX	—	4 to 26	4 to 26	See Section 10.4.2.1, "CFX divide execution."
divduo.	CFX	—	4 to 26	4 to 26	See Section 10.4.2.1, "CFX divide execution."
divw	CFX	—	4 to 16	4 to 16	See Section 10.4.2.1, "CFX divide execution."
divw.	CFX	—	4 to 16	4 to 16	See Section 10.4.2.1, "CFX divide execution."
divwo	CFX	—	4 to 16	4 to 16	See Section 10.4.2.1, "CFX divide execution."
divwo.	CFX	—	4 to 16	4 to 16	See Section 10.4.2.1, "CFX divide execution."
divwu	CFX	—	4 to 16	4 to 16	See Section 10.4.2.1, "CFX divide execution."
divwu.	CFX	—	4 to 16	4 to 16	See Section 10.4.2.1, "CFX divide execution."
divwuo	CFX	—	4 to 16	4 to 16	See Section 10.4.2.1, "CFX divide execution."
divwuo.	CFX	—	4 to 16	4 to 16	See Section 10.4.2.1, "CFX divide execution."

Table 10-4. e6500 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.
dni	COMP	Refetch	—	—	The **dni** instruction executes during completion and is not dispatched to an execution unit.
dsn	LSU	Store	1	3	
dss	—	—	—	—	This instruction is treated as a no-op.
dssall	—	—	—	—	This instruction is treated as a no-op.
dst	—	—	—	—	This instruction is treated as a no-op.
dstst	—	—	—	—	This instruction is treated as a no-op.
dststt	—	—	—	—	This instruction is treated as a no-op.
dstt	—	—	—	—	This instruction is treated as a no-op.
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	
extsw	SFX0, SFX1	—	1	1	
extsw.	SFX0, SFX1	—	1	1	
fabs	FPU	—	1	7	
fabs.	FPU	—	1	7	
fadd	FPU	—	1	7	
fadd.	FPU	—	1	7	
fadds	FPU	—	1	7	
fadds.	FPU	—	1	7	
fcfid	FPU	—	1	7	
fcfid.	FPU	—	1	7	
fcmpo	FPU	—	1	7	
fcmpu	FPU	—	1	7	
fctid	FPU	—	1	7	
fctid.	FPU	—	1	7	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
fctidz	FPU	—	1	7	
fctidz.	FPU	—	1	7	
fctiw	FPU	—	1	7	
fctiw.	FPU	—	1	7	
fctiwz	FPU	—	1	7	
fctiwz.	FPU	—	1	7	
fdiv	FPU	—	2 or 31	7 or 35	Lower repeat rate and latency when dividend is 0, divisor is 0 or either is a NaN or Infinity.
fdiv.	FPU	—	2 or 31	7 or 35	Lower repeat rate and latency when dividend is 0, divisor is 0 or either is a NaN or Infinity.
fdivs	FPU	—	2 or 16	7 or 20	Lower repeat rate and latency when dividend is 0, divisor is 0 or either is a NaN or Infinity.
fdivs.	FPU	—	2 or 16	7 or 20	Lower repeat rate and latency when dividend is 0, divisor is 0 or either is a NaN or Infinity.
fmadd	FPU	—	1	7	
fmadd.	FPU	—	1	7	
fmadds	FPU	—	1	7	
fmadds.	FPU	—	1	7	
fmr	FPU	—	1	7	
fmr.	FPU	—	1	7	
fmsub	FPU	—	1	7	
fmsub.	FPU	—	1	7	
fmsubs	FPU	—	1	7	
fmsubs.	FPU	—	1	7	
fmul	FPU	—	1	7	
fmul.	FPU	—	1	7	
fmuls	FPU	—	1	7	
fmuls.	FPU	—	1	7	
fnabs	FPU	—	1	7	
fnabs.	FPU	—	1	7	
fneg	FPU	—	1	7	
fneg.	FPU	—	1	7	
fnmadd	FPU	—	1	7	
fnmadd.	FPU	—	1	7	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
fnmadds	FPU	—	1	7	
fnmadds.	FPU	—	1	7	
fnmsub	FPU	—	1	7	
fnmsub.	FPU	—	1	7	
fnmsubs	FPU	—	1	7	
fnmsubs.	FPU	—	1	7	
fres	FPU	—	2	8	
fres.	FPU	—	2	8	
frsp	FPU	—	1	7	
frsp.	FPU	—	1	7	
frsqrte	FPU	—	2	8	
frsqrte.	FPU	—	2	8	
fsel	FPU	—	1	7	
fsel.	FPU	—	1	7	
fsub	FPU	—	1	7	
fsub.	FPU	—	1	7	
fsubs	FPU	—	1	7	
fsubs.	FPU	—	1	7	
icbi	LSU	Store	1	3	
icbiep	LSU	Store	1	3	
icblc	LSU	Store	1	3	
icblq.	LSU	Store	1	3	
icbt	LSU	—	1	3	Note **icbt** with CT = 0, is treated as a no-op.
icbtls	LSU	Presync, postsync	1	3	Actual latency and repeat rate are likely much larger.
isel	SFX0, SFX1	—	1	1	
isync	COMP	Refetch	1	1	
lbarx	LSU	Presync	3	3	
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	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
lbzu	LSU	—	1	3	
lbzux	LSU	—	1	3	
lbzx	LSU	—	1	3	
ld	LSU	—	1	3	
ldarx	LSU	Presync	3	3	
ldbrx	LSU	—	1	3	
lddx	LSU	—	—	—	
ldepx	LSU	—	1	3	
ldu	LSU	—	1	3	
ldux	LSU	—	1	3	
ldx	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	
lharx	LSU	Presync	3	3	
lhau	LSU	—	1	3	
lhaux	LSU	—	1	3	
lhax	LSU	—	1	3	
lhbrx	LSU	—	1	3	
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	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
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** actually stalls in decode while completion queue entries are allocated for it each cycle.
lvebx	LSU	—	1	4	
lvehx	LSU	—	1	4	
lvewx	LSU	—	1	4	
lvexbx	LSU	—	1	4	
lvexhx	LSU	—	1	4	
lvexwx	LSU	—	1	4	
lvtlx	LSU	—	1	4	
lvtlxl	LSU	—	1	4	
lvtrx	LSU	—	1	4	
lvtrxl	LSU	—	1	4	
lvsl	LSU	—	1	4	
lvsm	LSU	—	1	4	
lvsr	LSU	—	1	4	
lvswx	LSU	—	1	4	
lvswxl	LSU	—	1	4	
lvx	LSU	—	1	4	
lvxl	LSU	—	1	4	
lwa	LSU	—	1	3	
lwarx	LSU	Presync	3	3	
lwaux	LSU	—	1	3	
lwax	LSU	—	1	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	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
lwzux	LSU	—	1	3	
lwzx	LSU	—	1	3	
mbar	LSU	Store	1	3	In general, **mbar** takes several more cycles to perform the ordering.
mcrf	BU	—	1	1	
mcrfs	FPU	—	1	7	
mcrxr	BU	Presync, postsync	1	1	
mfcr	CFX	Move-from	5	5	
mffs	FPU	—	1	7	
mffs.	FPU	—	1	7	
mfmsr	CFX	—	4	4	
mfocrf	SFX0, SFX1		1	1	
mfpmr	CFX	—	4	4	
mfspr (CTR)	SFX0, SFX1	—	1	1	CTR is fully renamed.
mfspr (DBSR)	CFX	Presync, postsync	4	4	
mfspr (LR)	SFX0, SFX1	—	1	1	LR is fully renamed.
mfspr (other)	CFX	—	4	4	
mfspr (XER)	CFX	Move-from	5	5	
mftmr	CFX	Move-from	5	5	
mftb	CFX	—	4	4	
mfvscr	VFPU	Unit	5	2	
miso	LSU	—	1	3	
msgclr	CFX	Move-to	1	1	
msgsnd	LSU	Store	1	3	
mtcrf	CFX	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	Unit	7	7	
mtfsb0.	FPU	Unit	7	7	
mtfsb1	FPU	Unit	7	7	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
mtfsb1.	FPU	Unit	7	7	
mtfsf	FPU	Unit	7	7	
mtfsf.	FPU	Unit	7	7	
mtfsfi	FPU	Unit	7	7	
mtfsfi.	FPU	Unit	7	7	
mtmsr	CFX	Presync, postsync, move-to	4	2	
mtocrf	CFX	—	1	1	
mtpmr	CFX	Move-to	1	1	
mtspr (CTR)	SFX0, SFX1	—	1	1	CTR is fully renamed.
mtspr (DBCR0, DBSR, or DBSRWR)	CFX	Presync, postsync, move-to	4	2	
mtspr (LR)	SFX0, SFX1	—	1	1	LR is fully renamed.
mtspr (NPIDR)	CFX	Postsync, move-to	4	2	
mtspr (other)	CFX	Move-to	1	1	
mtspr (PID)	CFX	Presync, postsync, move-to	4	2	
mtspr (XER)	CFX	Postsync, move-to	4	2	
mttmr	CFX	Postsync, move-to	4	2	
mtvscr	VFPU	Unit	5	2	
mulhd	CFX	—	2 to 4	4 to 7	See Section 10.4.2.2, “CFX multiply execution.”
mulhd.	CFX	—	2 to 4	4 to 7	See Section 10.4.2.2, “CFX multiply execution.”
mulhdu	CFX	—	2 to 4	4 to 7	See Section 10.4.2.2, “CFX multiply execution.”
mulhdu.	CFX	—	2 to 4	4 to 7	See Section 10.4.2.2, “CFX multiply execution.”
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.”
mulld	CFX	—	2 to 4	4 to 7	See Section 10.4.2.2, “CFX multiply execution.”

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
mulld.	CFX	—	2 to 4	4 to 7	See Section 10.4.2.2, "CFX multiply execution."
mulldo	CFX	—	2 to 4	4 to 7	See Section 10.4.2.2, "CFX multiply execution."
mulldo.	CFX	—	2 to 4	4 to 7	See Section 10.4.2.2, "CFX multiply execution."
mulli	CFX	—	2	4 or 5	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."
mvidsplt	LSU	—	1	4	
mviwsplt	LSU	—	1	4	
nand	SFX0, SFX1	—	1	1	
nand.	SFX0, SFX1	—	1	1	
neg	SFX0, SFX1	—	1	1	
neg.	SFX0, SFX1	—	1	1	
nego	SFX0, SFX1	—	2	2	
nego.	SFX0, SFX1	—	2	2	
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	
popcntb	SFX0	—	2	2	
popcntd	SFX0	—	2	2	
popcntw	SFX0	—	2	2	
prtyd	SFX0, SFX1	—	1	1	
prtyw	SFX0, SFX1	—	1	1	
rfci	COMP	Refetch	—	—	Return-from-interrupt instructions execute during completion and are not dispatched to an execution unit.

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
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.
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.
rldcl	SFX0, SFX1	—	2	2	
rldcl.	SFX0, SFX1	—	2	2	
rldcr	SFX0, SFX1	—	2	2	
rldcr.	SFX0, SFX1	—	2	2	
rldic	SFX0, SFX1	—	1	1	
rldic.	SFX0, SFX1	—	1	1	
rldicl	SFX0, SFX1	—	1	1	
rldicl.	SFX0, SFX1	—	1	1	
rldicr	SFX0, SFX1	—	1	1	
rldicr.	SFX0, SFX1	—	1	1	
rldimi	SFX0, SFX1	—	1	1	
rldimi.	SFX0, SFX1	—	1	1	
rlwimi	SFX0, SFX1	—	1	1	
rlwimi.	SFX0, SFX1	—	1	1	
rlwinm	SFX0, SFX1	—	1	1	
rlwinm.	SFX0, SFX1	—	1	1	
rlwnm	SFX0, SFX1	—	2	2	
rlwnm.	SFX0, SFX1	—	2	2	
sc	COMP	Refetch	—	—	The **sc** instruction executes during completion and is not dispatched to an execution unit.
sld	SFX0, SFX1	—	2	2	
sld.	SFX0, SFX1	—	2	2	
slw	SFX0, SFX1	—	2	2	
slw.	SFX0, SFX1	—	2	2	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
srad	SFX0, SFX1	—	2	2	
srad.	SFX0, SFX1	—	2	2	
sradi	SFX0, SFX1	—	1	1	
sradi.	SFX0, SFX1	—	1	1	
sraw	SFX0, SFX1	—	2	2	
sraw.	SFX0, SFX1	—	2	2	
srawi	SFX0, SFX1	—	1	1	
srawi.	SFX0, SFX1	—	1	1	
srd	SFX0, SFX1	—	2	2	
srd.	SFX0, SFX1	—	2	2	
srw	SFX0, SFX1	—	2	2	
srw.	SFX0, SFX1	—	2	2	
stb	LSU	Store	1	3	
stbcx.	LSU	Presync, postsync, 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	
std	LSU	Store	1	3	
stdbrx	LSU	Store	1	3	
stdcx.	LSU	Presync, postsync, store	1	3	
stddx	LSU	Store	1	3	
stdepx	LSU	Store	1	3	
stdu	LSU	Store	1	3	
stdux	LSU	Store	1	3	
stdx	LSU	Store	1	3	
stfd	LSU	Store	1	3	
stfddx	LSU	Store	1	3	
stfdepx	LSU	Store	1	3	
stfdu	LSU	Store	1	3	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
stfdux	LSU	Store	1	3	
stfdx	LSU	Store	1	3	
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	
sthcx.	LSU	Presync, postsync, 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** actually stalls in decode while completion queue entries are allocated for it each cycle.
stvebx	LSU	—	1	4	
stvehx	LSU	—	1	4	
stvewx	LSU	—	1	4	
stvexbx	LSU	—	1	4	
stvexhx	LSU	—	1	4	
stvexwx	LSU	—	1	4	
stvflx	LSU	—	1	4	
stvflxl	LSU	—	1	4	
stvfrx	LSU	—	1	4	
stvfrxl	LSU	—	1	4	
stvswx	LSU	—	1	4	
stvswxl	LSU	—	1	4	
stvx	LSU	—	1	4	
stvxl	LSU	—	1	4	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
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	—	2	2	
subfco.	SFX0, SFX1	—	2	2	
subfe	SFX0, SFX1	—	1	1	
subfe.	SFX0, SFX1	—	1	1	
subfeo	SFX0, SFX1	—	2	2	
subfeo.	SFX0, SFX1	—	2	2	
subfic	SFX0, SFX1	—	1	1	
subfme	SFX0, SFX1	—	1	1	
subfme.	SFX0, SFX1	—	1	1	
subfmeo	SFX0, SFX1	—	2	2	
subfmeo.	SFX0, SFX1	—	2	2	
subfo	SFX0, SFX1	—	2	2	
subfo.	SFX0, SFX1	—	2	2	
subfze	SFX0, SFX1	—	1	1	
subfze.	SFX0, SFX1	—	1	1	
subfzeo	SFX0, SFX1	—	2	2	
subfzeo.	SFX0, SFX1	—	2	2	
sync (msync)	LSU	Postsync, store	1	3	In general, **sync** takes several more cycles to perform the ordering.
td	SFX0	—	2	2	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
tdi	SFX0	—	2	2	
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	CFX	Presync, postsync, move-to	4	2	
tlbsx	CFX	Presync, postsync, move-to	4	2	
tlbsync	LSU	Store	1	3	
tlbwe	CFX	Presync, postsync, move-to	4	2	
tw	SFX0	—	2	2	
twi	SFX0	—	2	2	
vabsdub	VSFX	—	1	1	
vabsduh	VSFX	—	1	1	
vabsduw	VSFX	—	1	1	
vaddcuw	VSFX	—	1	1	
vaddfp	VFPU	—	1	6	
vaddsbs	VSFX	—	1	1	
vaddshs	VSFX	—	1	1	
vaddsws	VSFX	—	1	1	
vaddubm	VSFX	—	1	1	
vaddubs	VSFX	—	1	1	
vadduhm	VSFX	—	1	1	
vadduhs	VSFX	—	1	1	
vadduwm	VSFX	—	1	1	
vadduws	VSFX	—	1	1	
vand	VSFX	—	1	1	
vandc	VSFX	—	1	1	
vavgsb	VSFX	—	1	1	
vavgsh	VSFX	—	1	1	
vavgsw	VSFX	—	1	1	
vavgub	VSFX	—	1	1	
vavguh	VSFX	—	1	1	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
vavguw	VSFX	—	1	1	
vcfsx	VFPU	—	1	6	
vcfux	VFPU	—	1	6	
vcmpbfp	VPERM	—	1	2	
vcmpbfp.	VPERM	—	1	2	CR result is 2 cycle latency to Branch unit.
vcmpeqfp	VPERM	—	1	2	
vcmpeqfp.	VPERM	—	1	2	CR result is 2 cycle latency to Branch unit.
vcmpequb	VSFX	—	1	1	
vcmpequb.	VSFX	—	1	1	CR result is 2 cycle latency to Branch unit.
vcmpequh	VSFX	—	1	1	
vcmpequh.	VSFX	—	1	1	CR result is 2 cycle latency to Branch unit.
vcmpequw	VSFX	—	1	1	
vcmpequw.	VSFX	—	1	1	CR result is 2 cycle latency to Branch unit.
vcmpgefp	VPERM	—	1	2	
vcmpgefp.	VPERM	—	1	2	CR result is 2 cycle latency to Branch unit.
vcmpgtfp	VPERM	—	1	2	
vcmpgtfp.	VPERM	—	1	2	CR result is 2 cycle latency to Branch unit.
vcmpgtsb	VSFX	—	1	1	
vcmpgtsb.	VSFX	—	1	1	CR result is 2 cycle latency to Branch unit.
vcmpgtsh	VSFX	—	1	1	
vcmpgtsh.	VSFX	—	1	1	CR result is 2 cycle latency to Branch unit.
vcmpgtsw	VSFX	—	1	1	
vcmpgtsw.	VSFX	—	1	1	CR result is 2 cycle latency to Branch unit.
vcmpgtub	VSFX	—	1	1	
vcmpgtub.	VSFX	—	1	1	CR result is 2 cycle latency to Branch unit.
vcmpgtuh	VSFX	—	1	1	
vcmpgtuh.	VSFX	—	1	1	CR result is 2 cycle latency to Branch unit.
vcmpgtuw	VSFX	—	1	1	
vcmpgtuw.	VSFX	—	1	1	CR result is 2 cycle latency to Branch unit.
vctsxs	VFPU	—	1	6	
vctuxs	VFPU	—	1	6	
vexptefp	VFPU	—	1	6	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
vlogefp	VFPU	—	1	6	
vmaddfp	VFPU	—	1	6	
vmaxfp	VPERM	—	1	2	
vmaxsb	VSFX	—	1	1	
vmaxsh	VSFX	—	1	1	
vmaxsw	VSFX	—	1	1	
vmaxub	VSFX	—	1	1	
vmaxuh	VSFX	—	1	1	
vmaxuw	VSFX	—	1	1	
vmhaddshs	VCFX	—	1	4	
vmhraddshs	VCFX	—	1	4	
vminfp	VPERM	—	1	2	
vminsb	VSFX	—	1	1	
vminsh	VSFX	—	1	1	
vminsw	VSFX	—	1	1	
vminub	VSFX	—	1	1	
vminuh	VSFX	—	1	1	
vminuw	VSFX	—	1	1	
vmladduhm	VCFX	—	1	4	
vmrghb	VPERM	—	1	2	
vmrghh	VPERM	—	1	2	
vmrghw	VPERM	—	1	2	
vmrglb	VPERM	—	1	2	
vmrglh	VPERM	—	1	2	
vmrglw	VPERM	—	1	2	
vmsummbm	VCFX	—	1	4	
vmsumshm	VCFX	—	1	4	
vmsumshs	VCFX	—	1	4	
vmsumubm	VCFX	—	1	4	
vmsumuhm	VCFX	—	1	4	
vmsumuhs	VCFX	—	1	4	
vmulesb	VCFX	—	1	4	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
vmulesh	VCFX	—	1	4	
vmuleub	VCFX	—	1	4	
vmuleuh	VCFX	—	1	4	
vmulosb	VCFX	—	1	4	
vmulosh	VCFX	—	1	4	
vmuloub	VCFX	—	1	4	
vmulouh	VCFX	—	1	4	
vnmsubfp	VFPU	—	1	6	
vnor	VSFX	—	1	1	
vor	VSFX	—	1	1	
vperm	VPERM	—	1	2	
vpkpx	VPERM	—	1	2	
vpkshss	VPERM	—	1	2	
vpkshus	VPERM	—	1	2	
vpkswss	VPERM	—	1	2	
vpkswus	VPERM	—	1	2	
vpkuhum	VPERM	—	1	2	
vpkuhus	VPERM	—	1	2	
vpkuwum	VPERM	—	1	2	
vpkuwus	VPERM	—	1	2	
vrefp	VFPU	—	2	7	
vrfim	VFPU	—	1	6	
vrfin	VFPU	—	1	6	
vrfip	VFPU	—	1	6	
vrfiz	VFPU	—	1	6	
vrlb	VSFX	—	1	1	
vrlh	VSFX	—	1	1	
vrlw	VSFX	—	1	1	
vrsqrtefp	VFPU	—	2	7	
vsel	VSFX	—	1	1	
vsl	VPERM	—	1	2	
vslb	VSFX	—	1	1	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
vsldoi	VPERM	—	1	2	
vslh	VSFX	—	1	1	
vslo	VPERM	—	1	2	
vslw	VSFX	—	1	1	
vspltb	VPERM	—	1	2	
vsplth	VPERM	—	1	2	
vspltisb	VPERM	—	1	2	
vspltish	VPERM	—	1	2	
vspltisw	VPERM	—	1	2	
vspltw	VPERM	—	1	2	
vsr	VPERM	—	1	2	
vsrab	VSFX	—	1	1	
vsrah	VSFX	—	1	1	
vsraw	VSFX	—	1	1	
vsrb	VSFX	—	1	1	
vsrh	VSFX	—	1	1	
vsro	VPERM	—	1	2	
vsrw	VSFX	—	1	1	
vsubcuw	VSFX	—	1	1	
vsubfp	VFPU	—	1	6	
vsubsbs	VSFX	—	1	1	
vsubshs	VSFX	—	1	1	
vsubsws	VSFX	—	1	1	
vsububm	VSFX	—	1	1	
vsububs	VSFX	—	1	1	
vsubuhm	VSFX	—	1	1	
vsubuhs	VSFX	—	1	1	
vsubuwm	VSFX	—	1	1	
vsubuws	VSFX	—	1	1	
vsum2sws	VCFX	—	1	4	
vsum4sbs	VCFX	—	1	4	
vsum4shs	VCFX	—	1	4	

Table 10-4. e6500 instruction latencies (continued)

Mnemonic	Execution Unit(s)	Serialization	Repeat Rate (cycles)	Latency (cycles)	Notes
vsum4ubs	VCFX	—	1	4	
vsumsws	VCFX	—	1	4	
vupkhpx	VPERM	—	1	2	
vupkhsb	VPERM	—	1	2	
vupkhsb	VPERM	—	1	2	
vupklpx	VPERM	—	1	2	
vupklsb	VPERM	—	1	2	
vupklsh	VPERM	—	1	2	
vxor	VSFX	—	1	1	
wait	COMP	Refetch	—	—	The **wait** instruction executes during completion and is not dispatched to an execution unit.
wrtee	CFX	Move-to, postsync	4	2	
wrteei	CFX	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 full 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 28 instructions in each thread (with the instruction that sets CR in CQ0 and the dependent branch instruction in IQ11), there is no advantage to having more than 26 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 16 instructions can be assigned CR and GPR renames and can be assigned CQ entries; therefore, 16 instructions can be in the execute stages at any one time. (Note the exception, however, of load or store with update instructions, which are broken into two instructions at dispatch.)
- Avoid branches where possible; favor using **isel** over not-taken branches over taken branches.
- Lay out your instruction effective address space such that there are not large numbers of 4 GB chunks being used simultaneously. The branch predictor predicts the upper 32-bits of the fetch address for branches and can keep only eight simultaneous upper 32-bit values.

Chapter 11
Core and Cluster 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 e6500 cluster, each of the cores in the cluster, and each thread (processor) within each core.

One thread of one core is generally used to execute the software used to initialize both its own core and resources that are shared by all cores within its cluster. In a device that integrates more than one e6500 cluster, the same thread can initialize non-core resources in the other e6500 cluster(s) and other parts of the integrated device. After completing initialization of the cluster(s) and integrated device, the thread signals the integrated device to allow software execution by other cores. One thread of every other core then executes the software required to initialize its core and the other thread in its core. There are other requirements for software to initialize areas outside of the core and its cluster (for example, other e6500 clusters and remainder of an integrated device), which are not addressed here. See the integrated device reference manual for more information.

11.1 Core and cluster state and software initialization after reset

The state of each area of the cluster and each core within the cluster 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 and cluster. Note that, in general, the boot loader software performs most of these actions. Thus, operating systems or hypervisors generally start execution with this state appropriately initialized.

11.2 MMU state

At reset, the valid bit (TLB[V]) of every entry in both TLB0 and TLB1 of the MMU is set to 0 (invalid) except for the initial boot page, which is described in Section 6.7, “TLB and LRAT states after reset.” In addition, all instruction and data L1MMU entries are invalidated. No other information in the invalid TLB entries is initialized. If later software depends on certain fields in TLB entries to be set to known values, software must write those values by individually writing the fields of the TLB entries to the required values.

11.3 Thread state

At reset, thread 0 of each core is enabled and begins execution at 0xFFFFFFFC. Software executed by thread 0 should perform core initialization and then may set the initial starting execution address and MSR of the other thread(s) by writing the appropriate INIA and IMSR before enabling thread 1 to execute by writing TENS.

Note that, generally, only one core in an integrated device is allowed by the integrated device to fetch instructions until software running on that core configures the integrated device to allow other cores to proceed. See the integrated device reference manual for more information.

11.4 Core register state

11.4.1 GPRs

At 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 (for example, 0) after reset. Zeroing the GPRs 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
```

11.4.2 FPRs

At reset, the FPU is disabled, and the 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 immediately after the FPU has been enabled after reset. This can be accomplished by loading the FPRs with a known value from memory.

Note that loading the FPRs from memory may not be able to be performed until later in the boot process or possibly at the start of the operating system or hypervisor, when software has properly initialized memory. The following code sequence can be used to enable the FPU and clear the FPRs, 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
```

11.4.3 VRs

At reset, the AltiVec unit is disabled and the VRs may contain random values that may differ from core to core or may differ from reset to reset. Practically, an VR 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 VRs should be set to known values immediately after the AltiVec unit is enabled after reset. This can be accomplished by loading the VPRs with a known value from either a GPR or memory. If the desired known value for all VPRs is 0, the following code sequence can be used:

```
mfmsr    r5              // get current MSR
xor      r4,r4,r4        // set r4 to 0
oris     r4,r5,0x0200    // set MSR[SPV]
mtmsr    r4
isync
vxor     v0,v0,v0        // set to 0
vxor     v1,v1,v1        // set to 0
...                      // do for all 32 vector registers
mtmsr    r5              // restore MSR (turn off SPV if desired)
isync
```

11.4.4 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. Writable SPRs that have initial values other than 0 out of reset are shown in the following table.

Table 11-1. SPRs and TMRs with non-zero reset values

SPR	Description of Non-Zero Reset Values
CDCSR0	Set to configuration information denoting presence of Floating-Point capability and presence and state of AltiVec facility.
CIR	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. This register is an alias to SVR.
DBSR	DBSR[MRR] is set to reflect the most recent reset, which after a hard reset will be 0b10.
EPTCFG	Set to configuration information describing the pagetable capabilities and organization.
INIA0, INIA1	Set to 0xFFFFFFFC.
L1CFG0	Set to configuration information describing the L1 cache capabilities and organization.
L1CFG1	Set to configuration information describing the L1 cache capabilities and organization.
LRATCFG	Set to configuration information describing the LRAT capabilities and organization.
LRATPS	Set to configuration information describing the LRAT page size availability.
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 and TMRs with non-zero reset values (continued)

SPR	Description of Non-Zero Reset Values
SCCSRBAR	Set to a value which reflects the current setting of the SoC CCSRBAR.
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. This register is an alias to CIR.
TLB0CFG	Set to configuration information describing the TLB0 capabilities and organization.
TLB1CFG	Set to configuration information describing the TLB1 capabilities and organization.
TLB0PS	Set to configuration information describing the TLB0 page size availability.
TLB1PS	Set to configuration information describing the TLB1 page size availability.
TENC, TENS, TENSR	Set to thread 0 enabled, all other threads disabled.

Other SPRs need to be set up by software, particularly those SPRs that enable and control various aspects about how the core operates.

This table lists SPRs for which software should initialize to appropriate values at boot time.

Table 11-2. SPRs to configure the e6500

SPR	What to Configure
BUCSR	Branch unit control and status register of each thread (processor). See Section 2.7.4, “Branch Unit Control and Status (BUCSR) register.”
L1CSR0	L1 Control and Status register. See Section 2.11, “L1 cache registers.”
L1CSR1	L1 Control and Status register. See Section 2.11, “L1 cache registers.”
L1CSR2	L1 Control and Status register. See Section 2.11, “L1 cache registers.”
PWRMGTCR0	Power Management Control register. See Section 2.7.7, “Power Management Control 0 (PWRMGTCR0) register.”
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. See Section 2.7.5, “Hardware Implementation-Dependent 0 (HID0) register.”

11.4.5 MSR, FPSCR, and VSCR

At reset, the MSR, FPSCR, and VSCR of each thread are set to 0. The FPSCR and VSCR do not require initialization and can be set at a later time before floating-point or AltiVec is used depending on which modes software wishes to operate in.

11.5 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 must write 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 begins counting immediately out of reset. However, because Time Base ticks are externally signaled to the core, the Time Base begins 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.

11.6 L1 cache state

At reset, both the instruction and data L1 caches are disabled. The contents of the L1 caches is random. There can be random values for tag bits, data bits, valid bits, coherency bits, and lock bits. Software must properly initialize an L1 cache before it is enabled. (Enabling an L1 cache is described in Section 5.6.2, "Enabling and disabling the L1 caches. Note that enabling either L1 cache without first enabling the L2 cache is not supported.)

Initialization of an L1 cache can be accomplished by flash invalidating the L1 cache and the L1 cache locks. These operations clear the valid and lock bits for all cache lines. The tag bits and data bits do not need to be initialized after flash invalidation because all lines and tags are invalid and are set correctly when a new cache line is allocated. Software should execute the following code sequence to flash invalidate the L1 caches prior to enabling them:

```
      // L1 data cache
      xor     r4,r4,r4      // set r4 to 0
      ori     r5,r4,0x0102  // set CFLC and CFI 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 written 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 ICFLC and ICFI 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 written bits
      bne     iloop          // check again if not complete
      isync                  // discard prefetched instructions
```

After the L1 caches have been invalidated, they can be enabled by writing to the L1CSR0[CE] and L1CSR1[ICE] bits, respectively. Parity checking can also be enabled by writing to the appropriate bits in L1CSR0 and L1CSR1. See Section 2.11, "L1 cache registers," for descriptions of L1CSR0 and L1CSR1.

11.7 L2 cache state

The L2 cache is shared by all cores in an e6500 cluster and contains both the cache memory and the interface between the cluster and the integrated device. The L2 cache is controlled by memory mapped registers, as described in Section 2.2.3, "Memory-mapped registers (MMRs)."

At reset, the L2 cache memory is disabled and its contents are random (that is, 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 clearing the valid bits and lock bits for all L2 cache lines by flash invalidating both the L2 cache and the L2 cache locks. The lock bits must be cleared because the L2 cache supports persistent locks. If the lock bits are not cleared, then, on average, 50% of the cache appears to be locked, and those lines are not available for allocation, which causes serious performance consequences. The L2 cache tag bits and data bits do not need to be initialized after flash invalidation, because all lines and tags are invalid and are set correctly when a new line is allocated.

The code sequence below assumes 64-bit mode, and real address mapped the same as effective addresses. If SCCSRBAR resides in the first 4 GB of real address space, this code also works in 32-bit mode.

To flash invalidate the L2 cache, software should execute the following code sequence in prior to enabling the L2 cache:

```
      // L2 data cache invalidation & unlocking
      lis      r4,0x0020      // create flash invalidate & unlock bit mask (see Table 2-19)
      ori      r4,r4,0x0400   //
      mfspr    r5,SCCSRBAR    // get base address of memory mapped registers
      li       r7,24          // get shift count
      sld      r5,r5,r7
      lis      r6,0x00C2      // block offset for desired cluster (see Table 2-4)
      // subsequent cluster L2 caches may be invalidated & unlocked by adding 0x40000 to r6
      add      r6,r6,r5       //
      addi     r6,r6,0        // L2SCR0 offset (see Table 2-5), included here only for example
      sync                    // ensure prior memory transactions are performed
      stw      r4,0(r6)       // write L2SCR0 MMR to flash invalidate L2 cache and locks

l2loop:
      lwz      r5,(0)r6       // get current L2SCR0 MMR value
      and.     r5,r5,r4       // compare to mask to see if complete
      bne      l2loop
      isync                   //
```

After the L2 cache has been invalidated, it can be enabled by writing to the L2CSR0[L2E] bit (bit 32). Error detection and correction can be enabled, as well, by writing to the appropriate bits in the L2CSR0 register. See Section 2.12, "L2 cache registers," for descriptions of L2CSR0 and L2 error management registers.

11.8 Branch target buffer state

At reset, the branch prediction mechanisms of all threads of all cores are disabled. To obtain full performance of the e6500 core, branch prediction mechanisms should be enabled.

Also at reset, the contents of the branch target buffer is random, with random target addresses and random valid bits for BTB entries. While this does not cause any specific problem because the BTB and other branch predictor elements self-correct over time and mispredicted branches are resolved correctly, software should invalidate the contents of the BTB at boot. This assists in debugging boot software because fetch accesses are 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
oris     r5,r5,0x0140   // set STAC_EN and LS_EN
mtspr    BUCSR,r5       // flash invalidate and enable branch prediction
isync                    // synchronize setting of BUCSR
```

Appendix A Simplified Mnemonics

This chapter 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.

A.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 a conditional branch to a relative target if CR4 specifies a greater than condition without using simplified mnemonics, the programmer codes 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.

A.2 Subtract simplified mnemonics

This section describes simplified mnemonics for subtract instructions.

A.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 the following table.

Table A-1. Subtract immediate simplified mnemonics

Simplified Mnemonic	Standard Mnemonic
subi rD,**r**A,value	**addi r**D,**r**A,–value
subis rD,**r**A,value	**addis r**D,**r**A,–value

Table A-1. Subtract immediate simplified mnemonics (continued)

Simplified Mnemonic	Standard Mnemonic
subic rD,**r**A,value	**addic r**D,**r**A,–value
subic. rD,**r**A,value	**addic. r**D,**r**A,–value

A.2.2 Subtract

Subtract from instructions subtract the second operand (**r**A) from the third (**r**B). The simplified mnemonics in the following table use the more common order in which the third operand is subtracted from the second.

Table A-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.

A.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. (No simplified mnemonic is provided for insertion of a left-justified field when operating on doublewords because such an insertion requires more than one instruction.)
- 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 non-negative) array index by the width of an element.

A.3.1 Operations on words

The simplified mnemonics in the following table can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

Table A-3. Word rotate and shift simplified mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Extract and left justify word immediate	**extlwi** rA,**r**S,*n*,*b* (*n* > 0)	**rlwinm** rA,**r**S,*b*,**0**,*n* – 1
Extract and right justify word immediate	**extrwi** rA,**r**S,*n*,*b* (*n* > 0)	**rlwinm** rA,**r**S,*b* + *n*, 32 – *n*,**31**
Insert from left word immediate	**inslwi** rA,**r**S,*n*,*b* (*n* > 0)	**rlwimi** rA,**r**S,32 – *b*,*b*,(*b* + *n*) – 1
Insert from right word immediate	**insrwi** rA,**r**S,*n*,*b* (*n* > 0)	**rlwimi** rA,**r**S,32 – (*b* + *n*),*b*,(*b* + *n*) – 1
Rotate left word immediate	**rotlwi** rA,**r**S,*n*	**rlwinm** rA,**r**S,*n*,**0,31**
Rotate right word immediate	**rotrwi** rA,**r**S,*n*	**rlwinm** rA,**r**S,32 – *n*,**0,31**
Rotate word left	**rotlw** rA,**r**S,**r**B	**rlwnm** rA,**r**S,**r**B,**0,31**
Shift left word immediate	**slwi** rA,**r**S,*n* (*n* < 32)	**rlwinm** rA,**r**S,*n*,**0**,31 – *n*
Shift right word immediate	**srwi** rA,**r**S,*n* (*n* < 32)	**rlwinm** rA,**r**S,32 – *n*,*n*,**31**
Clear left word immediate	**clrlwi** rA,**r**S,*n* (*n* < 32)	**rlwinm** rA,**r**S,**0**,*n*,**31**
Clear right word immediate	**clrrwi** rA,**r**S,*n* (*n* < 32)	**rlwinm** rA,**r**S,**0,0**,31 – *n*
Clear left and shift left word immediate	**clrlslwi** rA,**r**S,*b*,*n* (*n* ≤ *b* ≤ 31)	**rlwinm** rA,**r**S,*n*,*b* – *n*,31 – *n*

The following examples use word mnemonics:

1. Extract the sign bit (bit 0) of **r**S and place the result right-justified into **r**A.
   extrwi rA,rS,1,0 equivalent to **rlwinm rA,rS,1,31,31**
2. Insert the bit extracted in (1) into the sign bit (bit 0) of **r**B.
   insrwi rB,rA,1,0 equivalent to **rlwimi rB,rA,31,0,0**
3. Shift the contents of **r**A left 8 bits.
   slwi rA,rA,8 equivalent to **rlwinm rA,rA,8,0,23**
4. Clear the high-order 16 bits of **r**S and place the result into **r**A.
   clrlwi rA,rS,16 equivalent to **rlwinm rA,rS,0,16,31**

A.3.2 Operations on doublewords

The simplified mnemonics in the following table can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

Table A-4. Doubleword rotate and shift simplified mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Extract and left justify doubleword immediate	**extldi** rA,**r**S,*n*,*b* (*n* > 0)	**rldicr** rA,**r**S,*b*,*n* – 1
Extract and right justify doubleword immediate	**extrdi** rA,**r**S,*n*,*b* (*n* > 0)	**rldicl** rA,**r**S,*b* + *n*, 64 – *n*
Insert from right doubleword immediate	**insrdi** rA,**r**S,*n*,*b* (*n* > 0)	**rldimi** rA,**r**S,64 – (*b* + *n*),*b*
Rotate left doubleword immediate	**rotldi** rA,**r**S,*n*	**rldicl** rA,**r**S,*n*,**0**
Rotate right doubleword immediate	**rotrdi** rA,**r**S,*n*	**rldicl** rA,**r**S,64 – *n*,**0**

Table A-4. Doubleword rotate and shift simplified mnemonics (continued)

Operation	Simplified Mnemonic	Equivalent to:
Rotate doubleword left	**rotld** rA,**r**S,**r**B	**rldcl** rA,**r**S,**r**B,**0**
Shift left doubleword immediate	**sldi** rA,**r**S,*n* (*n* < 64)	**rldicr** rA,**r**S,*n*,**63** – *n*
Shift right doubleword immediate	**srdi** rA,**r**S,*n* (*n* < 64)	**rldicl** rA,**r**S,64 – *n*,*n*
Clear left doubleword immediate	**clrldi** rA,**r**S,*n* (*n* < 64)	**rldicl** rA,**r**S,**0**,*n*
Clear right doubleword immediate	**clrrdi** rA,**r**S,*n* (*n* < 64)	**rldicr** rA,**r**S,**0**,**63** – *n*
Clear left and shift left doubleword immediate	**clrlsldi** rA,**r**S,*b*,*n* (*n* ≤ *b* ≤ 63)	**rldic** rA,**r**S,*n*,*b* – *n*

The following examples use word mnemonics:

1. Extract the sign bit (bit 0) of **r**S and place the result right-justified into **r**A.
   extrdi rA,**r**S,**1**,**0** equivalent to **rldicl r**A,**r**S,**1**,**63**
2. Insert the bit extracted in (1) into the sign bit (bit 0) of **r**B.
   insrdi rB,**r**A,**1**,**0** equivalent to **rldimi r**B,**r**A,**63**,**0**
3. Shift the contents of **r**A left 8 bits.
   sldi rA,**r**A,**8** equivalent to **rldicr r**A,**r**A,**8**,**55**
4. Clear the high-order 32 bits of **r**S and place the result into **r**A.
   clrldi rA,**r**S,**32** equivalent to **rldicl r**A,**r**S,**0**,**32**

A.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). The following table 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 A-5. 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 shown in the following figure for Branch Conditional (**bc**, **bca**, **bcl**, and **bcla**) instructions.

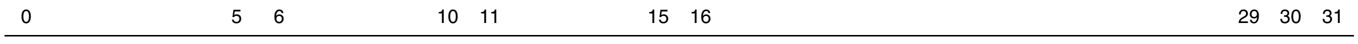


Figure A-1. Branch conditional (bc) instruction format

0 0 1 0 0 0	BO	BI	BD	AA	LK

Figure A-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 (for example, whether a comparison is less than or greater than). 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 A-7); the CR0 equal field is indicated by a BI value of 2 (as shown in Table A-8). 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 A-8. 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. For example, specifying a BI value of 22 actually selects bit 54 (BI value 22 + 32 = 54).

A.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 CR is not involved in the branch, the BI operand can be deleted.
- If 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).

A.4.2 Eliminating the BO operand

The 5-bit BO field, shown in Figure A-2, encodes the following operations in conditional branch instructions:

- Decrement count (CTR) register
  - And test if result is equal to zero
  - And test if result is not equal to zero
- Test condition (CR) register
  - 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 A.4.3, “Incorporating the BO branch prediction.”

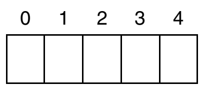


Figure A-2. BO field (bits 6–10 of the instruction encoding)

BO bits can be interpreted individually, as described in the following table.

Table A-6. 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 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 A.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 A-7. A *z* bit indicates that the bit is ignored. However, these bits should be cleared because they may be assigned a meaning in a future version of the architecture.

As shown in the following table, 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 that 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. If the BO field specifies that the CTR is to be decremented, the entire 64-bit CTR is decremented, regardless of the mode.

Table A-7. BO operand encodings

BO Field	Value[1] (Decimal)	Description	Symbol
0000*z*[2]	0	Decrement the CTR, then branch if the decremented CTR[M:63] ≠ 0; condition is FALSE.	**dnzf**
0001*z*	2	Decrement the CTR, then branch if the decremented CTR[M:63] = 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[M:63] ≠ 0; condition is TRUE.	**dnzt**
0101*z*	10	Decrement the CTR, then branch if the decremented CTR[M:63] = 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[M:63] ≠ 0.	**dnz**[6]
1*a*01*t*[5]	18	Decrement the CTR, then branch if the decremented CTR[M:63] = 0.	**dz** [6]
1*z*1*zz* [5]	20	Branch always.	—

[1] Assumes t = z = 0. Section A.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 because 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 A.4.6, “Simplified mnemonics that incorporate CR conditions (eliminate BO and replace 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.

A.4.3 Incorporating the BO branch prediction

As shown in Table A-7, 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 non-negative 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 standard of simplified branch conditional mnemonic.

For relative and absolute branches (**bc**[**l**][**a**]), the setting of the *at* bits depends on whether the displacement field is negative or non-negative. 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:

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–

A.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 A.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 A.4, "Branch instruction simplified mnemonics," where **bdnzt 4 * cr5 + eq,***target* could be used instead of **bdnzt 22,***target*. This is described in Section A.4.4.1.1, "Specifying a CR bit."

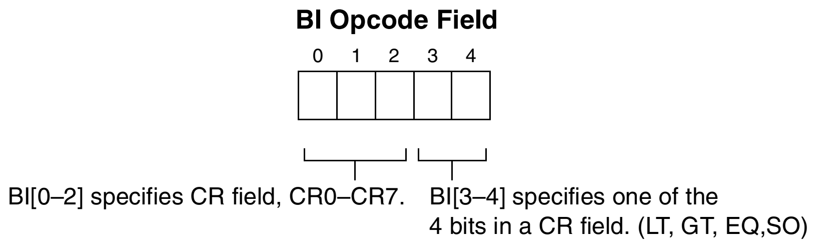
  The simplified mnemonics in Section A.4.5, "Simplified mnemonics that incorporate the BO operand," use one of these two methods to specify a CR bit.
- Additional simplified mnemonics are specified that incorporate CR conditions that would otherwise be specified by the BI operand, so the BI operand is replaced by the **cr**S operand to specify the CR field, CR0–CR7. See Section A.4.4.1, "BI operand instruction encoding."

  These mnemonics are described in Section A.4.6, "Simplified mnemonics that incorporate CR conditions (eliminate BO and replace BI with crS)."

A.4.4.1 BI operand instruction encoding

The entire 5-bit BI field, shown in Figure A-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 A.4.6, "Simplified mnemonics that incorporate CR conditions (eliminate BO and replace 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 the following figure shows, BI[0–2] indicates the CR field and BI[3–4] represents the condition to test.


BI Opcode Field
0
1
2
3
4
BI[0–2] specifies CR field, CR0–CR7.
BI[3–4] specifies one of the 4 bits in a CR field. (LT, GT, EQ,SO)

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 A-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 A-8.

A.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 A-8 and Table A-9.

Table A-8. 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.
CR1[0]	36(4)	001	00	Copy of FPSCR[FX] at the instruction's completion.
CR1[1]	37(5)	001	01	Copy of FPSCR[FEX] at the instruction's completion.
CR1[2]	38(6)	001	10	Copy of FPSCR[VX] at the instruction's completion.
CR1[3]	39(6)	001	11	Copy of FPSCR[OX] at the instruction's completion.

Some simplified mnemonics incorporate only the BO field, as described Section A.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 A-9.

Compare word instructions, described in Section A.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 A-8. CR logical instructions, described in Section A.7, “Condition register logical simplified mnemonics,” can update individual CR bits.

Table A-9. 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**)	0	32	000	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.
	4 * cr1 + lt	4	36	001		
	4 * cr2 + lt	8	40	010		
	4 * cr3+ lt	12	44	011		
	4 * cr4 + lt	16	48	100		
	4 * cr5 + lt	20	52	101		
	4 * cr6 + lt	24	56	110		
	4 * cr7 + lt	28	60	111		
CR*n*[1]	**4 * cr0 + gt** (or **gt**)	1	33	000	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.
	4 * cr1 + gt	5	37	001		
	4 * cr2 + gt	9	41	010		
	4 * cr3+ gt	13	45	011		
	4 * cr4 + gt	17	49	100		
	4 * cr5 + gt	21	53	101		
	4 * cr6 + gt	25	57	110		
	4 * cr7 + gt	29	61	111		
CR*n*[2]	**4 * cr0 + eq** (or **eq**)	2	34	000	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.
	4 * cr1 + eq	6	38	001		
	4 * cr2 + eq	10	42	010		
	4 * cr3+ eq	14	46	011		
	4 * cr4 + eq	18	50	100		
	4 * cr5 + eq	22	54	101		
	4 * cr6 + eq	26	58	110		
	4 * cr7 + eq	30	62	111		
CR*n*[3]	**4 * cr0 + so/un** (or **so/un**)	3	35	000	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.
	4 * cr1 + so/un	7	39	001		
	4* cr2 + so/un	11	43	010		
	4* cr3 + so/un	15	47	011		
	4* cr4 + so/un	19	51	100		
	4* cr5 + so/un	23	55	101		
	4* cr6 + so/un	27	59	110		
	4* cr7 + so/un	31	63	111		

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 which would be only marginally useful. The abbreviated set in Section A.4.5, “Simplified mnemonics that incorporate the BO operand,” covers useful cases. Unusual cases can be coded using a standard branch conditional syntax.

A.4.4.1.2 The crS operand

The **cr**S symbols are shown in the following table. Note that either the symbol or the operand value can be used in the syntax used with the simplified mnemonic.

Table A-10. 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**).

A.4.5 Simplified mnemonics that incorporate the BO operand

The mnemonics in the following table 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 A-11. 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**	—
Decrement CTR, branch if CTR = 0 [1]	**bdz**	**bdza**	**bdzlr**	—	**bdzl**	**bdzla**	**bdzlrl**	—

Table A-11. 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 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.

This table shows the syntax for basic simplified branch mnemonics

Table A-12. 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 A-7, where applicable.

[2] BI can be a numeric value or an expression as shown in Table A-10.

The simplified mnemonics in Table A-11 that test a condition require a corresponding CR bit as the first operand (as examples 2–5 in the following section illustrate). The symbols in Table A-10 can be used in place of a numeric value.

A.4.5.1 Examples that Eliminate the BO Operand

The simplified mnemonics in Table A-11 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*

This table lists simplified mnemonics and syntax for **bc** and **bca** without LR updating.

Table A-13. 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 A.4.6, "Simplified mnemonics that incorporate CR conditions (eliminate BO and replace 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.

This table lists simplified mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table A-14. 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 A-14. 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 A.4.6, "Simplified mnemonics that incorporate CR conditions (eliminate BO and replace BI with crS)."

This table provides simplified mnemonics and syntax for **bcl** and **bcla** with LR updating.

Table A-15. 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 A.4.6, "Simplified mnemonics that incorporate CR conditions (eliminate BO and replace 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.

This table provides simplified mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table A-16. 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 A-16. 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.

A.4.6 Simplified mnemonics that incorporate CR conditions (eliminate BO and replace BI with crS)

The mnemonics in Table A-19 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 eliminate 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 mnemonics 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 the following table 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 A-17. 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 A-17. Standard coding for branch conditions (continued)

Code	Description	Equivalent	Bit Tested
un	Unordered (after floating-point comparison)	—	SO
nu	Not unordered (after floating-point comparison)	—	SO

The following table 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 A-18. 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 A-17, where applicable.

[2] BI can be a numeric value or an expression as shown in Table A-10.

This table shows the simplified branch mnemonics incorporating conditions.

Table A-19. 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 A-19 indicate the condition bit but not the CR field. If no CR field is specified, CR0 is used. The CR field symbols defined in Table A-10 (**cr0–cr7**) are used for this operand, as shown in examples 2–4 in the following section.

A.4.6.1 Branch simplified mnemonics that incorporate CR conditions: examples

The following examples use the simplified mnemonics shown in Table A-19:

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**

A.4.6.2 Branch simplified mnemonics that incorporate CR conditions: listings

This table shows simplified branch mnemonics and syntax for **bc** and **bca** without LR updating.

Table A-20. 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.

The following table show the simplified mnemonics for **bc** and **bca** without LR updating, using the default CR0.

This table shows simplified branch mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table A-21. 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 unordered		**bunlr cr**S target		**bunctr 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
Branch if not unordered	—	**bnulr cr**S target	—	**bnuctr 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.

This table shows simplified branch mnemonics and syntax for **bcl** and **bcla** with LR updating.

Table A-22. 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

Table A-22. Simplified mnemonics for bcl and bcla with comparison conditions and LR update (continued)

Branch Semantics	bcl	Simplified Mnemonic	bcla	Simplified Mnemonic
Branch if summary overflow	**bcl 12**,BI[4],target	**bsol cr**S target	**bcla 12**,BI[4],target	**bsola cr**S target
Branch if unordered	—	**bunl cr**S target	—	**bunla 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
Branch if not unordered	—	**bnul cr**S target	—	**bnula 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.

This table shows the simplified mnemonics for **bcl** and **bcla** with LR updating, using the default CR0.

This table shows the simplified branch mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table A-23. 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 unordered	—	**bunlrl cr**S target	—	**bunctrl 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
Branch if not unordered	—	**bnulrl cr**S target	—	**bnuctrl 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.

A.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 the following table eliminate the L operand for word comparisons.

Table A-24. 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 the following three examples. 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

A.6 Compare doubleword simplified mnemonics

In compare double-word instructions, the L operand indicates a word (L = 0) or a double-word (L = 1). Simplified mnemonics in the following table eliminate the L operand for doubleword comparisons.

Table A-25. 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 the following three examples. 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,r**A**,100**
2. Same as (1), but place results in CR4.
   cmpdi cr4,rA**,100** equivalent to **cmpi 4,1,r**A**,100**
3. Compare **r**A and **r**B as unsigned 64-bit integers and place result in CR0.
   cmpld rA**,r**B equivalent to **cmpl 0,1,r**A**,r**B

A.7 Condition register logical simplified mnemonics

The CR logical instructions, shown in the following table, 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 A-9 can be used to identify the CR bit.

Table A-26. Condition register logical simplified mnemonics

Operation	Simplified Mnemonic	Equivalent to
Condition register set	**crset** bx	**creqv** bx,bx,bx
Condition register clear	**crclr** bx	**crxor** bx,bx,bx
Condition register move	**crmove** bx,by	**cror** bx,by,by
Condition register not	**crnot** bx,by	**crnor** bx,by,by

The following examples use the CR logical mnemonics:

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**

A.8 Trap instructions simplified mnemonics

The codes in the following table are for the most common combinations of trap conditions.

Table A-27. 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

Table A-27. Standard codes for trap instructions (continued)

Code	Description	TO Encoding	<	>	=	<U[1]	>U [2]
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 the following table are variations of trap instructions, with the most useful TO values represented in the mnemonic rather than specified as a numeric operand.

Table A-28. Trap simplified mnemonics

Trap Semantics	32-Bit Comparison		64-Bit Comparison	
	twi Immediate	tw Register	tdi Immediate	td Register
Trap unconditionally	—	**trap**	—	—
Trap if less than	**twlti**	**twlt**	**tdlti**	**tdlt**
Trap if less than or equal	**twlei**	**twle**	**tdlei**	**tdle**
Trap if equal	**tweqi**	**tweq**	**tdeqi**	**tdeq**
Trap if greater than or equal	**twgei**	**twge**	**tdgei**	**tdge**
Trap if greater than	**twgti**	**twgt**	**tdgti**	**tdgt**
Trap if not less than	**twnli**	**twnl**	**tdnli**	**tdnl**
Trap if not equal	**twnei**	**twne**	**tdnei**	**tdne**
Trap if not greater than	**twngi**	**twng**	**tdngi**	**tdng**
Trap if logically less than	**twllti**	**twllt**	**tdllti**	**tdllt**
Trap if logically less than or equal	**twllei**	**twlle**	**tdllei**	**tdlle**
Trap if logically greater than or equal	**twlgei**	**twlge**	**tdlgei**	**tdlge**
Trap if logically greater than	**twlgti**	**twlgt**	**tdlgti**	**tdlgt**
Trap if logically not less than	**twlnli**	**twlnl**	**tdlnli**	**tdlnl**
Trap if logically not greater than	**twlngi**	**twlng**	**tdlngi**	**tdlng**

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. This table lists these conditions.

Table A-29. 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

A.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.

The following examples use the SPR simplified mnemonics:

1. Copy the contents of the low-order 32 bits 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 the low-order 32 bits 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 the following table, along with the equivalent simplified mnemonic using the formula described above.

Table A-30. 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	

A.10 AltiVec simplified mnemonics

The following simplified mnemonics are supported:

Vector Move Register	
vmr **v**D,**v**S	equivalent to **vor** **v**D, **v**S, **v**S
Vector Logical Not	
vnot **v**D,**v**S	equivalent to **vnorv**D, **v**S, **v**S

A.11 Recommended simplified mnemonics

This section describes commonly-used operations (such as no-op, load immediate, load address, move register, and complement register).

A.11.1 No-op (nop)

Many instructions can be coded so that, effectively, no operation is performed. A mnemonic is provided for the preferred form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the following:

nop	equivalent to	**ori 0,0,0**

A.11.2 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 **r**D,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 **r**D,value equivalent to **addis** **r**D,**0,**value

A.11.3 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*

A.11.4 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

A.11.5 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

A.11.6 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

A.11.7 Sync (sync)

The **sync** extended mnemonics provide simpler mnemonics for specifying certain **sync** operations:

Elemental sync **esync** E	equivalent to	**sync** *x*,E Note: *x* should be set by the assembler to the complement of bit 2 of the 4-bit E (bits 0 to 3) field.

Lightweight sync		
lwsync	equivalent to	**sync 1** **sync 1,0**
Heavyweight sync		
hwsync	equivalent to	**sync 0** **sync 0,0**
Book E / PowerPC compatibility		
sync	equivalent to	**sync 0** **sync 0,0**
msync	equivalent to	**sync 0** **sync 0,0**

A.11.8 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**

A.11.9 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

Appendix B
Revision History

This is the initial revision of the document.